

# A Scalable Logical Framework

Florian Rabe

Cumulative Habilitation

Submitted: 05.06.2012

Amended: 13.06.2013

Jacobs University Bremen  
School of Engineering and Science

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                            | <b>1</b>  |
| 1.1      | Academic Works . . . . .                       | 1         |
| 1.2      | Overview . . . . .                             | 1         |
| <b>2</b> | <b>Research Summary</b>                        | <b>7</b>  |
| 2.1      | Theoretical Foundations . . . . .              | 7         |
| 2.1.1    | Logical Frameworks . . . . .                   | 7         |
| 2.1.2    | Individual Logics . . . . .                    | 10        |
| 2.2      | The MMT Framework . . . . .                    | 10        |
| 2.2.1    | The MMT Language . . . . .                     | 10        |
| 2.2.2    | The MMT System . . . . .                       | 13        |
| 2.3      | Towards MMT-based System Integration . . . . . | 16        |
| 2.3.1    | Integrating Logical Frameworks . . . . .       | 17        |
| 2.3.2    | Individual Integration Problems . . . . .      | 18        |
| 2.4      | A Library of Logics in MMT . . . . .           | 19        |
| 2.4.1    | The LATIN Atlas . . . . .                      | 19        |
| 2.4.2    | Individual Representations . . . . .           | 21        |
| 2.5      | Conclusion . . . . .                           | 22        |
| <b>3</b> | <b>Bibliography</b>                            | <b>22</b> |
| <b>A</b> | <b>Academic Works</b>                          | <b>31</b> |

# Chapter 1

## Introduction

### 1.1 Academic Works

This cumulative habilitation consists of 24 research articles and of 3 software systems as listed in Fig. 1.1 and 1.2.

All articles have passed a peer review-based quality control process and have appeared in journals (8 articles totaling 345 pages) or in archival proceedings of international meetings (16 articles totaling 240 pages). All articles are included in the habilitation documents.

Out of the 3 software systems, (5)[Rab13b] and (6)[RS09b] are tools that are represented by system descriptions. (7)[KMR09] is a set of formalizations that are available online and browsable via the MMT web server. (The bibliography entry refers to the project homepage.)

Sect. 2 summarizes the research underlying the habilitation and describes each academic work's contribution.<sup>1</sup>

### 1.2 Overview

**On Myself** I am a researcher and teacher in computer science specializing in the formal languages used in computer science, logic, and mathematics. These include in particular logics, set theories, type theories, category theory, and specification and programming languages; their proof and model theoretical semantics; and translations between them. I apply my research to the design, representation, and analysis of these systems as well as to the integration and management of the knowledge expressed in them.

I believe that, even though it is one of the classical fields, the field of formal logic will be a central force in the future of computer science and be strongly influential on mathematics and engineering throughout the 21st century. I hold that several advancements will be crucial for this development: logic-based systems must become *scalable* to a degree far beyond the current state of the art; they must be *interoperable* across

---

<sup>1</sup>Throughout this text, citations to the academic works are marked as, e.g., (1a)[Rab13a], where the first part refers to the lists in Fig. 1.1 and 1.2 and the second part to the bibliography.

| Number                     | Title   |
|----------------------------|---|
| Section 2.1                | Theoretical Foundations   |
| (1a)[Rab13a]               | A Logical Framework Combining Model and Proof Theory                                |
| (1b)[AR11]                 | Kripke Semantics for Martin-Löf's Extensional Type Theory                           |
| (1c)[Rab06]                | First-Order Logic with Dependent Types  |
| (1d)[SR09]                 | Translating Dependently-Typed Logic to First-Order Logic                            |
| (1e)[RPSS07]               | Solving the \$100 Modal Logic Challenge   |
| (1f)[KR12]                 | Semantics of OpenMath and MathML3   |
| Section 2.2                | The MMT Framework   |
| (2a)[RK13]                 | A Scalable Module System  |
| (2b)[KMR08]                | Notations for Living Mathematical Documents   |
| (2c)[GLR09]                | Integrating Web Services into Active Mathematical Documents                         |
| (2d)[ZKR10]                | A [insert XML Format] Database for [insert cool application]                        |
| (2e)[KRZ10]                | Towards MKM in the Large: Modular Representation and Scalable Software Architecture |
| (2f)[HIJ <sup>+</sup> 11]  | Combining Source, Content, Presentation, Narration, and Relational Representation   |
| (2g)[IR12]                 | Management of Change in Declarative Languages                                       |
| (2h)[Rab12]                | A Query Language for Formal Mathematical Libraries                                  |
| (2i)[HKR12]                | Extending MKM Formats at the Statement Level  |
| Section 2.3                | Towards MMT-based System Integration  |
| (3a)[Rab10]                | Representing Isabelle in LF   |
| (3b)[CHK <sup>+</sup> 12a] | A Proof Theoretic Interpretation of Model Theoretic Hiding                          |
| (3c)[RKS11]                | A Foundational View on Integration Problems   |
| (3d)[CHK <sup>+</sup> 12b] | Towards Logical Frameworks in the Heterogeneous Tool Set Hets                       |
| Section 2.4                | A Library of Logics in MMT  |
| (4a)[CHK <sup>+</sup> 11]  | Project Abstract: Logic Atlas and Integrator (LATIN)                                |
| (4b)[HR11]                 | Representing Model Theory in a Type-Theoretical Logical Framework                   |
| (4c)[IR11]                 | Formalizing Foundations of Mathematics  |
| (4d)[GMd <sup>+</sup> 07]  | An Institutional View on Categorical Logic  |
| (4e)[BRS08]                | THF0 – The core of the TPTP Language for Higher-Order Logic                         |

Figure 1.1: List of Academic Works: Research Articles by Topic

formal systems and across implementations; and they must permit far more sophisticated *user interaction* with our systems to attract users. I am convinced that these goals require some fundamentally novel perspectives on logic research, for which I have developed the theoretical and practical foundations.

My research vision is to structure, implement, and investigate logics and related formal systems in a comprehensive logical framework. The two dominating principles my framework develops are *little logics* and *foundation-independence*. *Little logics* means breaking down formal systems into the smallest independently meaningful components. Then, just like mathematicians routinely prove every theorem in the weakest theory in which it is provable, I can formulate and implement meta-logical theorems

| Number      | Title                   |
|-------------|-------------------------|
| (5)[Rab13b] | The MMT System          |
| (6)[RS09b]  | The Twelf Module System |
| (7)[KMR09]  | The LATIN Atlas         |

Figure 1.2: List of Academic Works: Software

and algorithms in the weakest possible logic. *Foundation-independence* means staying maximally neutral towards any particular ontological assumptions such as the choice of logic, type system, or mathematical foundation.

Both principles share a central goal: They maximize the reusability of individual theorems and implementations. This pays off especially at large scales where computer support permits developing major results generically and apply them to many logics. These include both logical results as well as results in knowledge management. In fact, much of my work can be seen as pioneering the field of *logical knowledge management*, which aims at combining the formal semantics of logical systems with scalable knowledge representation infrastructures.

This objective is formidable due to the enormous ontological and methodological differences separating the fields and communities. Even within logic, we find the divisions between platonic set theory and formalist type theory or between proof theoretical and model theoretical approaches, let alone philosophical differences such as intuitionism and predicativism. Similarly, the methods developed in the young field of mathematical knowledge management cannot be readily be applied to logics because they do not adequately take the logical semantics into account. However, this is all the more worthwhile because all these formal systems are ultimately motivated by similar applications – be it reasoning about numbers or verifying a piece of software – and have evolved from common roots. Therefore, our intuition compels us to seek their integration.

Consequently, my research draws from very different areas with often disjoint research communities: For example, I have to be as fluent in constructive proof theory as in algebraic development techniques as in knowledge representation languages. Similarly, my research ranges from theoretical foundations to practical implementations. Moving between and contributing to these often conflicting research areas, I experience a highly productive tension that constantly challenges and drives my research.

**On My Research** Since I started my PhD studies in 2005, I have pursued a number of coherent research strands to realize the above vision of a comprehensive logical framework. The central motivation driving my design is that many features can be realized foundation-independently. This permits a separation of concerns, which frees valuable resources that are currently expended to duplicate efforts for each formal system and for each implementation.

My framework abstracts from individual formal systems by defining them in a uniform meta-formalism (1a)[Rab13a]. The latter arises as the unification of the model and the proof theoretic approaches to defining the semantics of formal languages. I achieved

this by integrating the logical frameworks of institutions [GB92], which is based on categorical model theory, and of LF [HHP93], which is based on dependent type theory, into a coherent framework *LFI* that maintains the advantages of either one.

Concretely, *LFI* employs the *logics as theories* paradigm, where I give fully formal definitions of the syntax, proof theory, and model theory of formal systems as theories of the framework – an extensive example is given in (4b)[HR11]. In particular, the respective foundational assumptions needed to define the semantics of a formal system, such as axiomatic set theory or higher-order logic, are explicated in special theories of the framework themselves – extensive examples are given in (4c)[IR11]. Together, this permits reconciling the philosophical positions of formalism and platonism by giving a formalist account of model theory inspired by categorical logic [Law63].

The group of articles consisting of (1a)[Rab13a], (4b)[HR11], and (4c)[IR11] form a self-contained development that covers all theoretical aspects of *LFI* and of logic representations in *LFI*.

A key observation in this work was that many logical concepts can be defined and implemented at an even more abstract level than provided by a logical framework like *LFI*. Therefore, I designed the MMT language (Module system for Mathematical Theories) (2a)[RK13], uses a generic notion of a declarative language that subsumes virtually all formal systems. In particular, logical frameworks like *LFI*, logics defined in it, and the theories of these logics can be represented in MMT uniformly.

Despite its generality, many logical concepts become clearer and can be studied better in the abstract setting of MMT than in a specific logical framework, let alone in a specific logic. For example, MMT can precisely capture notions like categories of theories, proof objects, or functorial models. Moreover, it provides a module system that equips any formal system with a large-scale structuring mechanism for building large theories, translations, and models out of little components. This way I can factor out foundational assumptions such as classical, extensional, and impredicative reasoning into separate theories that are included only optionally. The same module system can be used to represent logical frameworks modularly as well so that I can develop, e.g., extensions and variants of *LFI* easily.

I leverage this framework in an MMT-based infrastructure for logical knowledge management. In particular, I can obtain substantial tool support at the MMT level, which be instantiated for any logical framework represented in *LFI*. This infrastructure is centered around two major implementations.

*Firstly*, the MMT API (5)[Rab13b] uses a systematically foundation-independent design to maximize reuse and interoperability. Specific features include an MMT-aware database (2e)[KRZ10] with fast hierachic and relational access; a generic project format for archival and build processes (2f)[HIJ<sup>+</sup>11]; a notation language and rendering engine for custom serializations (2b)[KMR08]; an interactive web server with corresponding JavaScript bindings (2c)[GLR09]; a general query language for mathematics (2g)[IR12]; and change management support (2h)[Rab12].

Because the underlying foundational assumptions of specific formal systems are represented explicitly as MMT-theories, their syntax and semantics stay transparent to the foundation-independent API services. Foundation-specific knowledge can be

injected into these services by

- giving plugins that implement the inference rules of a formal system directly, e.g., MMT includes a plugin for the dependent type theory LF [HHP93],
- formally defining the formal system in a logical framework (that is itself represented as an MMT-theory), e.g., (4c)[IR11] defines ZF set theory [Zer08, Fra22] in LF, or
- by using external services that transform custom representation formats into MMT, e.g., I have developed such imports for the LF implementation Twelf [PS99], the Mizar language for formalized mathematics [TB85], the ATP interface language TPTP [SS98], and the ontology language OWL [W3C09].

*Secondly*, I redesigned the basic data structures of the Twelf [PS99] implementation of LF to extend it with an MMT-compatible module system (6)[RS09b]. Because *LFI* makes use of LF, the modular Twelf provides advanced tool support for writing logic representations in MMT/*LFI*. Moreover, Twelf is coupled with the MMT API, via which it can be integrated with other tools. In particular, (3d)[CHK<sup>+</sup>12b] makes the logics and logic translations defined in Twelf available to the Hets [MML07] implementation of institutions, which forms the practical incarnation of the theory of *LFI*.

Despite the increased generality, MMT actually makes these implementations easier because the MMT perspective permits exposing exactly the foundation-specific aspects that are relevant to the task at hand (even to the extent that students can implement services for MMT that only experts could implement for a specific logic).

I have applied all of the above in the LATIN project (4a)[CHK<sup>+</sup>11] to obtain a highly modular library of representations of formal systems, their semantics, and their interrelations (7)[KMR09]. These representations are written in modular Twelf (6)[RS09b], their semantics is defined by the *LFI* logical framework (1a)[Rab13a], and the MMT language (2a)[RK13] and the MMT API (5)[Rab13b] tie them together on the theoretical and practical level, respectively.

This library includes formalizations of logics, set theories, type theories, and category theory, covering syntax, proof theory, and model theory, as well as translations between them, some of which are described in (4b)[HR11] (4c)[IR11] (4d)[GMd<sup>+</sup>07] (4e)[BRS08] and [DHS09, Soj10]. All formalizations are highly structured into interrelated, reusable components; these encapsulate individual ontological fundamentals such as the nature of propositions or terms as well as particular features such as conjunction or  $\lambda$ -abstraction. Overall, these components span over 1000 LF modules.



# Chapter 2

## Research Summary

### 2.1 Theoretical Foundations

This section summarizes the articles (1a)-(1f), which investigate the theoretical foundations of logical frameworks and of some individual logics.

The central result is (1a)[Rab13a], which develops my logical framework *LFI*. It is summarized in Sect. 2.1.1.

(1a)[Rab13a] is the result of a series of loosely related investigations that I conducted in the general area of formal systems. These provided the necessary overview and expertise required to design a good logical framework. Some of these investigations form independent contributions and have been published separately. These publications form the remaining articles in this group and are summarized Sect. 2.1.2.

#### 2.1.1 Logical Frameworks

Since the foundational crisis of mathematics in the early 20<sup>th</sup> century, logic has been an important research topic in mathematics and later in computer science. A central issue has always been what a logic actually is (let alone a logic translation). Over the course of the last hundred years researchers have provided very different answers to this question; for example, even in 2005 the World Congress on Universal Logic held a contest about what a logic is [Béz05]. Research areas that were initially connected have diverged and evolved into separate fields, and while this specialization has led to very successful results, it has also created divisions in the research on logic that are sometimes detrimental.

In response to these divisions, logical frameworks have been introduced. They unify different logical formalisms by representing them in a fixed meta-language. Logical frameworks have been used successfully for logic-independent investigations both on the theoretical (e.g., in the textbook on institution theory [Dia08]) and on the practical level (e.g., in the proof assistant Isabelle [Pau94]). However, today we observe that one division remains and that there are two groups of logical frameworks: model theoretical and proof theoretical ones.

**Model theoretical frameworks** are based on set theoretical (such as Zermelo-Fraenkel set theoretical [Zer08, Fra22]) or category theoretical [Mac98] foundations of

mathematics and characterize logics model theoretically in a way that goes back to Tarski’s view of logical consequence [Tar33, TV56, Rob50]. The central concept is that of a **model**, which interprets the non-logical symbols, and the **satisfaction** relation between formulas and models. Often category theory and initial models [GTW78] are employed to study the model classes. The most important such framework is that of institutions [GB92].

**Proof theoretical frameworks** are based on type theoretical foundations of mathematics (such as the principia [WR13] or simple type theory [Chu40]) and characterize logics proof theoretically (e.g., [Göd30, Gen34]) in a way most prominently expressed in Hilbert’s program [Hil26]. The central concept is that of a **proof**, which derives valid **judgments** from axioms via inference rules. Often the Curry-Howard correspondence [CF58, How80] is employed to represent proofs as expressions of a formal language. The most important such frameworks are Automath [dB70], Isabelle [Pau94], and the Edinburgh Logical Framework (LF [HHP93]).

While some of these frameworks integrate the other side, such as the institution-based general logics developed in [Mes89], almost all of them lean towards either model or proof theory. Often these sides are divided not only by research questions but also by “conflicting cultures and attitudes”, and “attempts to bridge these two cultures are rare and rather timid” (quoting an anonymous reviewer).

(1a)[Rab13a] makes one such attempt, trying to provide a balanced framework that integrates and subsumes both views on logic in a way that preserves and exploits their respective advantages. This unified framework *LFI* picks one of the most successful frameworks from each side and combines them: institutions and LF.

**Institutions** provide an abstract definition of the syntax and the model theoretical semantics of a logic. Research on institutions focuses on signatures, sentences, models, and satisfaction but deemphasizes proofs and derivability. Among the most characteristic features of the framework of institutions are: (i) It abstracts from the syntax of formulas and only assumes an abstract set of sentences; similarly, it uses an abstract class of models and an abstract satisfaction relation. (ii) Using a Galois connection between sentences and models, it permits viewing the relation between syntax and (model theoretical) semantics as a pair of adjoint functors [Law69]. (iii) Using category theory [Mac98], it provides an abstract notion of translations, both of translations within a logic – called signature or theory morphisms – and of translations between logics [GR02, MML07] – called institution (co)morphisms.

**LF** is a dependent type theory related to Martin-Löf’s type theory [ML74] featuring kinded type families and dependent function types. Research on logic encodings in LF focuses on syntax, proofs, and provability and dually deemphasizes models and satisfaction. Among the most characteristic features of this framework are: (i) It represents the logical and non-logical symbols as constants of the type theory using higher-order abstract syntax to represent binders. (ii) Using the judgments-as-types paradigm [ML96], it permits viewing syntax and (proof theoretical) semantics via judgments and inference systems and thus without appealing to a foundation of mathematics. (iii) Using a constructive point of view, translation functions are given as provably terminating programs (e.g., in [NSM01]).

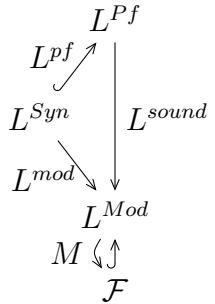
These two frameworks have been strongly influenced by the respective approaches, and this has created complementary strengths and weaknesses. For example, the bias towards an abstraction from the syntax exhibited by institutions is very useful for a logic-independent meta-logical analysis [Dia08]. But LF’s bias towards concrete syntax excels when defining individual logics and extracting programs from individual translations [Pfe01]. Therefore, (1a)[Rab13a] focuses on combining the respective strengths and motivations: While it is technically formulated within institution theory, it inherits a strictly formalist justification from LF.

(1a)[Rab13a] first extends institutions with an abstract notion of proof theory arriving at a formal definition of *logics*. These logics are defined in the spirit of institutions, in particular retaining the abstraction from concrete syntax. Then it defines a specific logic *LFI* based on LF that serves as a meta-logic. The central idea is that a theory of *LFI* captures syntax, proof theory, and model theory of an object logic. Similarly, *LFI*-theory morphisms capture translations between object logics, including the translations of syntax, proof theory, and model theory. *LFI* is defined in the spirit of LF, in particular retaining the focus on concrete syntax. Consequently, judgments are represented as types and proofs as terms. Moreover – inspired by the ideas of Lawvere [Law63] – models are represented as theory morphisms from the logical theory into a theory representing the foundation of mathematics.

Here the term “foundation of mathematics” refers to a fixed language in which mathematical objects are expressed. While mathematicians usually do not explicate a foundation (often implicitly assuming a variant of first-order set theory as the foundation), the choice of foundation is more difficult in computational logic, where for example higher-order logic [Chu40] and the calculus of constructions [CH88] are often used instead. By representing the foundation as a theory itself, *LFI* can formulate model theory without committing to a specific foundation.

Thus, *LFI* reconciles the type/proof and the set/model theoretical perspective: platonic, institution-based logics are expressed using the syntax of a formalist LF-based meta-logic.

In *LFI*, logics are represented as theories and translations as theory morphisms in the category of LF-theories. Logic representations formalize the syntax, proof theory, and model theory of a logic as separate theories as indicated on the right. The syntax of a logic  $L$  is represented as a theory  $L^{Syn}$ , which is then extended with the representation of proof rules to represent the proof theory as  $L^{Pf}$ . The model theory of the logic is represented as a theory  $L^{Mod}$  as an extension of the representation  $\mathcal{F}$  of a foundation of mathematics. Then individual models are represented as theory morphisms  $M$  into the foundation. Moreover, we can represent soundness proofs as a morphism  $L^{sound}$  from the proof theory to the model theory of  $L$ . Similarly, logic translations are represented as triples of theory morphisms that formalize the translations of syntax, proof theory, and model theory.



## 2.1.2 Individual Logics

The remaining articles in this group consider various aspects of individual formal systems including logics, type theories, and category theory. They include proof theoretical (1e)[RPSS07] and model theoretical (1b)[AR11] (1f)[KR12] semantics, soundness and completeness results (1b)[AR11] (1c)[Rab06], the design of new formal systems (1c)[Rab06], representation theorems relating formal systems (1d)[SR09] (1e)[RPSS07], as well as representation languages (1f)[KR12] for them.

(1b)[AR11] establishes a coherent sound and complete denotational semantics of dependent type theory. The key idea is to interpret dependent type theory in the locally cartesian closed category of presheaves on a poset.

(1c)[Rab06] develops a new logic for dependently-typed first-order logic (DFOL). It permits natural representations of many important data structures and was used in particular for the representation of categorical logics in (4d)[GMd<sup>+</sup>07].

(1d)[SR09] establishes a model theoretical logic translation from DFOL to first-order logic.

(1e)[RPSS07] provides a solution to a challenge problem for deduction systems that asks for automatically (dis)proving the subset relation between modal logics. It uses first-order logic as a framework for the representation of propositional modal logics.

(1f)[KR12] introduces a role system and a model theoretical semantics for OPEN-MATH [BCC<sup>+</sup>04], a formal language that features centrally in the representation of mathematical formulas in MMT.

## 2.2 The MMT Framework

The MMT language and system form the pivotal aspect of my research. In particular, MMT provides the abstract concepts, concrete syntax, and tool support for logic representations in *LFI* (1a)[Rab13a].

The research on MMT comprises the articles (2a)-(2i) as well as the MMT software system (5)[Rab13b]. It ranges from purely theoretical work that introduces and analyzes the MMT language to the design and implementation of the scalable infrastructure supplementing it.

The central article is (2a)[RK13], which gives a comprehensive overview of modular knowledge representation languages and then introduces and studies the MMT language. A small extension was given in (2i)[HKR12]. These results are summarized in Sect. 2.2.1.

The remaining articles (2b)-(2h) develop an MMT-based infrastructure for scalable logical knowledge management. Each article is self-contained and devoted to one particular research problem and its solution in MMT. All solutions are implemented coherently in the MMT system (5)[Rab13b]. The design and realization of the MMT system are summarized in Sect. 2.2.2.

### 2.2.1 The MMT Language

Mathematical and logical knowledge has become far too vast to be understood by one person – it has been estimated that the total amount of published mathematics doubles

every ten to fifteen years [Odl95].

This leads to increasing specialization and missed opportunities for knowledge transfer, and the question of supporting the management and dissemination of mathematical knowledge in the large remains difficult. This problem has been tackled in the field of mathematical knowledge management (MKM), which uses explicitly annotated content as the basis for mathematical software services such as semantics-based searching and navigation.

MKM in the large has been pioneered in the field of formal methods, where a sound logical foundation and the incorruptibility of computers are combined to verify computer systems, e.g., in the verification of L4 microkernel [KAE<sup>+</sup>10]. The field of formalized mathematics applies the same techniques to obtain computer-verified mathematics, e.g., in the formal proof of the Kepler conjecture [Hal05]. These computer-aided proofs rely on large amounts of formal knowledge about the logics, programming language constructs, and data structures, and the productivity of these methods is restricted in practice by the effectiveness of managing this knowledge.

There are currently five obstacles for large scale computerized MKM:

**Formalization** As computer programs still lack any real understanding of mathematics, human mathematicians must make structures in mathematical knowledge sufficiently explicit. This usually means that the knowledge has to be formalized, i.e., represented in a formal, logical system. While it is generally assumed that all mathematical knowledge can in principle be formalized, this is so expensive that it is seldom even attempted.

**Logical Heterogeneity** One of the advantages of informal-but-rigorous mathematics is that it does not force the choice of a formal system. There are many formal systems, each optimized for expressing and reasoning about different aspects of mathematical knowledge. All attempts to find the “mother of all logical systems” (and convince others to use it) have failed, e.g., the qed project [Ano94]. Even though logics themselves can be made the objects of mathematical investigation and even of formalization (in logical frameworks), we do not have scalable methods for efficiently dealing with heterogeneous, i.e., multi-logic, presentations of mathematical knowledge.

**Foundational Assumptions** Logical heterogeneity is not only a matter of optimization because different developments of mathematical knowledge make different foundational assumptions. The vast majority of classical mathematics has been formulated in axiomatic set theory following a platonist philosophy. And technical differences between set theories, like Zermelo-Fraenkel or Gödel-Bernays often do not matter. But there are other foundations such as those rejecting the axiom of choice, impredicative definitions, or the principle of excluded middle. Corresponding developments often take a more formalist stance and use different foundations such as higher-order logic or constructive type theory. These alternative foundations have become important in computer-supported verification, where almost all systems use foundations that are different from each other and different from classical set theory.

**Modularity** Modern developments of mathematical knowledge are highly modular. They take pains to identify minimal sets of assumptions so that results are applicable at the most general possible level. This modularity and the mathematical practice of

“framing”, i.e., of viewing objects of interest in terms of already understood structures, must be supported to even approach human capabilities of managing mathematical knowledge in computer systems.

**Global Scale** Mathematical research and applications are distributed globally, and mathematical knowledge is highly interlinked by explicit and implicit references. Therefore, a computer-supported management system for mathematical knowledge must support global interlinking, and management algorithms have to scale up to large (global) data sets.

The MMT language constitutes a uniform solution to four of the five challenges (all but formalization): It is a globally scalable module system for mathematical theories that abstracts from and mediates between different logics and foundations. Thus, it provides a conceptual and technical foundation for formal MKM in the large.

Modularity and Global Scale are realized by the MMT module system. It integrates successful features of existing paradigms

- reuse along theory morphisms from the “little theories” approach [FGT92],
- the theory graph abstraction from algebraic specification languages like [SW83],
- categories of theories and logics from model theoretical logical frameworks like institutions [GB92],
- the logics-as-theories representation from proof theoretical logical frameworks like LF [HHP93],
- the Curry-Howard correspondence from type/proof theory [CF58, How80],
- URIs as logical namespace identifiers from OPENMATH [BCC<sup>+</sup>04],
- standardized XML-based concrete syntax from web-oriented representation languages like OMDOC [Koh06],

and makes them available in a single, coherent representational system for the first time. The combination of these features is reduced to a small set of carefully chosen, orthogonal primitives in order to obtain a simple and extensible language design.

The module system is tightly coupled with the use of URIs as identifiers – a requirement that imposes so subtle constraints that it is difficult to meet a posteriori. In MMT, all knowledge items, including the virtual ones that only arise by reuse along morphisms have canonical, globally unique URIs. These are tripartite URIs  $ns?mod?sym$  formed from a namespace URI  $ns$ , a module name  $mod$ , and a qualified symbol name  $sym$ .

To facilitate distributing MMT content, all MMT URIs are logical identifiers, and their physical locations (URLs) remain transparent to the MMT language: the relation between logical URIs and physical URL is delegated to an extra-linguistic catalog.

Foundation-independence and heterogeneity are achieved by representing all logics, logical frameworks, and the foundational languages themselves simply as MMT theories. Thus, they are subject to the MMT module system themselves so that we can use sharing and reuse at the level of logics, logical frameworks, and foundation.

The rigorous semantics of the module system is contrasted by a flexible semantics at the level of mathematical expressions. Crucially, MMT does not prescribe a set of well-typed expressions, which would preclude foundation-independence. Instead, MMT uses generic term formation operators and is parametric in the typing relation.

A key innovation of MMT is the formal treatment of meta-theories. Let us write

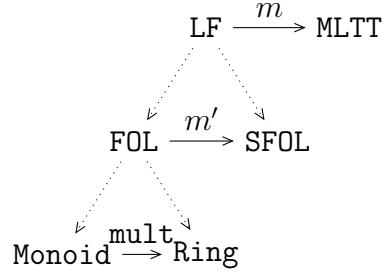
$M/T$  to express that we work in the object language  $T$  using the meta-language  $M$ . For example, most of mathematics is carried out in  $\text{FOL}/\text{ZFC}$ , i.e., first-order logic is the meta-language, in which set theory is defined.  $\text{FOL}$  itself might be defined in a logical framework such as  $\text{LF}$  [HHP93], and within  $\text{ZFC}$ , we can define the language of natural numbers, which yields  $\text{LF}/\text{FOL}/\text{ZFC}/\text{Nat}$ . In MMT, all of these languages are represented as theories with a binary meta-theory relation between them.

In the example on the right, the meta-theory relation is visualized using dotted arrows. The theory  $\text{FOL}$  for first-order logic is the meta-theory for **Monoid** and **Ring**.  $m'$  is a logic translation from  $\text{FOL}$  to sorted first-order logic  $\text{SFOL}$ . And the theory  $\text{LF}$  for the logical framework  $\text{LF}$  is the meta-theory of  $\text{FOL}$  and  $\text{SFOL}$ . Crucially, the meta-theory indicates both to humans and to machines how a theory is to be understood. For example, interpretations of  $\text{ZFC}$  must understand  $\text{FOL}$ , and the typing relation of  $\text{FOL}$  is inherited from  $\text{LF}$ .

In particular, all representations of logics and logic translations based on  $\text{LFI}$  can be expressed directly as MMT diagrams in which  $\text{LF}$  is the highest meta-theory. Thus, MMT operationalizes  $\text{LFI}$  and serves as a knowledge management interface to it.

Moreover, the foundation-independence of MMT lets us reuse the theoretical and practical aspects of the MMT infrastructure when generalizing  $\text{LFI}$ . For example, if we want to build a variant of  $\text{LFI}$  based on Martin-Löf type theory  $\text{MLTT}$  [ML74] instead of  $\text{LF}$ , we can give a translation  $m$ , and MMT can transfer all logic definitions seamlessly.

We can see MMT as the next step in a historical progression towards more abstract framework languages as indicated in Fig. 2.1. In traditional mathematics, domain knowledge is expressed directly in ad hoc formal notation. Starting in the 20th century, logic provided a formal syntax and semantics for expressing this knowledge. Starting in the 1970s, logical frameworks provided an additional level of abstraction, at which individuals logics could be treated uniformly. Now MMT takes this one step further by providing a meta-level, at which even different logical frameworks can be represented.



| Mathematics      | Logic                     | Logical Frameworks                             | MMT   |
|------------------|---------------------------|--|---|
| domain knowledge | logic<br>domain knowledge | logical framework<br>logic<br>domain knowledge | MMT framework<br>logical framework<br>logic<br>domain knowledge |

Figure 2.1: The Progression of Meta-Languages

## 2.2.2 The MMT System

Exploiting the small number of primitives in MMT, the MMT API provides a comprehensive, scalable implementation of MMT itself and of MMT-based knowledge management

services. All algorithms are implemented generically, and all logic-specific aspects are relegated to thin plugin interfaces.

It is written in the functional and object-oriented language Scala [OSV07]. Scala is fully compatible with Java so that API plugins and programs using the API can be written in any combination of Scala and Java. The API depends on only one external library – the lean web server tiscaf [Gay08]. Excluding plugins and libraries, it comprises over 20000 lines of Scala code compiling into about 3000 Java `class` files totaling about 5 MB of platform-independent bytecode. Sources, binaries, API documentation, and user manual are available at <http://trac.kwarc.info/MMT>.

**Knowledge Management Services** The MMT API provides a suite of coherently integrated MKM services.

MMT content is organized in MMT archives (2f)[HIJ<sup>+</sup>11], a light-weight project abstraction that integrates the representational dimensions of source files, content structure, narrative document structure, presentation, and semantic web-ready relational indices.

Source files form the original representation, which is usually subject to change by human editing. When importing sources into MMT, it splits them into content and narrative structure, which are given concretely as XML markup based on the OMDOC language [Koh06]. The former the semantically relevant information such as definitions and axioms; the latter contains the various possible linearized narrative structure, which can include connecting text and cross-references.

Both are used to generate presentation, which is dual to source in that it is tailored towards reading rather than editing, e.g., by using HTML with embedded presentation MathML [ABC<sup>+</sup>03]. The generation of presentation employs a notation language based on (2b)[KMR08]. The notations are grouped into styles, and the MMT a presentation engine serializes any MMT into arbitrary output formats according to the chosen style.

The crucial feature of MMT archives is that all representational dimensions share the MMT URIs, which permits concise cross-referencing. For example, MMT URIs are used for fine-granular parallel markup, i.e., every nodes in the presentation tree point to the corresponding content node, which in turn points to the corresponding point in the source file.

The relational index uses the MMT URIs as the individuals in an MMT ontology. A query language and evaluation engine (2h)[Rab12] integrates semantic web-style relational queries with hierarchical queries based on the content structure and unification-based queries provided by [KS06, IK12].

Moreover, the canonical URI feature centrally a change management infrastructure (2g)[IR12]: They permit the fine-granular representation of changes to MMT content. The MMT change management engine detects and propagates such changes.

**User and System Interfaces** The API can be run as an application, in which case it responds with a shell that interacts via standard input/output. The shell is scriptable in the sense that sequences of commands can be read from files, which also provides an easy way for users to initialize and configure the system.

The main frontend is given by an HTTP server. For machine interaction, it exposes all API functionality including services such as querying and presentation. For human interaction, it offers an interactive web browser based on HTML+presentation MathML generated by the presentation engine. Based on the JOBAD JavaScript library (2c)[GLR09] and MMT-specific JavaScript bindings, user interaction is handled via JavaScript and Ajax. The parallel markup between presentation, content, and source permits sophisticated interface functions, e.g., folding subexpressions, hiding/showing inferred types, implicit arguments, and redundant brackets, retrieving the definition of a symbol, or dynamically inferring the type of a subexpression.

The MMT API implements a catalog that maps MMT URIs to their physical locations. The catalog also acts as an abstraction over various backends that physically store MMT archives. MMT knowledge items are retrieved from these backends and loaded into memory transparently when needed. Thus, memory constraints are hidden from high-level knowledge management services and users.

Supported backends are file systems, SVN working copies and SVN repositories, and TNTBASE databases [ZK09]. In particular, the latter also supports MMT-specific indexing and querying functions (2d)[ZKR10], which permit, e.g., the efficient retrieval of the dependency closure of an MMT knowledge item. The combination of MMT module system and MMT-aware TNTBASE (2e)[KRZ10] received the MKM 2010 Best Paper Award.

**Instantiating MMT for Specific Languages** MMT archives are typically written in one fixed meta-theory (which is itself defined in a different archive). For example, the LATIN archive is written using the meta-theory LF. All API functionality is foundation-independent, but foundation-specific knowledge can be provided via one of two plugin interfaces: these provide information about the syntax or the semantics of the meta-theory.

Syntax plugins provide parsing methods that translate non-MMT source syntax into MMT content markup. This is crucial to turn existing non-MMT knowledge repositories into MMT archives. Such plugins have been developed for a varied list of languages: the ATP interface language TPTP [SS98], the ontology language OWL [W3C09], the Mizar language for formalized mathematics [TB85], and the Twelf implementation [PS99] of LF. In particular, these have been used to import the TPTP library and the Mizar library [IKRU13].

Semantics plugins provide implementations of the typing relation, in which the MMT language is parametric. Here the MMT module system remains transparent for the plugin so that only the core type checking algorithms have to be implemented. For example, the semantics plugin for LF comprises only 200 lines of code. Moreover, for all logics defined in *LFI*, no additional semantics plugin is required because their typing relations can be inherited from LF.

**An Example from The LATIN Library** Fig. 2.2 shows a screenshot of the MMT web server displaying a part of the LATIN archive (see Sect. 2.4.1), a collection of logic formalizations in *LFI*. This MMT archive is written in Twelf syntax and uses the Twelf

The screenshot shows a code editor on the left with the following OMDoc content:

```

document derived.omdoc
  remote module FalsityExt
  remote module NEGExt
  theory IMPExt meta lf
    include IMP
    imp2I : ((ded A → ded B → ded C) → ded A imp (B imp C))
      = [f:ded A → ded B → ded C]impl ([p:ded A]impl ([q:ded B]f p q))
    imp2E : (ded A imp (B imp C) → ded A → ded B → ded C)
      = [p:ded A imp (B imp C)][q:ded A][r:ded B]impE (impE p q) r
  remote module CONJExt
  remote module DISJExt
  remote module Equiv

```

To the right, a modal dialog titled "type" displays the inferred type: `ded A imp (B imp C)`. A context menu with the following options is visible:

- infer type
- reconstructed types
- implicit arguments
- implicit binders
- redundant brackets
- Fold

Figure 2.2: MMT Web Interface

tool as a syntax plugin. MMT generates presentation, which can be browsed in the MMT web server.

The selected declaration derives the rule  $\frac{A, B \vdash C}{\vdash A \text{imp} (B \text{imp} C)}$  in the theory IMP of the implication connective `imp`. Via the JOBAD context menu, the user called type inference on the selected subobject, which resulted in a dialog displaying the dynamically inferred type.

The implementation of this feature uses server-side evaluation of MMT queries. JOBAD JavaScript picks up on the parallel markup annotations that carry the MMT URIs of the presented content elements. These are used to build an MMT query that is evaluated on the server and results in the rendered inferred type, which is sent back to the client and displayed. The query consists of multiple steps, which retrieve the corresponding mathematical object, extract the respective subobject (including its context), infer its type using the LF semantics plugin, and finally present it using the notation language.

Due to the genericity of MMT and the MMT API, all this functionality can be made available to individual formal systems at minimal cost, a process we call *rapid prototyping for formal systems*. Given a syntax and a semantics plugin, MMT provides the logical and the practical knowledge management infrastructure out of the box.

## 2.3 Towards MMT-based System Integration

This group of academic works contains several loosely connected investigations that share the theme of employing MMT as a platform for system integration. Different systems can use MMT as a concrete knowledge interchange language that combines simplicity, general applicability, precise semantics, and tool support.

The main contribution of this group is the use of MMT to obtain a connection between the Twelf system [PS99] – an implementation of the proof theoretical logical

framework LF – and the Hets system [MML07] – an implementation of the model theoretical logical framework of institutions. Thus, it provides the practical counterpart to the theoretical connection established by the *LFI* framework (see Sect. 2.1.1). This contribution consists of the article (3d)[CHK<sup>+</sup>12b] and the software system (6)[RS09b]. They are summarized in Sect. 2.3.1.

In addition, I conducted several theoretical explorations of advanced integration problems. These are summarized in Sect. 2.3.2.

### 2.3.1 Integrating Logical Frameworks

While (1a)[Rab13a] introduces the theoretical foundation for an integration of proof and model theoretical frameworks, (3d)[CHK<sup>+</sup>12b] develops a practical integration.

On one side, (3d)[CHK<sup>+</sup>12b] uses the Twelf [PS99] implementation of LF as a representative of the implementations of proof theoretical logical frameworks. Twelf uses LF as a universal logic in which other logics are represented, usually using higher-order abstract syntax [PE88] and the Curry-Howard correspondence [CF58, How80].

On the other side, (3d)[CHK<sup>+</sup>12b] uses Hets [MML07], an institution-independent software interface for heterogeneous (i.e., covering multiple logics) specification and proof management. Hets works relative to a graph of logics and logic translations, which connects a variety of individual tools for specific logics such as automated reasoners and model finders.

(3d)[CHK<sup>+</sup>12b] and (6)[RS09b] combine these two systems using MMT an interlingua for system integration via its concrete syntax based on OMDOC. The benefit for LF/Twelf is that logics represented in LF can be (via Hets) easily connected to various interactive and automated theorem provers, model finders, model checkers, and conservativity checkers – thus providing much more efficient proof support than mere proof checking. The benefit for institutions/Hets is that (via Twelf) logics are defined declaratively and their implementations derived uniformly instead of each logic being implemented ad hoc as a part of the Hets code base. Thus, the trustworthiness of the implementation of logics is increased, and correctness of logic translations can be mechanically verified. Moreover, the effort of adding new logics or translations to Hets is greatly reduced.

The integration proceeds in two phases. Firstly, logics and logic translations are defined using Twelf. Secondly, these logic definitions are imported into Hets.

**Twelf Side** The Twelf language is non-modular, i.e., Twelf only processes lists of LF declarations. Besides impeding reuse, this precluded the definition of *LFI*-logics, which require defining LF signatures and morphisms. Therefore, (6)[RS09b] extends Twelf to modular Twelf. This Twelf module system follows the concepts of MMT specialized to the case where all theories have meta-theory LF. In particular, it uses MMT URIs and MMT theories and theory morphisms. Thus, modular Twelf becomes an MMT/LF processor.

Modular Twelf extends the existing SML implementation of Twelf. This extension comprises about 10 main source files for maintaining modules. Moreover, it required a drastic change to the central data structures, demonstrating how difficult it is to

add a module systems a posteriori. This led to changes in almost all of the about 500 Twelf source files. The implementation of modular Twelf including source code and documentation is available at [RS09a].

Modular Twelf includes an export to the OMDOC XML language serving as the concrete syntax for the MMT content representation. This let us use modular Twelf as a syntax plugin for the MMT system and thus to use MMT to manage modular Twelf archives. Thus, MMT can make Twelf developments available to other systems that understand the meta-theory LF.

**Hets Side** Logics that are defined in *LFI* induce institutions. Correspondingly, if such logics are defined concretely in modular Twelf, they induce implementations of institutions in the Hets system. This connection is presented in (3d)[CHK<sup>+</sup>12b].

Hets reads such logic definitions represented in MMT. Hets reads the logic definition and uses the meta-theory (which is LF) to choose an LF-aware processor that generates a new instance of the Hets' internal logic class. In particular, Twelf handles the advanced type reconstruction necessary to make declarative logic definitions feasible, and Hets receives fully reconstructed definitions (which are orders of magnitude easier to process). After importing a logic in this way, Hets permits writing and managing specifications in that logic.

**Generalization** (3d)[CHK<sup>+</sup>12b] also shows how the above workflow can be generalized to integrate Hets with other logical frameworks besides LF. Specifically, it proves that the construction of *LFI* in (1a)[Rab13a], which is specific to LF, can be generalized to a large class of logical frameworks including Isabelle [Pau94] and Maude [CELM96].

As far as MMT is concerned, the only effect of the choice of framework is that a different meta-theory is used. Moreover, using theory morphisms at the meta-theory level, MMT can even support moving logic definitions between frameworks.

### 2.3.2 Individual Integration Problems

The generic MMT module system goes a long way towards scalable integration of developments in different languages and systems. If different systems implementing the same foundation export their developments into MMT, these can import and reference each other via the MMT module system. Moreover, using theory morphisms between meta-theories, developments can be connected across foundations.

Based on this idea, (3c)[RKS11] gives a conceptualization and overview of the various integration problems that still lie ahead. Most importantly, it identifies partial theory morphisms as a crucial tool for practical integrations: Partiality is useful when only some features of one system can be soundly translated into another one. For that purpose, (3c)[RKS11] introduces filtering as a form of generalized theory morphisms that support partiality.

A major hurdle to integration can arise if the respective systems already use idiosyncratic module systems. In that case, morphisms between meta-theories become substantially more complex. Here the key to practically useful integration is to give

a translation that preserves modularity. Therefore, it is desirable to use MMT as an interlingua to connect different module systems. MMT was designed to make this possible, and I conducted two case studies where two very different module systems are represented in terms of MMT.

Firstly, (3a)[Rab10] represents the Isabelle module system [Pau94] in MMT. Isabelle is one of the leading type/proof theoretical systems and uses three different module concepts: theories, type classes, and locales. (3a)[Rab10] shows that all three can be uniformly represented as MMT theories, and all relations between them can be uniformly represented as MMT theory morphisms.

Secondly, (3b)[CHK<sup>+</sup>12a] represents structured specifications [ST88] in MMT. This is the leading categorical/model theoretical module system and uses a small set of theory-building operations within the theory of institutions. We find that all but one theory-building operation can be mapped to MMT analogues naturally. Moreover, (3b)[CHK<sup>+</sup>12a] introduces the feature of hiding in MMT, which permits representing also the remaining theory-building operation.

## 2.4 A Library of Logics in MMT

The central result of this group is the LATIN atlas: a library of logic formalizations, which I developed as a part of the LATIN project (7)[KMR09]. It is an online resource that is available in two ways:

- The MMT archive containing the library is available at <http://tntbase.mathweb.org/repos/cds>. This endpoint offers access via HTTP and SVN as well as via the XQuery functionality of TNTBASE [ZK09].
- An instance of the MMT web server is hosting the archive at <http://cds.omdoc.org:8080>. This includes interactive browsing as well as some experimental features.

It is summarized in Sect. 2.4.1.

The LATIN library as a whole is one of the software pieces of the cumulative habilitation. In addition, several important individual formalizations have been published independently in research articles. These form the articles (4a) - (4e), which are summarized in Sect. 2.4.2.

### 2.4.1 The LATIN Atlas

The LATIN atlas is a library of representations of formal systems and translations in LF. It includes not only logics but also type theories, set theories, and developments in category theory. The logics are represented following the *LFI* framework, i.e., they are represented as spans of LF theories including syntax, proof theory, and model theory. Similarly, the library includes various translations between these formal systems, many of which are logic translations in the sense of the *LFI* framework.

An overview of a fragment of the library is given in Fig. 2.3. It shows a high-level logic graph on the left, then zooms into the modular definition of propositional logic

in the middle, and on the right zooms in further to show the *LFI*-style definition of conjunction as a triple of syntax, proof theory, and model theory.

All major research strands of the habilitation culminate in their relation to this library:

- The logics are defined within the *LFI* framework (1a)[Rab13a] where a formalist definition using LF induces a Platonist one using institutions.
- The formalist definitions are written using modular Twelf (6)[RS09b], which provides advanced authoring support.
- Twelf exports logic definitions using the modular representation language MMT (2a)[RK13].
- Twelf acts as a plugin to the MMT system (5)[Rab13b], which maintains the LATIN library as an MMT archive and provides its knowledge management capabilities for it.
- The Platonist definitions become available as institutions to Hets via the integration of (3d)[CHK<sup>+</sup>12b].

The formalized logics include propositional, first-order, sorted first-order, common, higher-order, modal, description, and linear logics. Their model theories can be represented in various foundations included Zermelo-Fraenkel set theory, Church's higher-order logic [Chu40], or Mizar's formalized set theory [TB85].

Formalized logic translations include the relativization translations from modal, description, and sorted first-order logic to unsorted first-order logic, the negative translation from classical to intuitionistic logic, and the translation from first to sorted first- and higher-order logic.

All representations systematically exploit modularity and form a single highly interconnected graph that currently consists of over 1000 little modules spread over about 200 files. As a general principle, every logical principle is formalized in a separate module. Such features are, for example, conjunction in propositional logic, the universal quantifier of first-order logic, the typed universal quantifier of sorted or higher-order logic, or the extensionality principle of higher-order logic.

Moreover, type theoretical features can be freely combined with logical features. For example, the library includes modules for various type constructors such as the ones in the  $\lambda$ -cube or product and union types as well as base types like booleans or natural number. Often multiple alternative formalizations are given (and related to each other),

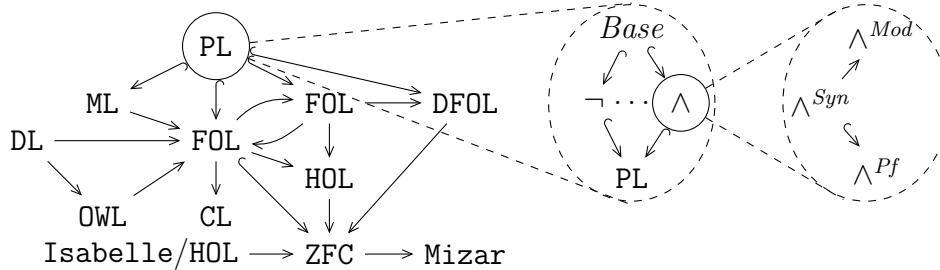


Figure 2.3: A Fragment of the LATIN Atlas

e.g., Curry- and Church-style typing, or Andrews and Prawitz-style higher-order logic.

Thus, logics can be composed by adjoining the required features using the MMT module system. For example, the intuitionistic higher-order logic underlying the Isabelle system [Pau94] can be obtained by combining the modules for Church-style typing, simple function types, a boolean type, implication, typed universal quantification, and typed equality, as well as corresponding modules for the semantics. Similarly, new logics can be added easily by reusing some feature, adding a few additional features, and possibly relating the new features to the existing ones using translations. In addition, some meta-results including logic translations and soundness proofs can be established feature-wise and composed modularly.

## 2.4.2 Individual Representations

The project description (4a)[CHK<sup>+</sup>11] gives a short overview of the LATIN project and library. This includes a high-level description of the general methodology and the various logic definitions.

(4b)[HR11] describes the formalization of first-order logic. It is written as a comprehensive case study and example of using *LFI* and MMT to define logics and to do so modularly. Specifically, it represents the syntax, proof theory, model theory and soundness of first-order logic. The representation is systematically modular, i.e., each connective is represented in a separate theory, each of which serves as a building block from which new logics can be built.

The representation of the model theory includes the set theoretical foundations, in which models are described. Concretely, it uses formalizations of higher-order logic and ZFC set theory as two possible foundations and gives a theory morphism that refines the former into the latter. This permits using the typed reasoning of higher-order logic to carry out the soundness proof and then to translate it into set theory.

(4c)[IR11] expands on the representation of foundations by describing 3 different formalizations of foundations of mathematics in LF. These can be used as the starting points of representations of the model theory of logics in *LFI*. These are my own formalization of ZFC theory that is used in (4b)[HR11] as well as two foundations of advanced proof assistants. The latter are the set theoretical foundation used in the Mizar system [TB85] and the type-theoretical foundation used in Isabelle/HOL [NPW02].

Finally, 2 peripheral articles describe very specific representations.

Firstly, (4d)[GMd<sup>+</sup>07] represents the categorical model theory of various propositional logics including modal and – most notably – linear logic. Being propositional, these representations do not have to use LF as a meta-theory but can use the much simpler DFOL logic (1c)[Rab06]. In fact, I designed DFOL for that purpose and already used LF to define it.

Secondly, (4e)[BRS08] defines higher-order logic. This formalization is conceptually straightforward but has been influential because it choose one out of many possible variants of higher-order logic as a standard variant that became part of the TPTP suite

of languages [SS98]. Moreover, it factors out various features such as product types or choice principles into separate modules. This provides system developers and users with human- and machine-accessible reference definitions of individual logics.

## 2.5 Conclusion

Logical methods – and especially so logical frameworks – pay off almost exclusively at large scales due to the high level of theoretical understanding and practical investment that they require from both developers and users. But for the same reason, it is difficult for individual approaches to reach those large scales.

For example, institutions [GB92] and LF [HHP93], have been developed in parallel in small, specialized communities of experts. Only *LFI* (1a)[Rab13a] – to be published 20 years later – unified their fundamentally different perspectives. And only this cross-fertilization between them permitted building a large library of interrelated logic formalizations (7)[KMR09].

Moreover, within those 20 years, the advances in computer and internet technology have dramatically changed our expectations regarding scalability, with collaboration, system interoperability, and modularity becoming critical factors. Therefore, methods as expensive as formal logic must maximize the reuse of all resources: humans, tools, and formalizations.

Here MMT (2a)[RK13], (5)[Rab13b] pioneers a novel design methodology for scalable logical systems. It is systematically foundation-independent, i.e., it separates large scale concerns, which are addressed generically by MMT, and small scale concerns, which are addressed by individual logics. Thus, the MMT methodology permits developers of logics to focus on the logical core of their systems instead of spending resources on ad hoc knowledge management support. Dually, knowledge management services can be developed generically at the MMT level without the currently required expertise on individual logics.

Foundation-independence makes MMT a meta-framework in the sense that even logical frameworks like *LFI* are represented as MMT theories. Besides a uniform treatment of logics and logical frameworks, this makes MMT robust against future language developments: We can develop new logical frameworks and use them to define new logics without any change to the MMT infrastructure. And MMT guarantees the theoretical and practical reusability of developments when migrating to these new languages.

# Bibliography

- [ABC<sup>+</sup>03] R. Ausbrooks, S. Buswell, D. Carlisle, S. Dalmas, S. Devitt, A. Diaz, M. Froumentin, R. Hunter, P. Ion, M. Kohlhase, R. Miner, N. Poptelier, B. Smith, N. Soiffer, R. Sutor, and S. Watt. Mathematical Markup Language (MathML) Version 2.0 (second edition), 2003. See <http://www.w3.org/TR/MathML2>.
- [Ano94] Anonymous. The QED Manifesto. In A. Bundy, editor, *Automated Deduction*, pages 238–251. Springer, 1994.
- [AR11] S. Awodey and F. Rabe. Kripke Semantics for Martin-Löf’s Extensional Type Theory. *Logical Methods in Computer Science*, 7(3), 2011.
- [BCC<sup>+</sup>04] S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See <http://www.openmath.org/standard/om20>.
- [Béz05] J. Béziau, editor. *Logica Universalis*. Birkhäuser Verlag, 2005.
- [BRS08] C. Benzmüller, F. Rabe, and G. Sutcliffe. THF0 – The core of the TPTP Language for Higher-Order Logic. In A. Armando, P. Baumgartner, and G. Dowek, editors, *4th International Joint Conference on Automated Reasoning*, pages 491–506. Springer, 2008.
- [CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4, pages 65–89, 1996.
- [CF58] H. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.
- [CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988.
- [CHK<sup>+</sup>11] M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. Project Abstract: Logic Atlas and Integrator (LATIN). In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 289–291. Springer, 2011.

- [CHK<sup>+</sup>12a] M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. A Proof Theoretic Interpretation of Model Theoretic Hiding. In T. Mossakowski and H. Kreowski, editors, *Recent Trends in Algebraic Development Techniques 2010*, pages 118–138. Springer, 2012.
- [CHK<sup>+</sup>12b] M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, F. Rabe, and K. Sojakova. Towards Logical Frameworks in the Heterogeneous Tool Set Hets. In T. Mossakowski and H. Kreowski, editors, *Recent Trends in Algebraic Development Techniques 2010*, pages 139–159. Springer, 2012.
- [Chu40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.
- [dB70] N. de Bruijn. The Mathematical Language AUTOMATH. In M. Laudet, editor, *Proceedings of the Symposium on Automated Demonstration*, volume 25 of *Lecture Notes in Mathematics*, pages 29–61. Springer, 1970.
- [DHS09] S. Dumbrava, F. Horozal, and K. Sojakova. A Case Study on Formalizing Algebra in a Module System. In F. Rabe and C. Schürmann, editors, *Workshop on Modules and Libraries for Proof Assistants*, pages 11–18. ACM, 2009.
- [Dia08] R. Diaconescu. *Institution-independent Model Theory*. Birkhäuser, 2008.
- [FGT92] W. Farmer, J. Guttman, and F. Thayer. Little Theories. In D. Kapur, editor, *Conference on Automated Deduction*, pages 467–581, 1992.
- [Fra22] A. Fraenkel. Zu den Grundlagen der Cantor-Zermeloschen Mengenlehre. *Mathematische Annalen*, 86:230–237, 1922. English title: On the Foundation of Cantor-Zermelo Set Theory.
- [Gay08] A. Gaydenko. tiscaf http server, 2008. <http://gaydenko.com/scala/tiscaf/httpd/>.
- [GB92] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.
- [Gen34] G. Gentzen. Untersuchungen über das logische Schließen. *Math. Z.*, 39, 1934. English title: Investigations into Logical Deduction.
- [GLR09] J. Gićeva, C. Lange, and F. Rabe. Integrating Web Services into Active Mathematical Documents. In J. Carette, L. Dixon, C. Sacerdoti Coen, and S. Watt, editors, *Intelligent Computer Mathematics*, pages 279–293. Springer, 2009.
- [GMd<sup>+</sup>07] J. Goguen, T. Mossakowski, V. de Paiva, F. Rabe, and L. Schröder. An Institutional View on Categorical Logic. *International Journal of Software and Informatics*, 1(1):129–152, 2007.

- [Göd30] K. Gödel. Die Vollständigkeit der Axiome des Logischen Funktionskalküls. *Monatshefte für Mathematik und Physik*, 37:349–360, 1930. English title: The Completeness of the Axioms of the Logical Calculus of Functions.
- [GR02] J. Goguen and G. Rosu. Institution morphisms. *Formal Aspects of Computing*, 13:274–307, 2002.
- [GTW78] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice Hall, 1978.
- [Hal05] T. Hales. Introduction to the Flyspeck Project. In T. Coquand, H. Lombardi, and M. Roy, editors, *Mathematics, Algorithms, Proofs*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.
- [HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
- [HIJ<sup>+</sup>11] F. Horozal, A. Iacob, C. Jucovschi, M. Kohlhase, and F. Rabe. Combining Source, Content, Presentation, Narration, and Relational Representation. In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 212–227. Springer, 2011.
- [Hil26] D. Hilbert. Über das Unendliche. *Mathematische Annalen*, 95:161–90, 1926.
- [HKR12] F. Horozal, M. Kohlhase, and F. Rabe. Extending MKM Formats at the Statement Level. In J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, and M. Wenzel, editors, *Intelligent Computer Mathematics*, pages 64–79. Springer, 2012.
- [How80] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.
- [HR11] F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. *Theoretical Computer Science*, 412(37):4919–4945, 2011.
- [IK12] M. Iancu and M. Kohlhase. Searching the Space of Mathematical Knowledge. In M. Kohlhase and P. Sojka, editors, *Mathematics Information Retrieval Workshop*, 2012.
- [IKRU13] M. Iancu, M. Kohlhase, F. Rabe, and J. Urban. The Mizar Mathematical Library in OMDoc: Translation and Applications. *Journal of Automated Reasoning*, 50(2):191–202, 2013.

- [IR11] M. Iancu and F. Rabe. Formalizing Foundations of Mathematics. *Mathematical Structures in Computer Science*, 21(4):883–911, 2011.
- [IR12] M. Iancu and F. Rabe. Management of Change in Declarative Languages. In J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, and M. Wenzel, editors, *Intelligent Computer Mathematics*, pages 325–340. Springer, 2012.
- [KAE<sup>+</sup>10] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010.
- [KMR08] M. Kohlhase, C. Müller, and F. Rabe. Notations for Living Mathematical Documents. In S. Autexier, J. Campbell, J. Rubio, V. Sorge, M. Suzuki, and F. Wiedijk, editors, *Mathematical Knowledge Management*, pages 504–519. Springer, 2008.
- [KMR09] M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN Project, 2009. see <https://trac.omdoc.org/LATIN/>.
- [Koh06] M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*. Number 4180 in Lecture Notes in Artificial Intelligence. Springer, 2006.
- [KR12] M. Kohlhase and F. Rabe. Semantics of OpenMath and MathML3. *Mathematics in Computer Science*, 6(3):235–260, 2012.
- [KRZ10] M. Kohlhase, F. Rabe, and V. Zhudev. Towards MKM in the Large: Modular Representation and Scalable Software Architecture. In S. Autexier, J. Calmet, D. Delahaye, P. Ion, L. Rideau, R. Rioboo, and A. Sexton, editors, *Intelligent Computer Mathematics*, pages 370–384. Springer, 2010.
- [KS06] M. Kohlhase and I. Şucan. A Search Engine for Mathematical Formulae. In T. Ida, J. Calmet, and D. Wang, editors, *Artificial Intelligence and Symbolic Computation*, pages 241–253. Springer, 2006.
- [Law63] F. Lawvere. *Functional Semantics of Algebraic Theories*. PhD thesis, Columbia University, 1963.
- [Law69] W. Lawvere. Adjointness in Foundations. *Dialectica*, 23(3–4):281–296, 1969.
- [Mac98] S. Mac Lane. *Categories for the working mathematician*. Springer, 1998.
- [Mes89] J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Proceedings, Logic Colloquium, 1987*, pages 275–329. North-Holland, 1989.
- [ML74] P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the '73 Logic Colloquium*, pages 73–118. North-Holland, 1974.

- [ML96] P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):3–10, 1996.
- [MML07] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522, 2007.
- [NPW02] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.
- [NSM01] P. Naumov, M. Stehr, and J. Meseguer. The HOL/NuPRL proof translator - a practical approach to formal interoperability. In *14th International Conference on Theorem Proving in Higher Order Logics*. Springer, 2001.
- [Odl95] A. Odlyzko. Tragic loss or good riddance? The impending demise of traditional scholarly journals. *International Journal of Human-Computer Studies*, 42:71–122, 1995.
- [OSV07] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. artima, 2007.
- [Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
- [PE88] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Programming Language Design and Implementation*, pages 199–208, 1988.
- [Pfe01] F. Pfenning. Logical frameworks. In *Handbook of automated reasoning*, pages 1063–1147. Elsevier, 2001.
- [PS99] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.
- [Rab06] F. Rabe. First-Order Logic with Dependent Types. In N. Shankar and U. Furbach, editors, *Automated Reasoning*, pages 377–391. Springer, 2006.
- [Rab10] F. Rabe. Representing Isabelle in LF. In K. Crary and M. Miculan, editors, *Logical Frameworks and Meta-languages: Theory and Practice*, pages 85–100. Electronic Proceedings in Theoretical Computer Science, 2010.
- [Rab12] F. Rabe. A Query Language for Formal Mathematical Libraries. In J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, and M. Wenzel, editors, *Intelligent Computer Mathematics*, pages 142–157. Springer, 2012.
- [Rab13a] F. Rabe. A Logical Framework Combining Model and Proof Theory. *Mathematical Structures in Computer Science*, 2013. to appear; see [http://kwarc.info/frabe/Research/rabe\\_combining\\_10.pdf](http://kwarc.info/frabe/Research/rabe_combining_10.pdf).

- [Rab13b] F. Rabe. The MMT API: A Generic MKM System. In D. Aspinall, J. Carette, C. Lange, and W. Windsteiger, editors, *Intelligent Computer Mathematics*. Springer, 2013. to appear.
- [RK13] F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, pages 1–95, 2013. to appear; see [http://kwarc.info/frabe/Research/RK\\_mmt\\_10.pdf](http://kwarc.info/frabe/Research/RK_mmt_10.pdf).
- [RKS11] F. Rabe, M. Kohlhase, and C. Sacerdoti Coen. A Foundational View on Integration Problems. In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 107–122. Springer, 2011.
- [Rob50] A. Robinson. On the application of symbolic logic to algebra. In *Proceedings of the International Congress of Mathematicians*, pages 686–694. American Mathematical Society, 1950.
- [RPSS07] F. Rabe, P. Pudlák, G. Sutcliffe, and W. Shen. Solving the \$100 Modal Logic Challenge. *Journal of Applied Logic*, 7(1):113–130, 2007.
- [RS09a] F. Rabe and C. Schürmann. A Module System for Twelf, 2009. see <https://cvs.concert.cs.cmu.edu/twelf/branches/twelf-mod>.
- [RS09b] F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, pages 40–48. ACM Press, 2009.
- [Soj10] K. Sojakova. Mechanically Verifying Logic Translations. Master’s thesis, Jacobs University Bremen, 2010.
- [SR09] K. Sojakova and F. Rabe. Translating Dependently-Typed Logic to First-Order Logic. In A. Corradini and U. Montanari, editors, *Recent Trends in Algebraic Development Techniques*, pages 326–341. Springer, 2009.
- [SS98] G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
- [ST88] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Control*, 76:165–210, 1988.
- [SW83] D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. In M. Karpinski, editor, *Fundamentals of Computation Theory*, pages 413–427. Springer, 1983.
- [Tar33] A. Tarski. Pojęcie prawdy w językach nauk dedukcyjnych. *Prace Towarzystwa Naukowego Warszawskiego Wydział III Nauk Matematyczno-Fizycznych*, 34, 1933. English title: The concept of truth in the languages of the deductive sciences.

- [TB85] A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985.
- [TV56] A. Tarski and R. Vaught. Arithmetical extensions of relational systems. *Compositio Mathematica*, 13:81–102, 1956.
- [W3C09] W3C. OWL 2 Web Ontology Language, 2009. <http://www.w3.org/TR/owl-overview/>.
- [WR13] A. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, 1913.
- [Zer08] E. Zermelo. Untersuchungen über die Grundlagen der Mengenlehre I. *Mathematische Annalen*, 65:261–281, 1908. English title: Investigations in the foundations of set theory I.
- [ZK09] V. Zholudev and M. Kohlhase. TNTBase: a Versioned Storage for XML. In *Proceedings of Balisage: The Markup Conference 2009*, volume 3 of *Balisage Series on Markup Technologies*. Mulberry Technologies, Inc., 2009.
- [ZKR10] V. Zholudev, M. Kohlhase, and F. Rabe. A [insert XML Format] Database for [insert cool application]. In *XMLPrague 2010*. XMLPrague.cz, 2010.



# Appendix A

## Academic Works

# A Logical Framework Combining Model and Proof Theory

FLORIAN RABE

*Computer Science, Jacobs University Bremen, Campus Ring 1, 28759 Bremen, Germany,  
f.rabe@jacobs-university.de*

*Received 12 October 2010; Revised 14 May 2011*

Mathematical logic and computer science drive the design of a growing number of logics and related formalisms such as set theories and type theories. In response to this population explosion, logical frameworks have been developed as formal meta-languages in which to represent, structure, relate, and reason about logics.

Research on logical frameworks has diverged into separate communities with often conflicting backgrounds and philosophies. In particular, among the most important logical frameworks are the framework of institutions from the area of model theory based on category theory, and the Edinburgh Logical Framework LF from the area of proof theory based on dependent type theory. Even though their ultimate motivations overlap – for example in applications to software verification – these take fundamentally different perspectives on logic.

We design a logical framework that integrates the frameworks of institutions and LF in a way that combines their complementary advantages while retaining the elegance of either one. In particular, our framework takes a balanced approach between model theory and proof theory and permits the representation of logics in a way that comprises all major ingredients of a logic: syntax, models, satisfaction, judgments, and proofs. This provides a theoretical basis for the systematic study of logics in a comprehensive logical framework. Our framework has been applied to obtain a large library of structured and machine-verified encodings of logics and logic translations.

## 1. Introduction

Since the foundational crisis of mathematics in the early 20<sup>th</sup> century, logic has been an important research topic in mathematics and later in computer science. A central issue has always been what a logic actually is (let alone a logic translation). Over the course of the last hundred years researchers have provided very different answers to this question; for example, even in 2005 the World Congress on Universal Logic held a contest about what a logic is [Béziau \(2005\)](#). Research areas that were initially connected have diverged and evolved into separate fields, and while this specialization has led to very successful results, it has also created divisions in the research on logic that are sometimes detrimental.

In response to these divisions, logical frameworks have been introduced. They unify

different logical formalisms by representing them in a fixed meta-language. Logical frameworks have been used successfully for logic-independent investigations both on the theoretical (e.g., in the textbook [Diaconescu \(2008\)](#)) and on the practical level (e.g., in the proof assistant [Paulson \(1994\)](#)).

However, today we observe that one division remains and that there are two groups of logical frameworks: *model theoretical* and *proof theoretical* ones. While some of these frameworks integrate the other side, such as the general proof theory developed in [Meseguer \(1989\)](#), almost all of them lean towards either model or proof theory. Often these sides are divided not only by research questions but also by “conflicting cultures and attitudes”, and “attempts to bridge these two cultures are rare and rather timid” (quoting an anonymous reviewer of a related paper). This paper makes one such attempt, trying to provide a balanced framework that integrates and subsumes both views on logic in a way that preserves and exploits their respective advantages.

**Model theoretical frameworks** are based on set theoretical (such as Zermelo-Fraenkel set theoretical [Fraenkel \(1922\)](#); [Zermelo \(1908\)](#)) or category theoretical ([Mac Lane \(1998\)](#)) foundations of mathematics and characterize logics model theoretically in a way that goes back to Tarski’s view of logical consequence ([Robinson \(1950\)](#); [Tarski \(1933\)](#); [Tarski and Vaught \(1956\)](#)). The central concept is that of a **model**, which interprets the non-logical symbols, and the **satisfaction** relation between formulas and models. Often category theory and initial models ([Goguen et al. \(1978\)](#)) are employed to study the model classes. The most important such framework is that of institutions ([Goguen and Burstall \(1992\)](#)).

**Proof theoretical frameworks** are based on type theoretical foundations of mathematics (such as the principia [Whitehead and Russell \(1913\)](#) or simple type theory [Church \(1940\)](#)) and characterize logics proof theoretically (e.g., [Gentzen \(1934\)](#); [Gödel \(1930\)](#)) in a way most prominently expressed in Hilbert’s program ([Hilbert \(1926\)](#)). The central concept is that of a **proof**, which derives valid **judgments** from axioms via inference rules. Often the Curry-Howard correspondence ([Curry and Feys \(1958\)](#); [Howard \(1980\)](#)) is employed to study proofs as expressions of a formal language. The most important such frameworks are Automath ([de Bruijn \(1970\)](#)), Isabelle ([Paulson \(1994\)](#)), and the Edinburgh Logical Framework (LF [Harper et al. \(1993\)](#)).

The ontological intersection of model and proof theory is the syntax of logical formulas and the **consequence** relation between formulas, which includes the study of **theorems**. Thus, both provide ontological frameworks for formulas and consequence, and this intersection will be our central interest in this work. But both frameworks also go beyond this intersection. Model theory studies the properties of models in general such as for example categoricity and initiality. Similarly, proof theory studies the properties of proofs in general such as for example normalization and program extraction. Our work may provide a starting point to investigate whether these advanced properties have interesting analogues in the respective other field.

Our unified framework picks one of the most successful frameworks from each side and combines them: institutions and LF.

**Institutions** provide an abstract definition of the syntax and the model theoretical

semantics of a logic. Research on institutions focuses on signatures, sentences, models, and satisfaction but deemphasizes proofs and derivability. Among the most characteristic features of the framework of institutions are: (i) It abstracts from the syntax of formulas and only assumes an abstract set of sentences; similarly, it uses an abstract class of models and an abstract satisfaction relation. (ii) Using a Galois connection between sentences and models, it permits viewing the relation between syntax and (model theoretical) semantics as a pair of adjoint functors ([Lawvere \(1969\)](#)). (iii) Using category theory ([Mac Lane \(1998\)](#)), it provides an abstract notion of translations, both of translations within a logic – called signature or theory morphisms – and of translations between logics ([Goguen and Rosu \(2002\)](#); [Mossakowski et al. \(2007\)](#)) – called institution (co)morphisms.

**LF** is a dependent type theory related to Martin-Löf’s type theory ([Martin-Löf \(1974\)](#)) featuring kinded type families and dependent function types. Research on logic encodings in LF focuses on syntax, proofs, and provability and dually deemphasizes models and satisfaction. Among the most characteristic features of this framework are: (i) It represents the logical and non-logical symbols as constants of the type theory using higher-order abstract syntax to represent binders. (ii) Using the judgments-as-types paradigm ([Martin-Löf \(1996\)](#)), it permits viewing syntax and (proof theoretical) semantics via judgments and inference systems and thus without appealing to a foundation of mathematics. (iii) Using a constructive point of view, translation functions are given as provably terminating programs (e.g., in [Naumov et al. \(2001\)](#)).

These two frameworks have been strongly influenced by the respective approaches, and this has created complementary strengths and weaknesses. For example, the bias towards an abstraction from the syntax exhibited by institutions is very useful for a logic-independent meta-logical analysis ([Diaconescu \(2008\)](#)). But LF’s bias towards concrete syntax excels when defining individual logics and extracting programs from individual translations ([Pfenning \(2001\)](#)). Therefore, we focus on combining the respective strengths and motivations: While our work is technically formulated within institution theory, it inherits a strictly formalist justification from LF.

In the **literature** on individual logics, the combination of proof and model theoretical perspective is commonplace, and logics are usually introduced giving their syntax, model theory, and proof theory. The major difference is often the order of the latter two. Sometimes the model theory is given primacy as *the* semantics, and then a proof theory is given and proved sound and complete (e.g., in [Barwise \(1977\)](#); [Smullyan \(1995\)](#)). And sometimes it is the other way round, e.g., in [Andrews \(1986\)](#); [Lambek and Scott \(1986\)](#). Often this difference reflects a philosophical inclination of the author.

A similar difference in primacy is found when looking at families of logics. For example, the model theoretical view takes precedence in description logic (see, e.g., [Baader et al. \(2003\)](#); [Brachman and Schmolze \(1985\)](#)) where the model theory is fixed and proof theory is studied chiefly to provide reasoning tools. This view is arguably paramount as it dominates accounts of (classical) first-order logic and most uses of logic in mathematics. On the other hand, the proof theoretical view takes precedence in, for example, higher-order logic (proof theory [Church \(1940\)](#), model theory [Henkin \(1950\)](#)); and some logics such as intuitionistic logics ([Brouwer \(1907\)](#)) are explicitly defined by their proof theory.

At the level of logical frameworks, proof theoretical principles have been integrated into model theoretical frameworks in several ways. Parchments are used in [Goguen and Burstall \(1986\)](#); [Mossakowski et al. \(1997\)](#) to express the syntax of a logic as a formal language. And for example, in [Fiadeiro and Sernadas \(1988\)](#); [Meseguer \(1989\)](#); [Mossakowski et al. \(2005\)](#), proof theory is formalized in the abstract style of category theory. These approaches are very elegant but often fail to exploit a major advantage of proof theory – the constructive reasoning over concrete syntax. Vice versa, model theoretical principles have been integrated into proof theoretical frameworks, albeit to a lesser extent. For example, models and model classes can be represented using axiomatic type classes in [Isabelle Haftmann and Wenzel \(2006\)](#); [Paulson \(1994\)](#) or using structures in Mizar ([Trybulec and Blair \(1985\)](#)).

But a logical framework that systematically subsumes frameworks from both perspectives has so far been lacking, and our contribution consists in providing such a framework. Our work is separated into three main parts.

Firstly – in Sect. 3 – we extend institutions with an abstract notion of proof theory arriving at what we call *logics*. Our logics are defined in the spirit of institutions, in particular retaining the abstraction from concrete syntax. Thus, they are very similar to the general logics of [Meseguer \(1989\)](#) and to the proof theoretic institutions of [Diaconescu \(2006\)](#); [Mossakowski et al. \(2005\)](#). We also discuss the use of meta-logics, which can be seen as a response to [Tarlecki \(1996\)](#), where the use of a meta-institution is explored in general and the use of LF suggested in particular.

Secondly – in Sect. 4 – we give a logic  $\mathbb{M}$  based on LF that we will use as our meta-logic. The central idea is that a signature of  $\mathbb{M}$  captures syntax, proof theory, and model theory of an object logic. Similarly,  $\mathbb{M}$ -signature morphisms capture translations between object logics, including the translations of syntax, proof theory, and model theory.  $\mathbb{M}$  is defined in the spirit of LF, in particular retaining the focus on concrete syntax. Consequently, judgments are represented as types and proofs as terms. Moreover – inspired by the ideas of Lawvere ([Lawvere \(1963\)](#)) – models are represented as signature morphisms from the logical theory into a signature representing the foundation of mathematics.

In this paper, we will use the term “foundation of mathematics” to refer to a fixed language in which mathematical objects are expressed. While mathematicians usually do not explicate a foundation (often implicitly assuming a variant of first-order set theory as the foundation), the choice of foundation is more difficult in computational logic, where for example higher-order logic ([Church \(1940\)](#)) and the calculus of constructions ([Coquand and Huet \(1988\)](#)) are often used instead. By representing such a foundation as an LF signature itself, our framework can formulate model theory without committing to a specific foundation.

Thirdly – in Sect. 5 – we demonstrate how to use  $\mathbb{M}$  as a meta-logic. We also show how existing logics and translations can be encoded in  $\mathbb{M}$  and how we can reason about the adequacy of such encodings. This section reconciles the type/proof and the set/model theoretical perspective: institution-based logics are expressed using the syntax of an LF-based meta-logic. We will exemplify all our definitions by giving a simple logic translation from modal logic to first-order logic as a running example. Moreover, we have

implemented our framework and evaluated it in large scale case studies (Codescu et al. (2011); Horozal and Rabe (2011); Iancu and Rabe (2011)).

We discuss and evaluate our framework in Sect. 6 and conclude in Sect. 7. We give introductions to the frameworks of institutions and LF in Sect. 2.

## 2. Logical Frameworks

### 2.1. The Model Theoretical Framework of Institutions

Institutions were introduced in Goguen and Burstall (1992) as a means to manage the population explosion among logics in use. The key idea is to abstract from the satisfaction relation between the sentences and the models. We will only give an intuitive introduction to the notion of an institution here. A rigorous and more comprehensive treatment is inherent in our definitions in Sect. 3.

The components of an institution are centered around a class **Sig** of **signatures**. To each signature  $\Sigma \in \mathbf{Sig}$  are associated a set **Sen**( $\Sigma$ ) of **sentences**, a class **Mod**( $\Sigma$ ) of **models**, and the **satisfaction** relation  $\models_{\Sigma} \subseteq \mathbf{Mod}(\Sigma) \times \mathbf{Sen}(\Sigma)$ .

**Example 1 (Modal Logic).** For a simple institution  $\text{ML}$  of modal logic, the signatures in  $\mathbf{Sig}^{\text{ML}}$  are the finite sets of propositional variables (chosen from some fixed set of symbols). The set  $\mathbf{Sen}^{\text{ML}}(\Sigma)$  is the smallest set containing  $p$  for all  $p \in \Sigma$  and closed under sentence formation  $F \supset G$  and  $\Box F$ .

A model  $M \in \mathbf{Mod}^{\text{ML}}(\Sigma)$  is a Kripke model, i.e., a tuple  $(W, \prec, V)$  for a set  $W$  of worlds, an accessibility relation  $\prec \subseteq W \times W$ , and a valuation  $V : \Sigma \times W \rightarrow \{0, 1\}$  for the propositional variables.  $V$  is extended to a mapping  $\mathbf{Sen}(\Sigma) \times W \rightarrow \{0, 1\}$  in the usual way, and the satisfaction relation  $M \models_{\Sigma}^{\text{ML}} F$  holds iff  $V(F, w) = 1$  for all worlds  $w \in W$ .

This abstract definition yields a rich setting in which institutions can be studied (see, e.g., Diaconescu (2008)). Most importantly, we can define an abstract notion of **theory** as a pair  $(\Sigma, \Theta)$  for a set  $\Theta \subseteq \mathbf{Sen}(\Sigma)$  of axioms. (See Term. 1 for different uses of the term “theory”.) A sentence  $F \in \mathbf{Sen}(\Sigma)$  is a  $(\Sigma, \Theta)$ -**theorem** iff for all models  $M \in \mathbf{Mod}(\Sigma)$  we have that  $M \models_{\Sigma} A$  for all  $A \in \Theta$  implies  $M \models_{\Sigma} F$ . This is the model theoretical consequence relation and written as  $\Theta \models_{\Sigma} F$ .

The above concepts have a second dimension when all classes are extended to categories (Mac Lane (1998)). **Sig** is in fact a category, and **Sen** a functor  $\mathbf{Sig} \rightarrow \mathcal{SET}$ . Similarly, **Mod** is a functor  $\mathbf{Sig} \rightarrow \mathcal{CAT}^{op}$  (see also Not. 1). Signature morphisms represent translations between signatures, and the actions of **Sen** and **Mod** on signature morphism extend these translations to sentences and models. Sentence and model translations go in opposite directions, which corresponds to an adjunction between syntax and semantics. Finally  $\models_{\Sigma}$  is subject to the satisfaction condition, which guarantees that the satisfaction of a sentence in a model is invariant under signature morphisms.

**Example 2 (Continued).** In the case of modal logic, signature morphisms  $\sigma : \Sigma \rightarrow \Sigma'$  are substitutions of  $\Sigma'$ -sentences for the propositional variables in  $\Sigma$ , and  $\mathbf{Sen}^{\text{ML}}(\sigma) : \mathbf{Sen}^{\text{ML}}(\Sigma) \rightarrow \mathbf{Sen}^{\text{ML}}(\Sigma')$  is the induced homomorphic extension. The functor  $\mathbf{Mod}^{\text{ML}}(\sigma)$

reduces a model  $(W, \prec, V') \in \mathbf{Mod}^{\text{ML}}(\Sigma')$  to the model  $(W, \prec, V) \in \mathbf{Mod}^{\text{ML}}(\Sigma)$  by putting  $V(p, w) = V'(\sigma(p), w)$ .

A **theory morphism**  $\sigma : (\Sigma, \Theta) \rightarrow (\Sigma', \Theta')$  is a signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$  such that for all axioms  $A \in \Theta$ , we have that  $\mathbf{Sen}(\sigma)(A)$  is a  $(\Sigma', \Theta')$ -theorem. This implies that  $\mathbf{Sen}(\sigma)$  maps  $(\Sigma, \Theta)$ -theorems to  $(\Sigma', \Theta')$ -theorems, i.e., theory morphisms preserve truth. Adjointly, one can show that  $\sigma$  is a theory morphism iff  $\mathbf{Mod}(\sigma)$  maps  $(\Sigma', \Theta')$ -models to  $(\Sigma, \Theta)$ -models.

Institutions can be described elegantly as functors out of a signature category into a fixed category  $C$  of *rooms*. This yields a notion of institution translations via the well-known notion of the lax slice category of functors into  $C$ . Such translations are called institution **comorphisms** (Goguen and Rosu (2002); Meseguer (1989); Tarlecki (1996)). This is the basis of a number of logic translations that have been expressed as translations between institutions (see, e.g., Mossakowski et al. (2007) for examples).

A comorphism from  $\mathbb{I}$  to  $\mathbb{I}'$  translates signatures and sentences from  $\mathbb{I}$  to  $\mathbb{I}'$  and models in the opposite direction. The sentence translation preserves the (model theoretical) consequence relation  $\Theta \models_{\Sigma} F$ . If the model translations are surjective – called the model expansion property – the consequence relation is also reflected. This is the basis of the important borrowing application (Cerioli and Meseguer (1997)), where the consequence relation of one institution is reduced to that of another, e.g., one for which an implementation is available.

In fact, very few logic translations can be expressed as comorphisms in this sense – instead, it is often necessary to translate  $\mathbb{I}$ -signatures to  $\mathbb{I}'$ -theories. However, such translations can be represented as institution comorphisms from  $\mathbb{I}$  to  $\text{Th}(\mathbb{I}')$ . Here  $\text{Th}(\mathbb{I}')$  is a new institution whose signatures are the theories of  $\mathbb{I}'$ , a straightforward construction that is possible for all institutions.

## 2.2. The Proof Theoretical Framework LF

LF (Harper et al. (1993)) is a dependent type theory related to Martin-Löf type theory (Martin-Löf (1974)). It is the corner of the  $\lambda$ -cube (Barendregt (1992)) that extends simple type theory with dependent function types. We will work with the Twelf implementation of LF (Pfenning and Schürmann (1999)).

The main use of LF and Twelf is as a **logical framework** in which deductive systems are represented. Typically, **kinded type families** are declared to represent the syntactic classes of the system. For example, to represent higher-order logic, we use a signature *HOL* with declarations

```
tp      : type
tm      : tp → type
bool   : tp
ded    : tm bool → type
```

Here **type** is the LF-kind of types, and **tp** is an LF-type whose LF-terms represent the STT-types. **tp → type** is the kind of type families that are indexed by terms of

LF-type  $tp$ ; then  $tm A$  is the LF-type whose terms represent the STT-terms of type  $A$ . For example,  $bool$  represents the HOL-type of propositions such that HOL-propositions are represented as LF-terms of type  $tm bool$ . LF employs the Curry-Howard correspondence to represent proofs-as-term (Curry and Feys (1958); Howard (1980)) and extends it to the **judgments-as-types** methodology (Martin-Löf (1996)):  $ded$  declares the truth judgment on HOL-propositions, and the type  $ded F$  represents the judgment that  $F$  is derivable. HOL-derivations of  $F$  are represented as LF-terms of type  $ded F$ , and  $F$  is provable in HOL iff there is an LF term of type  $ded F$ .

Additionally, **typed constants** are declared to construct expressions of these syntactic classes, i.e., types (e.g.,  $bool$  above), terms, propositions, and derivations of HOL. For example, consider:

$$\begin{aligned} \Rightarrow & : tm\;bool \rightarrow tm\;bool \rightarrow tm\;bool \\ \Rightarrow_E & : \{F : bool\} \{G : bool\} ded(F \Rightarrow G) \rightarrow ded\;F \rightarrow ded\;G \\ \forall & : \{A : tp\} (tm\;A \rightarrow bool) \rightarrow bool \end{aligned}$$

Here  $\Rightarrow$  encodes implication as a binary proposition constructor.  $\Rightarrow_E$  encodes the implication elimination rule (modus ponens): It takes first two propositions  $F$  and  $G$  as arguments, and then two derivations of  $F \Rightarrow G$  and  $F$ , respectively, and returns a derivation of  $G$ . Note that the types of the later arguments depend on the values  $F$  and  $G$  of the first two arguments. The encoding of the universal quantifier  $\forall$  uses **higher-order abstract syntax** to declare binders as constants: LF terms of type  $S \rightarrow T$  are in bijection to LF terms of type  $T$  with a free variable of type  $S$ . Thus,  $\forall$  encodes a proposition constructor that takes a HOL type  $A$  and a proposition with a free variable of type  $tm\;A$ .

We will always use Twelf notation for the LF primitives of binding and application: The type  $\Pi_{x:A}B(x)$  of dependent functions taking  $x : A$  to an element of  $B(x)$  is written  $\{x : A\}B\;x$ , and the function term  $\lambda_{x:A}t(x)$  taking  $x : A$  to  $t(x)$  is written  $[x : A]t\;x$ . As usual, we write  $A \rightarrow B$  instead of  $\{x : A\}B$  if  $x$  does not occur in  $B$ .

This yields the following grammar for the three levels of LF expressions:

$$\begin{array}{lll} \text{Kinds:} & K & ::= \text{type} \mid \{x : A\}K \\ \text{Type families:} & A, B & ::= a \mid A\;t \mid [x : A]B \mid \{x : A\}B \\ \text{Terms:} & s, t & ::= c \mid x \mid [x : A]t \mid s\;t \end{array}$$

$\{x : A\}K$  and  $\{x : A\}B$  are used to abstract over variables occurring in kinds and types, respectively. Similarly, there are two  $\lambda$ -abstractions and two applications. The three productions for kind level abstraction are given in gray.

The **judgments** are  $\Gamma \vdash_\Sigma K \text{ kind}$  for well-formed kinds,  $\Gamma \vdash_\Sigma A : K$  for well-kinded type families, and  $\Gamma \vdash_\Sigma s : A$  for well-typed terms. They are defined relative to a signature  $\Sigma$  and a context  $\Gamma$ , which declare the symbols and variables, respectively, that can occur in expressions. Moreover, all levels come with an **equality** judgment, and both abstractions respect  $\alpha$ ,  $\beta$ , and  $\eta$ -conversion. All judgments are decidable Harper et al. (1993).

The grammar for signatures and contexts and their translations is as follows:

$$\begin{array}{ll} \text{Signatures} & \Sigma ::= \cdot | \Sigma, c : A | \Sigma, a : K, \\ \text{Morphisms} & \sigma ::= \cdot | \sigma, c := t | \Sigma, a := A \\ \text{Contexts} & \Gamma ::= \cdot | \Gamma, x : A \\ \text{Substitutions} & \sigma ::= \cdot | \gamma, x := t \end{array}$$

An LF **signature**  $\Sigma$  holds the globally available declared names; it is a list of kinded type family declarations  $a : K$  and typed constant declarations  $c : A$ . Similarly, a  $\Sigma$ -**context** holds the locally available declared names; it is a list typed variable declarations  $x : A$ .

Both signatures and contexts come with a notion of homomorphic translations. Given two signatures  $\Sigma$  and  $\Sigma'$ , a **signature morphism**  $\sigma : \Sigma \rightarrow \Sigma'$  is a typing- and kinding-preserving map of  $\Sigma$ -symbols to closed  $\Sigma'$ -expressions. Thus,  $\sigma$  maps every constant  $c : A$  of  $\Sigma$  to a term  $\sigma(c) : \bar{\sigma}(A)$  and every type family symbol  $a : K$  to a type family  $\sigma(a) : \bar{\sigma}(K)$ . Here,  $\bar{\sigma}$  is the homomorphic extension of  $\sigma$  to all closed  $\Sigma$ -expressions, and we will write  $\sigma$  instead of  $\bar{\sigma}$  from now on.

Signature morphisms preserve typing, kinding, and equality of expressions, i.e., if  $\cdot \vdash_{\Sigma} E : F$ , then  $\cdot \vdash_{\Sigma'} \sigma(E) : \sigma(F)$ , and similarly for equality. In particular, because  $\sigma$  must map all axioms or inference rules declared in  $\Sigma$  to proofs or derived inference rules, respectively, over  $\Sigma'$ , signature morphisms preserve the provability of judgments. Two signature morphisms are equal if they map the same constants to  $\alpha\beta\eta$ -equal expressions. Up to this equality, signatures and signature morphisms form a category, which we denote by  $\mathbb{LF}$ .

$\mathbb{LF}$  has inclusion morphisms  $\Sigma \hookrightarrow \Sigma, \Sigma'$  and pushouts along inclusions (Harper et al. (1994)). Moreover, a coherent system of pushouts along inclusions can be chosen canonically: Given  $\sigma : \Sigma \rightarrow \Sigma'$  and an inclusion  $\Sigma \hookrightarrow \Sigma, c : A$ , the canonical **pushout** is given by

$$\sigma, c := c : \Sigma, c : A \rightarrow \Sigma', c : \sigma(A)$$

(except for possibly renaming  $c$  if it is not fresh for  $\Sigma'$ ). The canonical choices for pushouts along other inclusions are obtained accordingly.

Similarly, given two  $\Sigma$ -contexts  $\Gamma$  and  $\Gamma'$ , a **substitution**  $\gamma : \Gamma \rightarrow \Gamma'$  is a typing-preserving map of  $\Gamma$ -variables to  $\Gamma'$ -expressions (which leaves the symbols of  $\Sigma$  fixed). Thus,  $\sigma$  maps every variable  $x : A$  of  $\Gamma$  to a term  $\gamma(x) : \bar{\gamma}(A)$ . Again  $\bar{\gamma}$  is the homomorphic extension of  $\gamma$ , and we write  $\gamma$  instead of  $\bar{\gamma}$ . Similar to signature morphisms, substitutions preserve typing, kinding, and equality of expressions in contexts, e.g., if  $\Gamma \vdash_{\Sigma} E : F$ , then  $\Gamma' \vdash_{\Sigma} \gamma(E) : \gamma(F)$ . Two substitutions are equal if they map the same variables to  $\alpha\beta\eta$ -equal terms. Up to this equality, for a fixed signature  $\Sigma$ , the contexts over  $\Sigma$  and the substitutions between them form a category.

Finally a signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$  can be applied component-wise to contexts and substitutions:  $\sigma(\cdot) = \cdot$  and  $\sigma(\Gamma, x : A) = \sigma(\Gamma), x : \sigma(A)$  for contexts, and  $\sigma(\cdot) = \cdot$ , and  $\sigma(\gamma, x := t) = \sigma(\gamma), x := \sigma(t)$  for substitutions. We have the invariant that if  $\gamma : \Gamma \rightarrow \Gamma'$  over  $\Sigma$  and  $\sigma : \Sigma \rightarrow \Sigma'$ , then  $\sigma(\gamma) : \sigma(\Gamma) \rightarrow \sigma(\Gamma')$ . Categorically,  $\sigma(-)$  is a functor from the category of  $\Sigma$ -contexts to the category of  $\Sigma'$ -contexts.

### 3. A Comprehensive Logical Framework

Our primary objective is to encode logics and logic translations in an LF-based metalogic. Therefore, we must do three things (i) define what logics and logic translations are, which we do in Sect. 3.1 and 3.2, respectively, (ii) define what it means to encode them in a meta-logic, which we do in Sect. 3.3, and (iii) give the LF-based meta-logic, which we do in Sect. 4. Note that (ii) and (iii) could also be given the other way round. We give (ii) first for an arbitrary meta-logic so that it is possible to substitute other meta-logics later.

#### 3.1. Logics

**Model Theories and Institutions** In order to give a unified approach to model and proof theory, we will refactor the definition of institutions by splitting an institution into syntax and model theory:

**Definition 1 (Logic Syntax).** A **logic syntax** is a pair  $(\mathbf{Sig}, \mathbf{Sen})$  such that  $\mathbf{Sig}$  is a category and  $\mathbf{Sen} : \mathbf{Sig} \rightarrow \mathcal{SET}$  is a functor.

**Definition 2 (Model Theory).** For a logic syntax  $(\mathbf{Sig}, \mathbf{Sen})$ , a **model theory** is a tuple  $(\mathbf{Mod}, \models)$  such that  $\mathbf{Mod} : \mathbf{Sig} \rightarrow \mathcal{CAT}^{op}$  is a functor and  $\models_\Sigma \subseteq \mathbf{Sen}(\Sigma) \times |\mathbf{Mod}(\Sigma)|$  is family of relations such that the satisfaction condition holds: For all  $\sigma : \Sigma \rightarrow \Sigma'$ ,  $F \in \mathbf{Sen}(\Sigma)$ , and  $M' \in |\mathbf{Mod}(\Sigma')|$ , we have  $\mathbf{Mod}(\sigma)(M') \models_\Sigma F$  iff  $M' \models_{\Sigma'} \mathbf{Sen}(\sigma)(F)$ .

In particular, institutions are exactly the pairs of a logic syntax and a model theory for it.

**Notation 1.** In the literature, the model theory functor is usually given as  $\mathbf{Mod} : \mathbf{Sig}^{op} \rightarrow \mathcal{CAT}$ . We choose the dual orientation here in order to emphasize the symmetry between model and proof theory later on.

**Example 3 (Modal Logic).**  $(\mathbf{Sig}^{\text{ML}}, \mathbf{Sen}^{\text{ML}})$  and  $(\mathbf{Mod}^{\text{ML}}, \models^{\text{ML}})$  from Ex. 1 are examples of a logic syntax and a model theory for it. To complete the example, we have to show the satisfaction condition: Assume  $\sigma : \Sigma \rightarrow \Sigma'$ ,  $F \in \mathbf{Sen}^{\text{ML}}(\Sigma)$ , and  $(W, \prec, V) \in \mathbf{Mod}^{\text{ML}}(\Sigma')$ ; then the satisfaction condition follows if we show that  $V(F, w) = V'(\mathbf{Sen}^{\text{ML}}(\sigma)(F), w)$ , which is easy to see.

Note that we proved that  $\sigma$  preserves truth in all worlds, whereas the satisfaction condition only requires the weaker condition that  $\mathbf{Sen}^{\text{ML}}(\sigma)(F)$  holds in all worlds of  $(W, \prec, V)$  iff  $F$  holds in all worlds of  $(W, \prec, V')$ . This extension of the satisfaction condition to all components of syntax and model theory is typical but different for each institution. A uniform formulation was given in Aiguier and Diaconescu (2007) using stratified institutions.

**Proof Theories and Logics** Like in the case of model theories, for a given logic syntax, we will define proof theories as pairs  $(\mathbf{Pf}, \vdash)$ . First, we define proof categories, the values of the  $\mathbf{Pf}$  functor:

**Definition 3 (Proof Categories).** A **proof category** is a category  $P$  with finite products (including the empty product).  $\mathcal{PFCAT}$  is the category of proof categories together with the functors preserving finite products.

**Notation 2.** If a category has the product  $(F_1, \dots, F_n)$ , we write it as  $(F_i)_1^n$  or simply  $F_1^n$ . Similarly, for  $p_i : E \rightarrow F_i$ , we write  $p_1^n$  for the universal morphism  $(p_1, \dots, p_n) : E \rightarrow F_1^n$ .

We think of the objects of  $P$  as **judgments** about the syntax that can be assumed and derived. The intuition of a morphism  $p_1^n$  from  $E_1^m$  to  $F_1^n$  is that it provides **evidence**  $p_i$  for each judgment  $F_i$  under the assumptions  $E_1, \dots, E_m$ .

**Example 4 (Continued).** We obtain a proof theory functor  $\mathbf{Pf}^{\text{ML}} : \mathbf{Sig}^{\text{ML}} \rightarrow \mathcal{PFCAT}$  as follows. The proof category  $\mathbf{Pf}^{\text{ML}}(\Sigma)$  is the category of finite families  $F_1^n$  for  $F_i \in \mathbf{Sen}^{\text{ML}}(\Sigma)$ . The morphisms in  $\mathbf{Pf}^{\text{ML}}(\Sigma)$  from  $E_1^m$  to  $F_1^n$  are the families  $p_1^n$  such that every  $p_i$  is a proof of  $F_i$  from assumptions  $E_1, \dots, E_m$ . We can choose any calculus to make the notion of proofs precise, e.g., a Hilbert calculus with rules for modus ponens and necessitation.

$\mathbf{Pf}^{\text{ML}}(\Sigma)$  has finite products by taking products of families. The action of  $\mathbf{Pf}^{\text{ML}}$  on signature morphisms  $\sigma : \Sigma \rightarrow \Sigma'$  is induced in the obvious way.

**Remark 1 (Proof Categories).** By assuming proof categories to have products in Def. 7, we are implicitly assuming the rules of weakening (given by morphisms from  $(A, B)$  to  $A$ ), contraction (given by morphisms from  $A$  to  $(A, A)$ ), and exchange (given by morphisms from  $(A, B)$  to  $(B, A)$ ). A substructural framework, e.g., one akin to linear logic (Girard (1987)), could be obtained by weakening the condition on the existence of products.

We only require finite products because deductive systems typically use words over countable alphabets, in which only finite products of judgments can be expressed. This amounts to a restriction to compact deductive systems, which corresponds to the constructive flavor of proof theory.

It is natural to strengthen the definition by further assuming that all objects of a proof category  $P$  can be written as products of irreducible objects, in which case proof categories become many-sorted Lawvere categories (Lawvere (1963)). Then, up to isomorphism, the judgments can be seen as the finite multi-sets of irreducible objects, and canonical products are obtained by taking unions of multi-sets. This is the typical situation, but we do not assume it here because we will not make use of this additional assumption.

An even stronger definition could require that the irreducible objects of  $\mathbf{Pf}(\Sigma)$  are just the sentences in  $\mathbf{Sen}(\Sigma)$ . This is the case in Ex. 4 where the irreducible objects are the singleton multi-sets and thus essentially the sentences. But it is desirable to avoid this assumption in order to include deductive systems where auxiliary syntax is introduced, such as hypothetical judgments (as used in natural deduction, see Ex. 5) or signed formulas (as used in tableaux calculi).

**Example 5.** The proof theory of the first-order natural deduction calculus arises as follows. For a  $\text{FOL}$ -signature  $\Sigma$ , an atomic  $\Sigma$ -judgment is of the form  $\Gamma; \Delta \vdash F$  where  $\Gamma$  is a context declaring some free variables that may occur in  $\Delta$  and  $F$ ;  $\Delta$  is a list of  $\Sigma$ -formulas; and  $F$  is a  $\Sigma$ -formula. The objects of  $\mathbf{Pf}^{\text{FOL}}(\Sigma)$  are the multi-sets of atomic  $\Sigma$ -judgments, and products are given by unions of multi-sets. A morphism in  $\mathbf{Pf}^{\text{FOL}}(\Sigma)$  from  $J_1^m$  to a singleton multi-set ( $K$ ) is a natural deduction proof tree whose root is labelled with  $K$  and whose leaves are labelled with one element of  $J_1, \dots, J_m$ . Morphisms with other codomains are obtained from the universal property of the product.

Giving a proof category  $\mathbf{Pf}(\Sigma)$  for a signature  $\Sigma$  is not sufficient to define a proof theoretical semantics of  $\Sigma$  – we also have to relate the  $\Sigma$  sentences and the  $\Sigma$ -judgments. For that purpose, we use a map  $\vdash_\Sigma: \mathbf{Sen}(\Sigma) \rightarrow \mathbf{Pf}(\Sigma)$  assigning to every sentence its truth judgment. In particular,  $\mathbf{Pf}(\Sigma)$ -morphisms from the empty family () to  $\vdash_\Sigma F$  represent the proofs of  $F$ . Corresponding to the satisfaction condition, the truth judgment should be preserved under signature morphisms. This yields the following definition:

**Definition 4 (Proof Theory).** For a logic syntax  $(\mathbf{Sig}, \mathbf{Sen})$ , a **proof theory** is a tuple  $(\mathbf{Pf}, \vdash)$  such that  $\mathbf{Pf}: \mathbf{Sig} \rightarrow \mathcal{PFCAT}$  is a functor and  $\vdash_\Sigma$  is a mapping from  $\mathbf{Sen}(\Sigma)$  to  $\mathbf{Pf}(\Sigma)$  such that for all  $\sigma: \Sigma \rightarrow \Sigma'$  and  $F \in \mathbf{Sen}(\Sigma)$ , we have  $\mathbf{Pf}(\sigma)(\vdash_\Sigma F) = \vdash_{\Sigma'} \mathbf{Sen}(\sigma)(F)$ .

Finally, we can put together syntax, model theory, and proof theory to form a logic:

**Definition 5 (Logics).** A **logic** is a tuple  $\mathbb{I} = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models, \mathbf{Pf}, \vdash)$  such that  $(\mathbf{Sig}, \mathbf{Sen})$  is a logic syntax and  $(\mathbf{Mod}, \models)$  and  $(\mathbf{Pf}, \vdash)$  are a model theory and a proof theory, respectively, for it.

**Example 6 (Continued).** We obtain a proof theory and thus a logic (in our formal sense) for modal logic by using  $\mathbf{Pf}^{\text{ML}}$  as in Ex. 4 and putting  $\vdash_\Sigma^{\text{ML}} F = (F)$  where  $(F)$  is the family containing a single element  $F$ . Indeed, we have  $\mathbf{Pf}^{\text{ML}}(\sigma)(\vdash_\Sigma^{\text{ML}} F) = (\mathbf{Sen}^{\text{ML}}(\sigma)(F)) = \vdash_{\Sigma'}^{\text{ML}} \mathbf{Sen}^{\text{ML}}(\sigma)(F)$ .

Thus, a logic consists of a category of signatures  $\mathbf{Sig}$ , a sentence functor  $\mathbf{Sen}$ , a model theory functor  $\mathbf{Mod}$ , a proof theory functor  $\mathbf{Pf}$ , and model and proof theoretical definitions of truth  $\models$  and  $\vdash$ , respectively. This is visualized on the right for a signature morphism  $\sigma: \Sigma \rightarrow \Sigma'$ . Technically, this is not a diagram in the sense of category theory. But if we treat the sets  $\mathbf{Sen}(\Sigma)$  and  $\mathbf{Sen}(\Sigma')$  as discrete categories, then the lower half is a commuting diagram of categories. Moreover, if we forget morphisms and treat functors as special cases of relations between classes, then the upper half is a commutative diagram in the category of classes and relations. This is made more precise in the following remark.

$$\begin{array}{ccc}
 \mathbf{Mod}(\Sigma) & \xleftarrow{\mathbf{Mod}(\sigma)} & \mathbf{Mod}(\Sigma') & \mathcal{CAT} \\
 \models_\Sigma \downarrow & & \downarrow \models_{\Sigma'} & \\
 \mathbf{Sen}(\Sigma) & \xrightarrow{\mathbf{Sen}(\sigma)} & \mathbf{Sen}(\Sigma') & \mathcal{SET} \\
 \vdash_\Sigma \downarrow & & \downarrow \vdash_{\Sigma'} & \\
 \mathbf{Pf}(\Sigma) & \xrightarrow{\mathbf{Pf}(\sigma)} & \mathbf{Pf}(\Sigma') & \mathcal{PFCAT}
 \end{array}$$

**Remark 2.** The definition of logics can be phrased more categorically, which also makes the formal symmetry between model and proof theory more apparent. Let  $\mathcal{CLASS}$  and  $\mathcal{REL}$  be the categories of classes with mappings and relations, respectively, and let  $|-| : \mathcal{CAT}^{\text{op}} \rightarrow \mathcal{CLASS}$  and  $|-|^r : \mathcal{CAT}^{\text{op}} \rightarrow \mathcal{REL}$  be the functors forgetting morphisms. Let us also identify  $\mathcal{SET}$  with its two inclusions into  $\mathcal{REL}$  and  $\mathcal{PFCAT}$ . Then  $\models$  and  $\vdash$  are natural transformations  $\models : \mathbf{Sen} \rightarrow |-|^r \circ \mathbf{Mod}$  and  $\vdash : \mathbf{Sen} \rightarrow |-| \circ \mathbf{Pf}$ .

Then a logic  $(\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models, \mathbf{Pf}, \vdash)$  yields the following diagram in the category  $\mathcal{CAT}$ , where double arrows indicate natural transformations between the functors making up the surrounding rectangles:

$$\begin{array}{ccccc}
 & & \mathcal{CAT}^{\text{op}} & \xrightarrow{|-|^r} & \mathcal{REL} \\
 & \nearrow \mathbf{Mod} & \uparrow \models & \nearrow & \\
 \mathbf{Sig} & \xrightarrow{\mathbf{Sen}} & \mathcal{SET} & & \\
 & \searrow \mathbf{Pf} & \downarrow \vdash & \searrow & \\
 & & \mathcal{PFCAT} & \xrightarrow{|-|} & \mathcal{CLASS}
 \end{array}$$

**Remark 3.** Even more categorically than in Rem. 2, assume  $U : \mathcal{SET} \rightarrow \mathcal{REL} \times \mathcal{CLASS}$  maps a set  $S$  to  $(S, S)$ , and  $V : \mathcal{CAT}^{\text{op}} \times \mathcal{PFCAT} \rightarrow \mathcal{REL} \times \mathcal{CLASS}$  maps a pair of a model category and a proof category to itself but forgetting morphisms. Then logics are the functors from some category of signatures  $\mathbf{Sig}$  to the comma category  $(U \downarrow V)$ .

**Theories and Consequence** While the notion of theories only depends on a logic syntax, both the model theoretical semantics and the proof theoretical semantics induce one consequence relation each: As usual, we write  $\models$  and  $\vdash$  for the model and proof theoretical consequence, respectively. Consequently, we obtain two notions of theory morphisms.

**Definition 6 (Theories).** Given a logic syntax  $(\mathbf{Sig}, \mathbf{Sen})$ , a **theory** is a pair  $(\Sigma, \Theta)$  for  $\Sigma \in \mathbf{Sig}$  and  $\Theta \subseteq \mathbf{Sen}(\Sigma)$ . The elements of  $\Theta$  are called the **axioms** of  $(\Sigma, \Theta)$ .

**Definition 7 (Consequence).** Assume a logic  $\mathbb{I} = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models, \mathbf{Pf}, \vdash)$ . For a theory  $(\Sigma, \Theta)$  and a  $\Sigma$ -sentence  $F$ , we define **consequence** as follows:

- $\Theta$  **entails**  $F$  proof theoretically, written  $\Theta \vdash_{\Sigma}^{\mathbb{I}} F$ , iff there exist a finite subset  $\{F_1, \dots, F_n\} \subseteq \Theta$  and a  $\mathbf{Pf}(\Sigma)$ -morphism from  $(\vdash_{\Sigma}^{\mathbb{I}} F_i)_1^n$  to  $\vdash_{\Sigma}^{\mathbb{I}} F$ ,
- $\Theta$  **entails**  $F$  model theoretically, written  $\Theta \models_{\Sigma}^{\mathbb{I}} F$ , iff every model  $M \in \mathbf{Mod}(\Sigma)$  satisfying all sentences in  $\Theta$  also satisfies  $F$ .

We will drop the superscript  $\mathbb{I}$  if it is clear from the context.

A signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$  is called a model theoretical (or proof theoretical) **theory morphism** from  $(\Sigma, \Theta)$  to  $(\Sigma', \Theta')$  if for all  $F \in \Theta$ , we have  $\Theta' \models_{\Sigma'} \mathbf{Sen}(\sigma)(F)$  (or  $\Theta' \vdash_{\Sigma'} \mathbf{Sen}(\sigma)(F)$  in case of proof theoretical theory morphisms).

**Terminology 1.** In the literature, the term *theory* is sometimes used only for a set of sentences that is closed under the consequence relation. In that case theories in our sense are called *presentations*.

Theory morphisms preserve the respective consequence relation:

**Lemma 1 (Truth Preservation).** Assume a (i) model theoretical or (ii) proof theoretical theory morphism  $\sigma : (\Sigma, \Theta) \rightarrow (\Sigma', \Theta')$  and a sentence  $F \in \mathbf{Sen}(\Sigma)$ . Then in case

- (i):  $\Theta \models_{\Sigma} F$  implies  $\Theta' \models_{\Sigma'} \mathbf{Sen}(\sigma)(F)$ ,
- (ii):  $\Theta \vdash_{\Sigma} F$  implies  $\Theta' \vdash_{\Sigma'} \mathbf{Sen}(\sigma)(F)$ .

The well-known concepts of soundness and completeness relate the two consequence relations:

**Definition 8 (Soundness and Completeness).** A logic is **sound** iff  $\emptyset \vdash_{\Sigma} F$  implies  $\emptyset \models_{\Sigma} F$  for all signatures  $\Sigma$  and all sentences  $F$ . It is **strongly sound** iff  $\Theta \vdash_{\Sigma} F$  implies  $\Theta \models_{\Sigma} F$  for all theories  $(\Sigma, \Theta)$  and all sentences  $F$ . Similarly, a logic is **strongly complete** or **complete** if the respective converse implication holds.

In particular, if a logic is strongly sound and strongly complete, then proof and model theoretical consequence and theory morphisms coincide.

### 3.2. Logic Translations

**Logic Translations** We will now define translations between two logics. Since our framework is based on institutions, we will build upon the existing notion of an institution *comorphism*.

**Definition 9 (Logic Comorphism).** Assume two logics  $\mathbb{I} = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models, \mathbf{Pf}, \vdash)$  and  $\mathbb{I}' = (\mathbf{Sig}', \mathbf{Sen}', \mathbf{Mod}', \models', \mathbf{Pf}', \vdash')$ . A **logic comorphism** from  $\mathbb{I}$  to  $\mathbb{I}'$  is a tuple  $(\Phi, \alpha, \beta, \gamma)$  consisting of a functor  $\Phi : \mathbf{Sig} \rightarrow \mathbf{Sig}'$  and natural transformations  $\alpha : \mathbf{Sen} \rightarrow \mathbf{Sen}' \circ \Phi$ ,  $\beta : \mathbf{Mod} \rightarrow \mathbf{Mod}' \circ \Phi$ , and  $\gamma : \mathbf{Pf} \rightarrow \mathbf{Pf}' \circ \Phi$  such that

1. for all  $\Sigma \in \mathbf{Sig}$ ,  $F \in \mathbf{Sen}(\Sigma)$ ,  $M' \in \mathbf{Mod}'(\Phi(\Sigma))$ :

$$\beta_{\Sigma}(M') \models_{\Sigma} F \quad \text{iff} \quad M' \models'_{\Phi(\Sigma)} \alpha_{\Sigma}(F),$$

2. for all  $\Sigma \in \mathbf{Sig}$ ,  $F \in \mathbf{Sen}(\Sigma)$ :

$$\gamma_{\Sigma}(\vdash_{\Sigma} F) = \vdash'_{\Phi(\Sigma)} \alpha_{\Sigma}(F).$$

With the obvious choices for identity and composition, we obtain the category  $\mathcal{LOG}$  of logics and comorphisms.

Recall that  $\beta_{\Sigma}$  is a morphism in the category  $\mathcal{CAT}^{op}$  and thus the same as a functor  $\beta_{\Sigma} : \mathbf{Mod}'(\Phi(\Sigma)) \rightarrow \mathbf{Mod}(\Sigma)$ ; thus,  $\beta_{\Sigma}(M')$  is a well-formed functor application. Note that the syntax and the proof theory are translated from  $\mathbb{I}$  to  $\mathbb{I}'$ , whereas the model theory is translated in the opposite direction. The two conditions on comorphisms are

truth preservation conditions: The model and proof theory translations must preserve model and proof theoretical truth, respectively.

Just like logics can be decomposed into syntax, model theory, and proof theory, a logic comorphism consists of a

- a syntax translation  $(\Phi, \alpha) : (\mathbf{Sig}, \mathbf{Sen}) \rightarrow (\mathbf{Sig}', \mathbf{Sen}')$ ,
- for a given syntax translation  $(\Phi, \alpha)$ , a model translation  $\beta : (\mathbf{Mod}, \models) \rightarrow (\mathbf{Mod}', \models')$  which must satisfy condition 1.,
- for a given syntax translation  $(\Phi, \alpha)$ , a proof translation  $\gamma : (\mathbf{Pf}, \vdash) \rightarrow (\mathbf{Pf}', \vdash')$  which must satisfy condition 2..

**Remark 4.** Continuing Rem. 3, we observe that if logics are slices  $\mathbb{I}_i : \mathbf{Sig}_i \rightarrow (U \downarrow V)$ , then a logic comorphisms from  $\mathbb{I}_1$  to  $\mathbb{I}_2$  is a functor  $\Phi : \mathbf{Sig}_1 \rightarrow \mathbf{Sig}_2$  together with a natural transformation from  $\mathbb{I}_1$  to  $\mathbb{I}_2 \circ \Phi$ . Thus,  $\mathcal{LOG}$  can be seen as the lax slice category of objects over  $(U \downarrow V)$ .

**Comorphism Modifications** The category of logics and logic comorphisms can be turned into a 2-category. Again we are inspired by the existing notion for institutions, for which the 2-cells are called institution *comorphism modifications* (Diaconescu (2002)):

**Definition 10 (Modifications).** Assume two logic comorphisms  $\mu^i = (\Phi^i, \alpha^i, \beta^i, \gamma^i) : \mathbb{I} \rightarrow \mathbb{I}'$  for  $i = 1, 2$ . A **comorphism modification** from  $\mu^1$  to  $\mu^2$  is a natural transformation  $m : \Phi_1 \rightarrow \Phi_2$  such that the following diagrams commute:

$$\begin{array}{ccc}
 \begin{array}{c}
 \mathbf{Sen}'(\Phi^1(\Sigma)) \\
 \uparrow \quad \downarrow \\
 \mathbf{Sen}(\Sigma) \qquad \mathbf{Sen}'(m_\Sigma) \\
 \downarrow \quad \uparrow \\
 \mathbf{Sen}'(\Phi^2(\Sigma))
 \end{array}
 &
 \begin{array}{c}
 \mathbf{Mod}'(\Phi^1(\Sigma)) \\
 \uparrow \quad \downarrow \\
 \mathbf{Mod}(\Sigma) \qquad \mathbf{Mod}'(m_\Sigma) \\
 \downarrow \quad \uparrow \\
 \mathbf{Mod}'(\Phi^2(\Sigma))
 \end{array}
 &
 \begin{array}{c}
 \mathbf{Pf}'(\Phi^1(\Sigma)) \\
 \uparrow \quad \downarrow \\
 \mathbf{Pf}(\Sigma) \qquad \mathbf{Pf}'(m_\Sigma) \\
 \downarrow \quad \uparrow \\
 \mathbf{Pf}'(\Phi^2(\Sigma))
 \end{array}
 \end{array}$$

The diagram consists of three separate commutative squares. The top-left square involves the Sen category. The top-right square involves the Mod category. The bottom square involves the Pf category. Arrows are labeled with morphisms:  $\alpha_\Sigma^1$  and  $\alpha_\Sigma^2$  in the Sen squares;  $\beta_\Sigma^1$  and  $\beta_\Sigma^2$  in the Mod square; and  $\gamma_\Sigma^1$  and  $\gamma_\Sigma^2$  in the Pf square. Vertical arrows represent the identity morphism for each category. The middle row of squares ( $\mathbf{Sen}'$ ,  $\mathbf{Mod}'$ ,  $\mathbf{Pf}'$ ) are connected by vertical arrows labeled  $m_\Sigma$ , which are the comorphism modifications.

where the diagram for the model theory is drawn in  $\mathcal{CAT}$ .

Thus, a comorphism modification  $m$  is a family  $(m_\Sigma)_{\Sigma \in \mathbf{Sig}}$  of  $\mathbb{I}'$ -signature morphisms  $m_\Sigma : \Phi^1(\Sigma) \rightarrow \Phi^2(\Sigma)$ . It can be understood as modifying  $\mu_1$  in order to make it equal to  $\mu_2$ : For example, the sentence translations  $\alpha_\Sigma^1 : \mathbf{Sen}(\Sigma) \rightarrow \mathbf{Sen}'(\Phi^1(\Sigma))$  and  $\alpha_\Sigma^2 :$

$\mathbf{Sen}(\Sigma) \rightarrow \mathbf{Sen}'(\Phi^2(\Sigma))$  may be quite different. Yet, by composing  $\alpha_\Sigma^1$  with  $\mathbf{Sen}'(m_\Sigma)$ , the difference can be bridged. We obtain a comorphism modification if syntax, model, and proof translation can be bridged uniformly, i.e., by using the respective translations induced by  $m_\Sigma$ .

### 3.3. Meta-Logics

Logic comorphisms  $\mathbb{I} \rightarrow \mathbb{I}'$  permit representing a logic  $\mathbb{I}$  in a logic  $\mathbb{I}'$ . An important special case arises when this representation is **adequate** in the intuitive sense that the semantics of  $\mathbb{I}$  is preserved when representing it in  $\mathbb{I}'$ . Then our understanding of  $\mathbb{I}'$  can be applied to study properties of  $\mathbb{I}$ . If we use a fixed logic  $\mathbb{M} = \mathbb{I}'$ , in which multiple other logics are represented adequately, we speak of **logic encodings** in a **meta-logic**. Occasionally, the term “universal logic” is used instead of “meta-logic”, e.g., in Tarlecki (1996), but we avoid it here to prevent confusion with the general field of “universal logic”, which investigates common structures of all logics.

Alternatively, instead of giving a logic comorphism from a given logic  $\mathbb{I}$  to  $\mathbb{M}$ , we can start with a partial logic – e.g., an institution or even only a category **Sig** – and translate only that into  $\mathbb{M}$ . The missing components of a logic can then be inherited – often called **borrowed** – from  $\mathbb{M}$ . We speak of **logic definitions**.

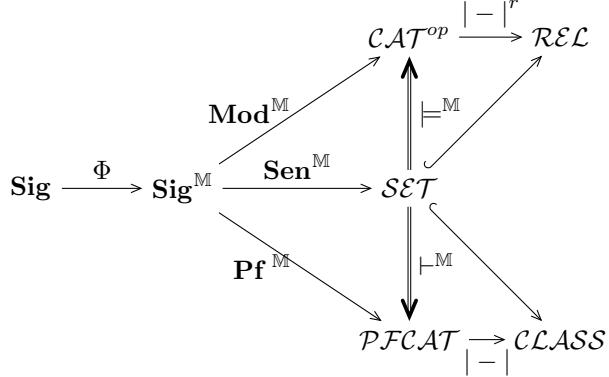
Similarly, a meta-logic can be applied to define or encode logic comorphisms so that we have four cases in the end: defining/encoding a logic (comorphism). In the following we will derive the general properties of these four concepts for an arbitrary meta-logic  $\mathbb{M}$ . Then Sect. 4 will give the concrete definition of our meta-logic  $\mathbb{M}$  as well as examples.

**Defining Logics** Given a signature category **Sig**, to **define a logic** in a meta-logic  $\mathbb{M}$  means to relate **Sig** to  $\mathbb{M}$  in such a way that the syntax, model theory, and proof theory of  $\mathbb{M}$  can be borrowed to extend **Sig** to a logic.

**Notation 3.** If  $\models$  is a family  $(\models_\Sigma)_{\Sigma \in \mathbf{Sig}}$  and  $\Phi : \mathbf{Sig}' \rightarrow \mathbf{Sig}$  is a functor, then we write  $\models \circ \Phi$  for the family  $(\models_{\Phi(\Sigma)})_{\Sigma \in \mathbf{Sig}'}$ .  $\vdash \circ \Phi$  is defined accordingly. Considering Rem. 2, this is simply the composition of a functor and a natural transformation.

**Definition 11.** Let  $\mathbb{M}$  be a logic. A **logic definition** in  $\mathbb{M}$  consists of a category **Sig** and a functor  $\Phi : \mathbf{Sig} \rightarrow \mathbf{Sig}^\mathbb{M}$ . Such a logic definition induces a logic  $\mathbb{M} \circ \Phi = (\mathbf{Sig}, \mathbf{Sen}^\mathbb{M} \circ \Phi, \mathbf{Mod}^\mathbb{M} \circ \Phi, \models^\mathbb{M} \circ \Phi, \mathbf{Pf}^\mathbb{M} \circ \Phi, \vdash^\mathbb{M} \circ \Phi)$ .

**Remark 5.** With the notation of Rem. 2, this yields the diagram



Moreover,  $\Phi$  induces a canonical comorphism  $M \circ \Phi \rightarrow M$ .

**Theorem 1 (Logic Definitions).**  $M \circ \Phi$  is indeed a logic. Moreover, it is (strongly) sound or (strongly) complete if  $M$  is.

*Proof.* That  $M \circ \Phi$  is a logic follows immediately from the diagram in Rem. 5. In particular,  $\models^M \circ \Phi$  and  $\vdash^M \circ \Phi$  are natural transformation because they arise by composing a natural transformation with a functor.

It is inherent in Def. 11 that  $\Theta \vdash_{\Sigma}^{M \circ \Phi} F$  iff  $\Theta \vdash_{\Phi(\Sigma)}^M F$ , and similarly for the entailment relation. That yields the (strong) soundness/completeness result.  $\square$

**Remark 6.** In Rem. 3, we stated that institutions are functors into  $(U \downarrow V)$ , and thus in particular  $M : \text{Sig}^M \rightarrow (U \downarrow V)$ . Then  $M \circ \Phi$  is indeed the composition of two functors (which justifies our notation).

**Encoding Logics** While a logic definition starts with a signature category and borrows all other notions from  $M$ , a **logic encoding** starts with a whole logic  $\mathbb{I} = (\text{Sig}, \text{Sen}, \text{Mod}, \models, \text{Pf}, \vdash)$  and a functor  $\Phi : \text{Sig} \rightarrow \text{Sig}^M$ :

**Definition 12 (Logic Encoding).** An **encoding** of a logic  $\mathbb{I}$  in  $M$  is a logic comorphism  $(\Phi, \alpha, \beta, \gamma) : \mathbb{I} \rightarrow M$ . (Equivalently, we can say that  $(id, \alpha, \beta, \gamma)$  is a logic comorphism  $\mathbb{I} \rightarrow M \circ \Phi$ .)

When encoding logics in a meta-logic, a central question is **adequacy**. Intuitively, adequacy means that the encoding is correct with respect to the encoded logic. More formally, a logic encoding yields two logics  $\mathbb{I}$  and  $M \circ \Phi$ , and adequacy is a statement about the relation between them.

Ideally,  $\alpha$ ,  $\beta$ , and  $\gamma$  are isomorphisms, in which case  $\mathbb{I}$  and  $M \circ \Phi$  are isomorphic in the category  $\mathcal{LOG}$ . However, often it is sufficient to show that the consequence relations in  $\mathbb{I}$  and  $M \circ \Phi$  coincide. This motivates the following definition and theorem:

**Definition 13 (Adequate Encodings).** In the situation of Def. 12, the encoding is called **adequate** if

- each  $\beta_\Sigma$  is surjective on objects (model theoretical adequacy),

- each  $\gamma_\Sigma$  is “surjective” on morphisms in the following sense: for any  $A, B$ , if there is a morphism from  $\gamma_\Sigma(A)$  to  $\gamma_\Sigma(B)$ , then there is also one from  $A$  to  $B$  (proof theoretical adequacy).

Model theoretical adequacy is just the *model expansion property* (e.g., Cerioli and Meseguer (1997)); intuitively, it means that a property that holds for all models of  $\mathbf{Mod}^M(\Phi(\Sigma))$  can be used to establish the corresponding property for all models of  $\Sigma$ . Proof theoretical adequacy means that proofs found in  $M \circ \Phi$  give rise to corresponding proofs in  $\mathbb{I}$ . More precisely:

**Theorem 2 (Adequacy).** Assume an adequate logic encoding  $(\Phi, \alpha, \beta, \gamma)$  of  $\mathbb{I}$  in  $M$ . Then for all  $\mathbb{I}$ -theories  $(\Sigma, \Theta)$  and all  $\Sigma$ -sentences  $F$ :

$$\begin{aligned}\Theta \models_{\Sigma}^{\mathbb{I}} F &\quad \text{iff} \quad \alpha_\Sigma(\Theta) \models_{\Phi(\Sigma)}^M \alpha_\Sigma(F) \\ \Theta \vdash_{\Sigma}^{\mathbb{I}} F &\quad \text{iff} \quad \alpha_\Sigma(\Theta) \vdash_{\Phi(\Sigma)}^M \alpha_\Sigma(F)\end{aligned}$$

In particular,  $\mathbb{I}$  is strongly sound or complete if  $M$  is.

*Proof.* It is easy to show that the left-to-right implications hold for any logic comorphism.

The right-to-left implications are independent and only require the respective adequacy assumption. The model theoretical result is essentially the known borrowing result for institutions (see Cerioli and Meseguer (1997)). For the proof theoretical result, recall that  $\gamma_\Sigma$  is a functor  $\mathbf{Pf}^{\mathbb{I}}(\Sigma) \rightarrow \mathbf{Pf}^M(\Phi(\Sigma))$ ; since provability is defined by the existence of morphisms in the proof category, the proof theoretical adequacy yields the result.

Then the soundness and completeness results follow by using Thm. 1.  $\square$

Mixing Defining and Encoding Logic definitions, where we start with **Sig** and inherit all other components from  $M$ , and logic encodings, where we start with a logic and encode all components in  $M$ , are opposite extremes. We can also start, for example, with an institution  $\mathbb{I} = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$  and give an institution comorphism  $(\Phi, \alpha, \beta) : \mathbb{I} \rightarrow M$ : This encodes syntax and model theory of  $\mathbb{I}$  in  $M$  and inherits the proof theory. Dually, we can start with syntax and proof theory  $\mathbb{I} = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Pf}, \vdash)$  and give a comorphism  $(\Phi, \alpha, \gamma) : \mathbb{I} \rightarrow M$ : This encodes syntax and proof theory and inherits the model theory.

The former of these two is often used to apply an implementation of the proof theoretical consequence relation of  $M$  to reason about the model theoretical consequence relation of  $\mathbb{I}$ , a technique known as *borrowing* (Cerioli and Meseguer (1997)). The borrowing theorem can be recovered as a special case of the above, and we briefly state it in our notation:

**Theorem 3.** Let  $\mathbb{I}$  be an institution and  $M$  a logic, and let  $\mu = (\Phi, \alpha, \beta)$  be an institution comorphism from  $\mathbb{I}$  to  $M$  (qua institution). Then we extend  $\mathbb{I}$  to a logic  $\mathbb{I}'$  by putting  $\mathbf{Pf}' := \mathbf{Pf}^M \circ \Phi$  and  $\vdash' := (\vdash^M \circ \Phi) \circ \alpha$ . Moreover, we have that  $(\Phi, \alpha, \beta, id) : \mathbb{I}' \rightarrow M$  is a logic comorphism that is an adequate encoding if all  $\beta_\Sigma$  are surjective on objects.

*Proof.* It follows from basic properties of category theory that  $\mathbb{I}'$  is indeed a logic. Model

theoretical adequacy holds due to the surjectivity assumption, and proof theoretical follows because the proof translation is the identity.  $\square$

**Defining Logic Comorphisms** Now we turn to the treatment of logic comorphisms. To define a logic comorphism we give a family of signature morphisms in  $\mathbb{M}$ :

**Definition 14 (Defining Comorphisms).** Assume two logic definitions  $\Phi^i : \mathbf{Sig}^i \rightarrow \mathbf{Sig}^{\mathbb{M}}$  for  $i = 1, 2$ . A **comorphism definition** consists of a functor  $\Phi : \mathbf{Sig}^1 \rightarrow \mathbf{Sig}^2$  and a natural transformation  $m : \Phi^1 \rightarrow \Phi^2 \circ \Phi$ . Such a comorphism definition induces a logic comorphism  $(\Phi, \mathbb{M} \circ m) = (\Phi, \mathbf{Sen}^{\mathbb{M}} \circ m, \mathbf{Mod}^{\mathbb{M}} \circ m, \mathbf{Pf}^{\mathbb{M}} \circ m)$  from  $\mathbb{M} \circ \Phi^1$  to  $\mathbb{M} \circ \Phi^2$ .

**Remark 7.** With the notation of Rem. 5, this yields the diagram

$$\begin{array}{ccccc}
 \mathbf{Sig}^1 & & \mathcal{CAT}^{op} & \xrightarrow{|-\rvert^r} & \mathcal{REL} \\
 \downarrow \Phi & \searrow \Phi^1 & \nearrow \mathbf{Mod}^{\mathbb{M}} & \uparrow \vdash^{\mathbb{M}} & \swarrow \\
 \mathbf{Sig}^{\mathbb{M}} & \xrightarrow{\mathbf{Sen}^{\mathbb{M}}} & \mathcal{SET} & & \\
 \downarrow m & \uparrow \Phi^2 & \downarrow \vdash^{\mathbb{M}} & \uparrow & \searrow \\
 \mathbf{Sig}^2 & & \mathcal{PFCAT} & \xrightarrow{|-\rvert} & \mathcal{CLASS}
 \end{array}$$

where double arrows indicate natural transformations as before. Thus,  $m_{\Sigma}$  is a family of  $\mathbf{Sig}^{\mathbb{M}}$ -morphisms  $m_{\Sigma} : \Phi^1(\Sigma) \rightarrow \Phi^2(\Phi(\Sigma))$  for  $\Sigma \in \mathbf{Sig}^1$ . Sentence, proof, and model translation are given by  $\mathbf{Sen}^{\mathbb{M}}(m_{\Sigma})$ ,  $\mathbf{Mod}^{\mathbb{M}}(m_{\Sigma})$ , and  $\mathbf{Pf}^{\mathbb{M}}(m_{\Sigma})$ .

**Theorem 4 (Defining Comorphisms).**  $(\Phi, \mathbb{M} \circ m)$  is indeed a logic comorphism.

*Proof.* From the diagram in Rem. 7, it is clear that the types of the components of  $(\Phi, \mathbb{M} \circ m)$  are correct because they arise by composing functors and natural transformations.

It remains to show Condition 1. and 2. in Def. 9. For the latter, we substitute the respective definitions for  $\vdash$ ,  $\vdash'$ ,  $\alpha$ , and  $\gamma$  and obtain

$$(\mathbf{Pf}^{\mathbb{M}} \circ m)_{\Sigma}((\vdash^{\mathbb{M}} \circ \Phi^1)_{\Sigma} F) = (\vdash^{\mathbb{M}} \circ \Phi^2)_{\Phi(\Sigma)} (\mathbf{Sen}^{\mathbb{M}} \circ m)_{\Sigma}(F).$$

This simplifies to

$$\mathbf{Pf}^{\mathbb{M}}(m_{\Sigma})(\vdash^{\mathbb{M}}_{\Phi^1(\Sigma)} F) = \vdash^{\mathbb{M}}_{\Phi^2(\Phi(\Sigma))} \mathbf{Sen}^{\mathbb{M}}(m_{\Sigma})(F)$$

which is exactly the condition from Def. 4 for the morphism  $m_{\Sigma}$ .

In the same way, Condition 1. simplifies to the satisfaction condition from Def. 2.  $\square$

**Remark 8.** Recalling Rem. 3 and continuing Rem. 6, we observe that  $\mathbb{M} \circ m$  is a natural transformation  $\mathbb{M} \circ \Phi^1 \rightarrow \mathbb{M} \circ \Phi^2 \circ \Phi$  between functors  $\mathbf{Sig} \rightarrow (U \downarrow V)$ . Thus,  $(\Phi, \mathbb{M} \circ m) :$

$\mathbb{M} \circ \Phi^1 \rightarrow \mathbb{M} \circ \Phi^2$  is indeed a morphism in the lax slice category of objects over  $(U \downarrow V)$  (which justifies our notation). This yields a simpler, more abstract proof of Thm. 4.

Encoding Logic Comorphisms Finally, we study the encoding of existing logic comorphisms.

**Definition 15 (Encoding Comorphisms).** Assume a logic comorphism  $\mu : \mathbb{I}^1 \rightarrow \mathbb{I}^2$  and two logic encodings  $\mu^i : \mathbb{I}^i \rightarrow \mathbb{M}$ . An **encoding** of  $\mu$  relative to  $\mu_1$  and  $\mu_2$  is a logic comorphism modification  $\mu^1 \rightarrow \mu^2 \circ \mu$ .

To understand this definition better, let us assume a comorphism  $\mu = (\Phi, \alpha, \beta, \gamma) : \mathbb{I}^1 \rightarrow \mathbb{I}^2$  and two encodings  $\mu^i = (\Phi^i, \alpha^i, \beta^i, \gamma^i) : \mathbb{I}^i \rightarrow \mathbb{M}$ . Then an encoding  $m$  of  $\mu$  is a natural transformation  $\Phi^1 \rightarrow \Phi^2 \circ \Phi$  such that the following diagrams commute for all  $\Sigma \in \mathbf{Sig}^1$ :

$$\begin{array}{ccc}
\mathbf{Sen}^1(\Sigma) & \xrightarrow{\alpha_\Sigma} & \mathbf{Sen}^2(\Phi(\Sigma)) \\
\alpha_\Sigma^1 \downarrow & & \downarrow \alpha_{\Phi(\Sigma)}^2 \\
\mathbf{Sen}^{\mathbb{M}}(\Phi^1(\Sigma)) & \xrightarrow{\mathbf{Sen}^{\mathbb{M}}(m_\Sigma)} & \mathbf{Sen}^{\mathbb{M}}(\Phi^2(\Phi(\Sigma))) \\
& & \\
\mathbf{Mod}^1(\Sigma) & \xleftarrow{\beta_\Sigma} & \mathbf{Mod}^2(\Phi(\Sigma)) \\
\beta_\Sigma^1 \uparrow & & \uparrow \beta_{\Phi(\Sigma)}^2 \\
\mathbf{Mod}^{\mathbb{M}}(\Phi^1(\Sigma)) & \xleftarrow{\mathbf{Mod}^{\mathbb{M}}(m_\Sigma)} & \mathbf{Mod}^{\mathbb{M}}(\Phi^2(\Phi(\Sigma))) \\
& & \\
\mathbf{Pf}^1(\Sigma) & \xrightarrow{\gamma_\Sigma} & \mathbf{Pf}^2(\Phi(\Sigma)) \\
\gamma_\Sigma^1 \downarrow & & \downarrow \gamma_{\Phi(\Sigma)}^2 \\
\mathbf{Pf}^{\mathbb{M}}(\Phi^1(\Sigma)) & \xrightarrow{\mathbf{Pf}^{\mathbb{M}}(m_\Sigma)} & \mathbf{Pf}^{\mathbb{M}}(\Phi^2(\Phi(\Sigma)))
\end{array}$$

Intuitively,  $m_\Sigma : \Phi^1(\Sigma) \rightarrow \Phi^2(\Phi(\Sigma))$  is a signature morphism in the meta-logic such that  $\mathbf{Sen}^{\mathbb{M}}(m_\Sigma)$  has the same effect as  $\alpha_\Sigma$ . This is particularly intuitive in the typical case where  $\alpha_\Sigma^1$  and  $\alpha_{\Phi(\Sigma)}^2$  are bijections. Similarly,  $\mathbf{Mod}^{\mathbb{M}}(m_\Sigma)$  and  $\mathbf{Pf}^{\mathbb{M}}(m_\Sigma)$  must have the same effect as  $\beta_\Sigma$  and  $\gamma_\Sigma$ .

#### 4. A Meta-Logic

We will now define a specific logic  $\mathbb{M} = (\mathbf{Sig}^{\mathbb{M}}, \mathbf{Sen}^{\mathbb{M}}, \mathbf{Mod}^{\mathbb{M}}, \models^{\mathbb{M}}, \mathbf{Pf}^{\mathbb{M}}, \vdash^{\mathbb{M}})$  that we use as our meta-logic.  $\mathbb{M}$  is based on the dependent type theory of LF.

We could define  $\mathbb{M}$  such that the signatures of  $\mathbb{M}$  are the LF signatures. There are several choices for the sentences of such a logic, the most elegant one uses the types

as the sentences. In that case the proof categories are the categories of contexts, and models are usually based on locally cartesian closed categories ([Cartmell \(1986\)](#); [Seely \(1984\)](#)). For example, we gave a (sound and complete) logic in this style but without the terminology used here in [Awodey and Rabe \(2011\)](#).

However, this is not our intention here. We want to use  $\mathbb{M}$  as a logical framework, in which *all* components of an object logic are represented in terms of the syntax of the meta-logic. In order to capture all aspects of a logic, the  $\mathbb{M}$ -signatures must be more complex.

Below we first define the syntax, proof theory, and model theory of  $\mathbb{M}$  in Sect. [4.1](#), [4.2](#), and [4.3](#), respectively, before defining  $\mathbb{M}$  and discussing its basic properties in Sect. [4.4](#). We use propositional modal logic as a running example.  $\mathbb{M}$  is foundation-independent: Object logic models are represented using the syntax of the meta-logic in a way that does not commit to a particular foundation of mathematics. We discuss the use of foundations in more detail in Sect. [4.5](#).

#### 4.1. Syntax

The encoding of logical syntax in LF has been well-studied, and we motivate our definitions of  $\mathbb{M}$  signatures by analyzing existing logic encodings in LF.

Firstly, logic encodings in LF usually employ a distinguished type `form` for the formulas and a distinguished judgment `ded` : `form` → `type` for the truth judgment. But the names of these symbols may differ, and in some cases `form` is not even a symbol. Therefore, we fix a signature *Base* as follows

```
form  :  type
ded   :  form → type
```

Now by considering morphisms out of *Base*, i.e., pairs of a signature  $\Sigma$  and a morphism  $base : Base \rightarrow \Sigma$ , we can explicate these distinguished types.  $base(\text{form})$  is a type over  $\Sigma$  representing the syntactic class of formulas. And  $base(\text{ded})$  is a type family over  $\Sigma$  representing the truth of formulas. By investigating morphisms out of *Base*, we are restricting attention to those LF signatures that define formulas and truth – the central properties that separate logic encodings from other LF signatures.

Secondly, we want to represent models of a signature  $\Sigma$  as signature morphisms out of  $\Sigma$ . However, models often have components that are not mentioned in the syntax, e.g., the set of worlds in a Kripke model. Therefore, we will use two LF-signatures  $\Sigma^{syn}$  and  $\Sigma^{mod}$  to represent syntax and model theory separately.

$\Sigma^{syn}$  declares the syntax of a logical language, i.e., LF symbols that represent syntactic classes (e.g., terms or formulas), sorts, functions, predicates, etc.  $\Sigma^{mod}$  declares the model theory, i.e., LF symbols that axiomatize the properties of models, typically in terms of a foundational semantic language like set theory, type theory, or category theory. A morphism  $\mu : \Sigma^{syn} \rightarrow \Sigma^{mod}$  interprets the syntax in terms of the model theory and represents the inductive interpretation function that translates syntax into the semantic realm. Finally individual models are represented as morphisms out of  $\Sigma^{mod}$ .

Finally, because of the symmetry between model and proof theory, it is elegant to split

syntax and proof theory as well by introducing an LF-signature  $\Sigma^{pf}$  and a morphism  $\pi : \Sigma^{syn} \rightarrow \Sigma^{pf}$ .  $\Sigma^{pf}$  declares the proof theory, i.e., LF symbols that represent judgments about the syntax and inference rules for them. It typically deviates from  $\Sigma^{syn}$  by declaring auxiliary syntax such as signed formulas in a tableaux calculus. In many cases,  $\pi$  is an inclusion.

Then we arrive at the following diagram:

$$\begin{array}{ccc} & & \Sigma^{pf} \\ & \nearrow \pi & \\ Base & \xrightarrow{\text{base}} & \Sigma^{syn} \\ & \searrow \mu & \\ & & \Sigma^{mod} \end{array}$$

The symbol **ded** plays a crucial duplicate role. *Proof theoretically*, the  $\Sigma^{pf}$ -type

$$\pi(\text{base}(\text{ded}) F)$$

is the type of proofs of  $F$ . A proof from assumptions  $\Gamma$  is a  $\Sigma^{pf}$ -term in context  $\Gamma$ . This yields the well-known Curry-Howard representation of proofs as terms (Martin-Löf (1996); Pfenning (2001)). *Model theoretically*, the  $\Sigma^{mod}$ -type  $\mu(\text{base}(\text{form}))$  is the type of truth values, and  $\mu(\text{base}(\text{ded}))$  is a predicate on it giving the designated truth values. Below we will define models as morphisms  $m : \Sigma^{mod} \rightarrow M$ , and  $(M, m)$  satisfies  $F$  iff the type

$$m(\mu(\text{base}(\text{ded}) F))$$

is inhabited over  $M$ . This yields the well-known representation of models as morphisms out of an initial object in a suitable category (Goguen et al. (1978); Lawvere (1963)). Thus, **ded** is the mediator between proof and model theory, and  $\Sigma^{pf}$  and  $\Sigma^{mod}$  encode the two different ways to give meaning to  $\Sigma^{syn}$ .

We summarize the above in the following definition:

**Definition 16 (Signatures).** The **signatures** of  $\mathbb{M}$  are tuples

$$\Sigma = (\Sigma^{syn}, \Sigma^{pf}, \Sigma^{mod}, \text{base}, \pi, \mu)$$

forming a diagram of LF signatures as given above.  $\text{base}(\text{ded})$  must be a constant.

The restriction of  $\text{base}(\text{ded})$  to constants is a bit inelegant. But it is not harmful in practice and permits to exclude some degenerate cases that would make later definitions more complicated.

As expected, sentences are the LF terms of type **form** over  $\Sigma^{syn}$ :

**Definition 17 (Sentences).** Given an  $\mathbb{M}$  signature  $\Sigma$  as in Def. 16, we define the **sentences** by

$$\mathbf{Sen}^{\mathbb{M}}(\Sigma) = \{F \mid \cdot \vdash_{\Sigma^{syn}} F : \text{base}(\text{form})\}$$

**Example 7 (Modal Logic).** For our simple modal logic  $\mathbb{ML}$  from Ex. 1, we use a signature  $ML^{syn}$  as follows:

```

form  : type
▷    : form → form → form
□    : form → form
ded   : form → type

```

Here  $\triangleright$  and  $\square$  are formula constructors for implication and necessity. The morphism  $base : Base \rightarrow ML^{syn}$  is simply an inclusion. If we want to declare propositional variables, we can add them using declarations  $p : \text{form}$ . Let  $PQ^{syn}$  be the extension of  $ML^{syn}$  with  $p : \text{form}$  and  $q : \text{form}$ . Then an example sentence is  $T = \square p \triangleright p \in \mathbf{Sen}^{\mathbb{M}}(PQ)$ , an instance of the axiom scheme T. As here for  $\triangleright$ , we will use intuitive infix and bracket elimination rules when giving example expressions.

**Definition 18 (Signature Morphism).** An  $\mathbb{M}$ -signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$  is a tuple  $(\sigma^{syn}, \sigma^{pf}, \sigma^{mod})$  such that

- the diagram below commutes,
  - for every  $F \in \mathbf{Sen}^{\mathbb{M}}(\Sigma)$  there is an  $F'$  such that  $\sigma^{syn}(base(\text{ded}) F) = base'(\text{ded}) F'$ .
- Identity and composition for  $\mathbf{Sig}^{\mathbb{M}}$ -morphisms are defined component-wise.

$$\begin{array}{ccccc}
& & \Sigma^{pf} & & \Sigma'^{pf} \\
& \pi \nearrow & \xrightarrow{\sigma^{pf}} & & \searrow \pi' \\
\Sigma^{syn} & \xrightarrow{\sigma^{syn}} & \Sigma'^{syn} & \xrightarrow{\mu'} & \\
& \searrow \mu & & \swarrow \mu' & \\
& \Sigma^{mod} & \xrightarrow{\sigma^{mod}} & \Sigma'^{mod} &
\end{array}$$

$\sigma$  is **simple** if additionally  $\sigma^{syn} \circ base = base'$ .

**Remark 9.** The diagram in Def. 18 omits  $Base$  and thus does not imply  $\sigma^{syn} \circ base = base'$ , which only holds for simple morphisms. It is natural to restrict attention to simple morphisms. In particular, the second condition in Def. 18 is redundant for simple morphisms as we always have  $F' = \sigma^{syn}(F)$ .

However, our slightly weaker definition is crucial for the representation of many logic translations as we will see in Ex. 18 and discuss in Rem. 18.

Def. 18 is just strong enough to guarantee the existence of a **sentence translation**:

**Definition 19 (Sentence Translation).** In the situation of Def. 18, we put

$$\mathbf{Sen}^{\mathbb{M}}(\sigma)(F) = \text{the } F' \text{ such that } \sigma^{syn}(base(\text{ded}) F) = base'(\text{ded}) F'$$

This is well-defined because  $F'$  exists by Def. 18; and if it exists, it is necessarily unique.

**Remark 10.** In particular, sentence translation along simple morphisms is just the application of the morphism  $\sigma^{syn}$ :  $\sigma^{syn}(base(o)) = base'(o)$  and  $\mathbf{Sen}^{\mathbb{M}}(\sigma)(F) = \sigma^{syn}(F)$ .

**Example 8 (Continued).** A simple  $\mathbf{Sig}^{\mathbb{M}}$ -morphism  $w : PQ \rightarrow PQ$  can contain the component

$$w^{syn} = id, q := \square q, p := q$$

where  $id$  is the identity morphism for  $PQ^{syn}$ . Then we have  $w^{syn}(\text{form}) = \text{form}$  and  $w^{syn}(\text{ded}) = \text{ded}$ , and  $w^{syn}(\text{ded } T) = \text{ded } \square q \supset q$ . Thus, we obtain,  $\mathbf{Sen}^{\mathbb{M}}(w)(T) = \square q \supset q$ .

#### 4.2. Proof Theory

The idea behind the proof theory of  $\mathbb{M}$  is the Curry-Howard correspondence (Curry and Feys (1958); Howard (1980)): judgments  $(E_1, \dots, E_n)$  correspond to contexts  $x_1 : E_1, \dots, x_n : E_n$  where the variables act as names for hypotheses. Moreover, a morphism from  $\Gamma = x_1 : E_1, \dots, x_m : E_m$  to  $\Gamma' = x_1 : F_1, \dots, x_n : F_n$  provides one proof over  $\Gamma$  for every judgment assumed in  $\Gamma'$ , i.e., it is a tuple  $p_1^n$  of terms in context  $\Gamma \vdash_{\Sigma^{pf}} p_i : F_i$ . Such a tuple is simply a substitution from  $\Gamma'$  to  $\Gamma$ , and the proof categories are the well-understood categories of contexts and substitutions. The finite products are given by concatenations of contexts (possibly using  $\alpha$ -renaming to avoid duplicate variables).

**Definition 20 (Proof Categories).** Given an  $\mathbb{M}$  signature  $\Sigma$  as in Def. 16, we define the proof category  $\mathbf{Pf}^{\mathbb{M}}(\Sigma)$  as the dual of the category of contexts and substitutions over  $\Sigma^{pf}$ :

- the objects are the contexts over  $\Sigma^{pf}$ ,
- the morphisms from  $\Gamma$  to  $\Gamma'$  are the substitutions from  $\Gamma'$  to  $\Gamma$ .

**Example 9 (Continued).** For our modal logic, we can define  $ML^{pf}$  by extending  $ML^{syn}$  with a calculus for provability. Thus,  $\pi_{ML}$  is an inclusion. There are a number of ways to do that (see Avron et al. (1998)); for a simple Hilbert-style calculus,  $ML^{pf}$  consists of axioms for propositional logic and the declarations

$$\begin{aligned} mp &: \{x : \text{form}\} \{y : \text{form}\} \text{ded } x \supset y \rightarrow \text{ded } x \rightarrow \text{ded } y \\ nec &: \{x : \text{form}\} \text{ded } x \rightarrow \text{ded } \square x \\ K &: \{x : \text{form}\} \{y : \text{form}\} \text{ded } \square(x \supset y) \supset (\square x \supset \square y) \end{aligned}$$

Note how the  $\Pi$ -binder  $\{x : \text{form}\}$  is used to declare schema variables.

Similarly, we define  $PQ^{pf}$  by extending  $ML^{pf}$  with  $p : \text{form}$  and  $q : \text{form}$ . Then the proof category  $\mathbf{Pf}^{\mathbb{M}}(PQ)$  contains objects like

$$\Gamma_1 = x : \text{ded } \square(p \supset q), y : \text{ded } \square p$$

and

$$\Gamma_2 = z : \text{ded } \square q.$$

A  $\mathbf{Pf}^{\mathbb{M}}(PQ)$ -morphism from  $\Gamma_1$  to  $\Gamma_2$  is the substitution

$$\gamma_1 = z := mp \underline{\square p} \underline{\square q} (mp \underline{\square(p \supset q)} \underline{\square p} \supset \underline{\square q} (K \underline{p} \underline{q}) x) y$$

It gives a proof of the goal labelled  $z$  in terms of the two assumptions labelled  $x$  and  $y$ . To enhance readability, we have underlined the arguments that instantiate the schema variables.

Just like the proof categories are the categories of contexts and substitutions over the LF signature  $\Sigma^{pf}$ , the proof translation functor is given by the application of the LF morphism  $\sigma^{pf}$  to contexts and substitutions:

**Definition 21 (Proof Translation).** Given an  $\mathbb{M}$ -signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$  as in Def. 18, the proof translation functor  $\mathbf{Pf}^{\mathbb{M}}(\sigma) : \mathbf{Pf}^{\mathbb{M}}(\Sigma) \rightarrow \mathbf{Pf}^{\mathbb{M}}(\Sigma')$  is defined by

$$\begin{aligned}\mathbf{Pf}^{\mathbb{M}}(\sigma)(\Gamma) &= \sigma^{pf}(\Gamma) && \text{for } \Gamma \in |\mathbf{Pf}^{\mathbb{M}}(\Sigma)| \\ \mathbf{Pf}^{\mathbb{M}}(\sigma)(\gamma) &= \sigma^{pf}(\gamma) && \text{for } \gamma \in \mathbf{Pf}^{\mathbb{M}}(\Sigma)(\Gamma, \Gamma')\end{aligned}$$

**Example 10 (Continued).** Reusing the morphism  $w$  and the object  $\Gamma_1$  from our running example, we obtain

$$\mathbf{Pf}^{\mathbb{M}}(w)(\Gamma_1) = x : \mathbf{ded} \square(q \supset \square q), y : \mathbf{ded} \square q$$

and

$$\mathbf{Pf}^{\mathbb{M}}(w)(\gamma_1) = z := mp \underline{\square q} \underline{\square \square q} (mp \underline{\square(q \supset \square q)} \underline{\square q} \supset \underline{\square \square q} (K \underline{q} \underline{\square q}) x) y$$

which is an  $\mathbf{Pf}^{\mathbb{M}}(PQ)$ -morphism from  $\mathbf{Pf}^{\mathbb{M}}(w)(\Gamma_1)$  to  $\mathbf{Pf}^{\mathbb{M}}(w)(\Gamma_2)$ .

As expected, the truth judgment is given by the image of  $\mathbf{ded}$  in  $\Sigma^{pf}$ :

**Definition 22 (Truth Judgment).** Given an  $\mathbb{M}$  signature  $\Sigma$  as in Def. 16, the truth judgment  $\vdash_{\Sigma}^{\mathbb{M}} F$  is defined as the context  $x : \pi(\mathit{base}(\mathbf{ded}) F)$  for an arbitrary fixed variable  $x$ .

**Example 11 (Continued).** We have

$$\vdash_{PQ}^{\mathbb{M}} \square(p \supset q) \times \vdash_{PQ}^{\mathbb{M}} \square p \cong \Gamma_1$$

$$\vdash_{PQ}^{\mathbb{M}} \square q \cong \Gamma_2$$

The projection out of the product  $\Gamma_1$  are just inclusion substitutions. And the isomorphism in the second case is a variable renaming. Thus, the morphism  $\gamma_1$  proves  $\{\square(p \supset q), \square p\} \vdash_{PQ}^{\mathbb{M}} \square q$ . Similarly, the morphism  $\mathbf{Pf}^{\mathbb{M}}(w)(\gamma_1)$  proves  $\{\square(q \supset \square q), \square q\} \vdash_{PQ}^{\mathbb{M}} \square \square q$ .

#### 4.3. Model Theory

The definition of  $\Sigma^{mod}$  is the most difficult part because  $\Sigma^{mod}$  must declare symbols for all components that are present in a model. Since a model may involve any mathematical object,  $\Sigma^{mod}$  must be expressive enough to define arbitrary mathematical objects. Therefore,  $\Sigma^{mod}$  must include a representation of the whole foundation of mathematics. The foundation is flexible; for example, we have represented Zermelo-Fraenkel set theory, Mizar's set theory, and higher-order logic HOL as foundations in the applications of our framework (Codescu et al. (2011); Horozal and Rabe (2011); Iancu and Rabe (2011)). For simplicity, we will use higher-order logic as the foundation here. We start with our running example:

**Example 12 (Continued).** To describe the model theory of modal logics, we reuse the signature  $HOL$  that was introduced as a running example in Sect. 2.2. We assume it also provides a declaration

$$\forall : (tm A \rightarrow tm \text{ bool}) \rightarrow tm \text{ bool}$$

for universal quantification over elements of  $A$ .

Then  $ML^{mod}$  can be given by extending  $HOL$  with

$$\begin{aligned} \text{worlds} &: tp \\ \text{acc} &: tm \text{ worlds} \rightarrow tm \text{ worlds} \rightarrow tm \text{ bool} \end{aligned}$$

where  $\text{worlds}$  and  $\text{acc}$  represent the set of words and the accessibility relation of a Kripke model. Now the morphism  $\mu_{ML} : ML^{syn} \rightarrow ML^{mod}$  can be defined as

$$\begin{aligned} \text{form} &:= tm \text{ worlds} \rightarrow tm \text{ bool} \\ \supset &:= [f : tm \text{ worlds} \rightarrow tm \text{ bool}] [g : tm \text{ worlds} \rightarrow tm \text{ bool}] \\ &\quad [w : tm \text{ worlds}] (f w \Rightarrow g w) \\ \Box &:= [f : tm \text{ worlds} \rightarrow tm \text{ bool}] \\ &\quad [w : tm \text{ worlds}] \forall [w' : tm \text{ worlds}] (\text{acc } w w' \Rightarrow f w') \\ \text{ded} &:= [f : tm \text{ worlds} \rightarrow tm \text{ bool}] \text{ded} (\forall [w : tm \text{ worlds}] f w) \end{aligned}$$

$\mu_{ML}$  formalizes that formulas are interpreted as functions from the set of worlds to the set of booleans. Consequently,  $\mu_{ML}(\supset)$  takes two and returns one, and  $\mu_{ML}(\Box)$  takes one and returns one such functions. These formalize the interpretation of formulas in Kripke models in the usual way. Finally,  $\mu_{ML}(\text{ded})$  formalizes the satisfaction relation: A formula holds in a model if it is true in all worlds.

To avoid confusion, keep in mind that  $\text{ded}$  formalizes truth in the object-logic  $ML$ , whereas  $\text{ded}$  formalizes truth in the mathematical foundation  $HOL$ , which is used to express the model theory. As common in model theoretical definitions of truth, the truth of formulas is defined in terms of the truth of the mathematical foundation.

The signature  $PQ^{mod}$  arises by extending  $ML^{mod}$  with declarations  $p : \mu(\text{form})$ , i.e.,  $p : tm \text{ worlds} \rightarrow tm \text{ bool}$ , and similarly for  $q$ . The morphism  $\mu_{PQ} : PQ^{syn} \rightarrow PQ^{mod}$  is given by  $\mu$ ,  $p := p$ ,  $q := q$ .

Now a model of  $PQ$  must provide specific values for  $\text{worlds}$ ,  $\text{acc}$ ,  $p$ , and  $q$  in terms of  $HOL$ . Thus, we can represent models as morphisms from  $PQ^{mod}$  to  $HOL$ .

Then we arrive at the following definition of model categories:

**Definition 23 (Model Categories).** Given an  $\mathbb{M}$  signature  $\Sigma$  as in Def. 16, we define the model category  $\mathbf{Mod}^{\mathbb{M}}(\Sigma)$  as the slice category of objects under  $\Sigma^{mod}$ :

- objects are the pairs  $(M, m)$  for an LF-signature morphism  $m : \Sigma^{mod} \rightarrow M$ ,
- morphisms from  $(M, m)$  to  $(M', m')$  are LF-signature morphisms  $\varphi : M \rightarrow M'$  such that  $\varphi \circ m = m'$ ,
- the identity morphism of  $(M, m)$  is  $\text{id}_M$ ,
- the composition of  $\varphi : (M, m) \rightarrow (M', m')$  and  $\varphi' : (M', m') \rightarrow (M'', m'')$  is  $\varphi' \circ \varphi$ .

Here we use an arbitrary LF signature as the codomain of the models to avoid a commitment to a particular foundation. We get back to that in Rem. 12.

The model translation functor is given by composition with  $\sigma^{mod}$ :

**Definition 24 (Model Translation).** Given an  $\mathbb{M}$ -signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$  as in Def. 18, we define  $\mathbf{Mod}^{\mathbb{M}}(\sigma) : \mathbf{Mod}^{\mathbb{M}}(\Sigma') \rightarrow \mathbf{Mod}^{\mathbb{M}}(\Sigma)$  by

$$\mathbf{Mod}^{\mathbb{M}}(\sigma)(M, m) = (M, m \circ \sigma^{mod})$$

$$\mathbf{Mod}^{\mathbb{M}}(\sigma)(\varphi) = \varphi$$

The following commutative diagram illustrates the functor  $\mathbf{Mod}^{\mathbb{M}}$ :

$$\begin{array}{ccc} \Sigma^{mod} & \xrightarrow{\sigma^{mod}} & \Sigma'^{mod} \\ & \searrow m & \swarrow m' \\ M & \xrightarrow{\varphi} & M' \end{array}$$

**Example 13 (Continued).** We have an example model  $(HOL, m_1) \in \mathbf{Mod}^{\mathbb{M}}(PQ)$  by defining the LF-signature morphism  $m_1$  to map

- all symbols of  $HOL$  to themselves,
- *worlds* to the set  $\mathbb{N}$ ,
- *acc* to the  $\leq$  relation on  $\mathbb{N}$ ,
- $p$  and  $q$  to the predicates *odd* and *nonzero* on  $\mathbb{N}$  that are true for odd and non-zero numbers, respectively.

The model theoretical truth, i.e., the satisfaction relation, is given by the inhabitation of the type  $\text{base}(\text{ded}) F$  in a model:

**Definition 25 (Satisfaction).** For a  $\Sigma$ -model  $(M, m)$  and a  $\Sigma$ -sentence  $F$ , we define

$$(M, m) \models_{\Sigma}^{\mathbb{M}} F \quad \text{iff} \quad \text{there exists } t \text{ such that } \cdot \vdash_M t : m(\text{base}(\text{ded}) F).$$

**Example 14 (Continued).** We can assume that  $HOL$  has expressions  $true : tm \text{ bool}$  and  $false : tm \text{ bool}$  such that  $\text{ded } true$  is inhabited but  $\text{ded } false$  is empty. Thus, we obtain for the satisfaction in the model  $(HOL, m_1)$  from above

$$(HOL, m_1) \models_{PQ}^{\mathbb{M}} F \quad \text{iff} \quad \text{exists } t \text{ such that } \cdot \vdash_{HOL} t : \text{ded } (\forall [w : tm \mathbb{N}] m_1(F) w).$$

That holds iff  $m_1(F)$  is the constant function returning *true* (up to provable equality in  $HOL$ ). For example, if we assume that  $HOL$  contains proof rules to derive common HOL-theorems, we obtain  $(HOL, m_1) \models_{PQ}^{\mathbb{M}} p \supset \square q$  because we can simplify  $m_1(p \supset \square q) w$  to the formula

$$\text{odd } w \Rightarrow \forall [w' : tm \mathbb{N}] ((w \leq w') \Rightarrow \text{nonzero } w')$$

which is true for all  $w$ .

**Remark 11 (Lax Model Categories).** Def. 23 can be generalized by using lax slice categories under  $\Sigma^{mod}$ . Then model morphisms from  $(M, m)$  to  $(M', m')$  are pairs  $(\varphi, r)$  for an LF-signature morphism  $\varphi : M \rightarrow M'$  and a 2-cell  $r : \varphi \circ m \Rightarrow m'$ . Model reduction can be generalized accordingly. This is problematic, however, because there is no natural

way to turn the category of LF signatures into a 2-category. The same remark applies when other type theories are used instead of LF.

Recently we developed a theory of logical relations for LF as relations between LF signature morphisms Sojakova (2010). Binary logical relations do not form a 2-category either but behave like 2-cells in many ways. We expect that our present definition can be improved along those lines but omit the details here.

**Remark 12 (Restricted Model Categories).** This definition of models as morphisms  $(M, m)$  is very general permitting, e.g., the trivial model  $(\Sigma^{mod}, id_{\Sigma^{mod}})$ . It is often desirable to restrict attention to certain models or certain model morphisms.

Most importantly, we are usually interested in those models where the codomain is a fixed LF-signature  $\mathcal{F}$  representing the foundation of mathematics such as  $\mathcal{F} = HOL$  above. Then the LF signature morphism  $m \circ \mu$  maps the syntax into the foundation  $\mathcal{F}$ , i.e., every syntactical expression to its interpretation in the language of mathematics. We will get back to this in Sect. 4.5.

More generally, we can supplement our definition of signatures with a formal language in which structural properties of models and morphisms can be formulated. For example,  $\Sigma^{mod}$  might contain a declaration that requires  $m$  to be an elementary embedding. The key unsolved difficulty here is to design a formal language that is reasonably expressive without making the framework overly complex.

#### 4.4. Collecting the Pieces

We conclude the definition of  $\mathbb{M}$  with the following result.

**Theorem 5.**  $\mathbb{M} = (\mathbf{Sig}^{\mathbb{M}}, \mathbf{Sen}^{\mathbb{M}}, \mathbf{Mod}^{\mathbb{M}}, \models^{\mathbb{M}}, \mathbf{Pf}^{\mathbb{M}}, \vdash^{\mathbb{M}})$  is a logic.

*Proof.* We have to prove a number of conditions:

- $\mathbf{Sig}$  is a category. The only non-obvious aspect here is that the existence of  $F'$  in the definition of signature morphisms is preserved under composition. It is easy to see.
- $\mathbf{Sen}^{\mathbb{M}}$  is a functor. This is straightforward.
- $\mathbf{Mod}^{\mathbb{M}}$  is a functor. This is a standard result of category theory for slice categories (see, e.g., Mac Lane (1998)).
- It is a standard about the category of contexts in type theories that  $\mathbf{Pf}^{\mathbb{M}}$  is a functor  $\mathbf{Sig}^{\mathbb{M}} \rightarrow \mathcal{CAT}$  (see, e.g., Pitts (2000)). Thus, we only need to show that  $\mathbf{Pf}^{\mathbb{M}}(\Sigma)$  has finite products and that  $\mathbf{Pf}^{\mathbb{M}}(\sigma)$  preserves these products. The products are given by  $\Gamma_1 \times \dots \times \Gamma_n = \Gamma'_1, \dots, \Gamma'_n$  where  $\Gamma'_i$  arises from  $\Gamma_i$  by  $\alpha$ -renaming to eliminate duplicate variable names. The empty product (terminal object) is the empty context.
- $\models^{\mathbb{M}}$  meets the satisfaction condition. Assume  $\sigma : \Sigma \rightarrow \Sigma'$ ,  $F \in \mathbf{Sen}^{\mathbb{M}}(\Sigma)$ , and  $(M, m) \in \mathbf{Mod}^{\mathbb{M}}(\Sigma')$ . Then

$$\begin{aligned} \mathbf{Mod}^{\mathbb{M}}(\sigma)(M, m) \models_{\Sigma}^{\mathbb{M}} F &\text{ iff} \\ \text{there is an } M\text{-term of type } (m \circ \sigma^{mod})(\mu(\text{base}(\text{ded } F)) &\text{ iff} \\ \text{there is an } M\text{-term of type } m(\mu'(\sigma^{syn}(\text{base}(\text{ded } F))) &\text{ iff} \\ \text{there is an } M\text{-term of type } m(\mu'(\text{base}'(\text{ded } \mathbf{Sen}^{\mathbb{M}}(\sigma)(F)))) &\text{ iff} \\ (M, m) \models_{\Sigma'}^{\mathbb{M}} \mathbf{Sen}^{\mathbb{M}}(\sigma)(F). & \end{aligned}$$

$\dashv\vdash^{\mathbb{M}}$  is preserved. Assume  $\sigma : \Sigma \rightarrow \Sigma'$  and  $F \in \mathbf{Sen}^{\mathbb{M}}(\Sigma)$ . Then

$$\begin{aligned}\mathbf{Pf}^{\mathbb{M}}(\sigma)(\vdash_{\Sigma}^{\mathbb{M}} F) &= x : \sigma^{pf}(\pi(\text{base}(\text{ded}) F)) = \\ x : \pi'(\sigma^{syn}(\text{base}(\text{ded}) F)) &= x : \pi'(\text{base}'(\text{ded}) \mathbf{Sen}^{\mathbb{M}}(\sigma)(F)) = \\ \vdash_{\Sigma'}^{\mathbb{M}} \mathbf{Sen}^{\mathbb{M}}(\sigma)(F)\end{aligned}$$

□

Soundness and Completeness  $\mathbb{M}$  is intentionally neither sound nor complete because it is designed to be a meta-logic in which other possibly unsound or incomplete logics are represented. However, we have the following soundness criterion:

**Definition 26.** An  $\mathbb{M}$ -signature  $(\Sigma^{syn}, \Sigma^{pf}, \Sigma^{mod}, \text{base}, \pi, \mu)$  is called **sound** if there is an LF signature morphism  $\psi : \Sigma^{pf} \rightarrow \Sigma^{mod}$  such that the following diagram commutes:

$$\begin{array}{ccc}\Sigma^{syn} & \xrightarrow{\pi} & \Sigma^{pf} \\ & \searrow \mu & \downarrow \psi \\ & & \Sigma^{mod}\end{array}$$

This definition is justified by:

**Theorem 6.** The logic arising from  $\mathbb{M}$  by considering only the sound  $\mathbb{M}$ -signatures is strongly sound.

*Proof.* Let  $\Sigma$  be a sound signature (i) as in Def. 26,  $\Theta$  a set of  $\Sigma$ -sentences, and  $F$  a  $\Sigma$ -sentence. Assume  $\Theta \vdash_{\Sigma}^{\mathbb{M}} F$  (ii); then we need to show  $\Theta \models_{\Sigma}^{\mathbb{M}} F$ . So assume an arbitrary model  $(M, m) \in \mathbf{Mod}^{\mathbb{M}}(\Sigma)$  such that  $(M, m) \models_{\Sigma}^{\mathbb{M}} A$  for every  $A \in \Theta$  (iii); then we need to show  $(M, m) \models_{\Sigma}^{\mathbb{M}} F$ .

By (ii), there must be a substitution from  $\vdash_{\Sigma}^{\mathbb{M}} F$  to  $\vdash_{\Sigma}^{\mathbb{M}} F_1 \times \dots \times \vdash_{\Sigma}^{\mathbb{M}} F_n$  for some  $\{F_1, \dots, F_n\} \subseteq \Theta$ . Using the LF type theory, it is easy to show that this is equivalent to having a closed  $\Sigma^{pf}$ -term  $t$  of type

$$\pi(\text{base}(\text{ded}) F_1 \rightarrow \dots \rightarrow \text{base}(\text{ded}) F_n \rightarrow \text{base}(\text{ded}) F).$$

By (i), we obtain a closed  $M$ -term  $m(\psi(t))$  of type

$$m(\mu(\text{base}(\text{ded}) F_1 \rightarrow \dots \rightarrow \text{base}(\text{ded}) F_n \rightarrow \text{base}(\text{ded}) F)).$$

Now by (iii), there are  $M$ -terms  $t_i$  of type  $m(\mu(\text{base}(\text{ded}) F_i))$ . Thus, there is also an  $M$ -term  $m(\psi(t)) t_1 \dots t_n$  of type  $m(\mu(\text{base}(\text{ded}) F))$  and therefore  $(M, m) \models_{\Sigma}^{\mathbb{M}} F$ . □

We will refine this into a criterion to establish the soundness of a logic in Thm. 10 below.

**Remark 13 (Completeness).** It does not make sense to use the analogue of Def. 26 for completeness:  $\Sigma^{mod}$  is typically much stronger than  $\Sigma^{pf}$  – often as strong as mathematics

as a whole as in Sect. 4.5 – so that there can be no signature morphism  $\Sigma^{mod} \rightarrow \Sigma^{syn}$ . Moreover, even if we had such a morphism, it would not yield completeness because the proof of Thm. 6 crucially uses the composition  $m \circ \psi$ .

This is not surprising. A soundness result is naturally proved by showing inductively that the interpretation function maps every derivable syntactic statement to a true semantic statement; this is exactly what a signature morphism does. A completeness result on the other hand is usually proved by exhibiting a canonical model, which has a very different flavor.

However, it is still very promising to undertake completeness proofs within our framework: The canonical model is usually formed from the syntax, which is already formally represented in  $\Sigma^{syn}$ . A framework like LF is ideal to build canonical models as abstract data types. However, to show completeness, we still have to reflect these LF-level syntactical models into  $\Sigma^{mod}$ , e.g., by interpreting an LF type as the set of LF terms of that type. This technique is related to the definition of abstract data types in Isabelle/HOL (Nipkow et al. (2002)) and the quotations of Chiron (Farmer (2010)). We have to leave a further investigation to future work.

**Theories** Due to the Curry-Howard representation of proofs, there is no significant conceptual difference between signatures and theories from an LF perspective. In fact,  $\mathbb{M}$ -signatures subsume the finite theories already:

**Notation 4.** If  $\Theta = \{F_1, \dots, F_n\}$ , the  $\mathbb{M}$ -theory  $(\Sigma, \Theta)$  behaves in the same way as the  $\mathbb{M}$ -signature arising from  $\Sigma$  by appending  $a_1 : base(\text{ded}) F_1, \dots, a_n : base(\text{ded}) F_n$  to  $\Sigma^{syn}$ . We write  $(\Sigma, \Theta)^{syn}$  for this signature. Accordingly, we obtain (by pushout along  $\Sigma^{syn} \rightarrow (\Sigma, \Theta)^{syn}$ ) the signatures  $(\Sigma, \Theta)^{pf}$  and  $(\Sigma, \Theta)^{mod}$ .

Correspondingly, we can express  $\mathbb{M}$ -theory morphisms as LF-signature morphisms:

**Theorem 7.** Assume an  $\mathbb{M}$ -signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$  and finite theories  $(\Sigma, \Theta)$  and  $(\Sigma', \Theta')$ .

1. The following properties of  $\sigma$  are equivalent:
  - $\sigma$  is a proof theoretical theory morphism  $(\Sigma, \Theta) \rightarrow (\Sigma', \Theta')$ .
  - There is an LF-signature morphism  $(\sigma, \vartheta)^{pf}$  such that the left diagram below commutes.
2. The following properties of  $\sigma$  are equivalent:
  - $\sigma$  is a model theoretical theory morphism  $(\Sigma, \Theta) \rightarrow (\Sigma', \Theta')$ .
  - There is an LF-signature morphism  $(\sigma, \vartheta)^{mod}$  such that the right diagram below commutes.

$$\begin{array}{ccc}
 \Sigma^{pf} & \xrightarrow{\sigma^{pf}} & \Sigma^{pf} \\
 \downarrow & & \downarrow \\
 (\Sigma, \Theta)^{pf} & \xrightarrow{(\sigma, \vartheta)^{pf}} & (\Sigma', \Theta')^{pf}
 \end{array}
 \quad
 \begin{array}{ccc}
 \Sigma^{mod} & \xrightarrow{\sigma^{mod}} & \Sigma^{mod} \\
 \downarrow & & \downarrow \\
 (\Sigma, \Theta)^{mod} & \xrightarrow{(\sigma, \vartheta)^{mod}} & (\Sigma', \Theta')^{mod}
 \end{array}$$

*Proof.*

1.  $\sigma$  being a proof theoretical theory morphism is equivalent to there being proof terms  $p_i : \sigma^{syn}(\pi(base'(\text{ded}) F_i))$  over the LF-signature  $(\Sigma', \Theta')^{pf}$  for every  $F_i \in \Theta$ , i.e., for every  $a_i : base(\text{ded}) F_i$  in  $(\Sigma, \Theta)^{syn}$ . Then mapping every  $a_i$  to  $p_i$  yields the LF-signature morphism  $(\sigma, \vartheta)^{pf} : (\Sigma, \Theta)^{pf} \rightarrow (\Sigma', \Theta')^{pf}$ . The inverse direction is proved accordingly.
2. First, we observe, using the definition of satisfaction, that a  $\Sigma$ -model  $(M, m)$  satisfies all axioms in  $\Theta$  iff  $m : \Sigma^{mod} \rightarrow M$  can be factored through  $(\Sigma, \Theta)^{mod}$  (in other words:  $m$  can be extended to a signature morphism  $(\Sigma, \Theta)^{mod}$ ). Thus, given  $(\sigma, \vartheta)^{mod}$ , models of  $\Sigma'$  yield models of  $\Sigma$  simply by composition, which proves  $\sigma$  is a model theoretical theory morphism. Conversely, one obtains  $(\sigma, \vartheta)^{mod}$  by factoring the  $\Sigma$ -model  $((\Sigma', \Theta')^{mod}, \sigma^{mod})$  through  $(\Sigma, \Theta)^{mod}$ .

□

The criterion for proof theoretical theory morphisms is quite intuitive and useful: Via the Curry-Howard representation, axioms are mapped to proof terms. The model theoretical criterion is much less useful because there are many models in  $\mathbb{M}$ , which makes it a very strong condition. It guarantees that  $\sigma$  is a model theoretical theory morphism for any foundation. However, often the property of being a model theoretical theory morphism depends on the chosen foundation. In Sect. 4.5, we will look at a weaker, foundation-specific condition.

#### 4.5. Foundations

We call  $\mathbb{M}$  **foundation-independent** because it is not committed to a fixed foundation, in which the model theory of object logics is defined. Thus, we are able to express logics using different mathematical foundations and even translations between them. However, this is also a drawback because for any specific logic, we are usually only interested in certain models  $(M, m)$ , namely in those where  $M$  is the semantic realm of all mathematical objects. Therefore, we restrict  $\mathbb{M}$  accordingly using a fixed LF-signature  $\mathcal{F}$  that represents the foundation of mathematics.

The intuition is as follows.  $\mathcal{F}$  is a meta-language, which is used by  $\Sigma^{mod}$  to specify models. Thus,  $\Sigma^{mod}$  includes  $\mathcal{F}$  and adds symbols to it that represent the free parameters of a model, e.g., the universe and the function and predicate symbols for first-order models. Models must interpret those added symbols as expressions of the foundation in a way that preserves typing (and thus via Curry-Howard, in a way that satisfies the axioms). Therefore, we can represent models as signature morphisms  $m : \Sigma^{mod} \rightarrow \mathcal{F}$ .

However, representing a foundation in LF can be cumbersome due to the size of the theory involved. Our representation of ZFC that we use in Horozal and Rabe (2011) requires several thousand declarations just to build up the basic notions of ZFC set theory to a degree that supports simple examples. Moreover, the automation of foundations is a very difficult problem, and type checking or automated reasoning for untyped set theories is much harder than for most logics.

We remedy these drawbacks with a slightly more general approach. We use two LF-signatures  $\mathcal{F}_0$  and  $\mathcal{F}$  with a morphism  $P : \mathcal{F}_0 \rightarrow \mathcal{F}$ . The idea is that  $\mathcal{F}_0$  is a fragment or an approximation of the foundation  $\mathcal{F}$  that is refined into  $\mathcal{F}$  via the morphism  $P$ . It turns out that manageably simple choices of  $\mathcal{F}_0$  are expressive enough to represent the model theory of most logics.

This has the additional benefit that the same  $\mathcal{F}_0$  can be used together with different values for  $P$  and  $\mathcal{F}$ . This corresponds to the mathematical practice of not detailing the foundation unless necessary. For example, standard constructions and results like the real numbers are usually assumed without giving or relying on a specific definition. For example, in Horozal and Rabe (2011), we use a simply-typed higher-order logic for  $\mathcal{F}_0$ , ZFC set theory as  $\mathcal{F}$ , and  $P$  interprets types as sets and terms as elements of these sets.  $\mathcal{F}_0$  can be written down in LF within a few minutes, and strong implementations are available (Gordon (1988); Harrison (1996); Nipkow et al. (2002)).

If we combine this approach with a module system for LF – as we do for example in Codescu et al. (2011) – this gives rise to the paradigm of **little foundations**. It corresponds to the little theories used implicitly in Bourbaki (1974) and explicitly in Farmer et al. (1992). Using little foundations, we specify models in  $\mathcal{F}_0$  making as few foundational assumptions as possible and refine it to a stronger foundation when needed. In fact,  $\Sigma^{syn}$  can be seen as the extreme case of making no assumptions about the foundation, and  $\mu : \Sigma^{syn} \rightarrow \Sigma^{mod}$  as a first refinement step. At the opposite end of the refinement chain, we might find the extension  $ZF \hookrightarrow ZFC$ , which adds the axiom of choice.

We formalize these intuitions as follows:

**Definition 27 (Foundations).** Given an LF-signature  $\mathcal{F}_0$ ,  $\mathbf{Sig}_{\mathcal{F}_0}^{\mathbb{M}}$  is the subcategory of  $\mathbf{Sig}^{\mathbb{M}}$  that is restricted to

- those signatures  $\Sigma$  for which there is an inclusion from  $\mathcal{F}_0$  into  $\Sigma^{mod}$ ,
- those signature morphisms  $\sigma$  for which  $\sigma^{mod}$  is the identity on  $\mathcal{F}_0$ .

Thus,  $\mathbf{Sig}_{\mathcal{F}_0}^{\mathbb{M}}$  is the full subcategory of inclusions of the slice category of objects under  $\mathcal{F}_0$ . Note that we recover  $\mathbf{Sig}^{\mathbb{M}}$  when  $\mathcal{F}_0$  is the empty signature.

**Definition 28 (Founded Models).** For an LF-signature morphism  $P : \mathcal{F}_0 \rightarrow \mathcal{F}$ , we define  $\mathbf{Mod}_P^{\mathbb{M}}$  as the functor  $\mathbf{Sig}_{\mathcal{F}_0}^{\mathbb{M}} \rightarrow \mathcal{CAT}^{op}$  that maps

- signatures  $\Sigma$  to the subcategory of  $\mathbf{Mod}^{\mathbb{M}}(\Sigma)$  restricted to the models  $(\mathcal{F}, m)$  for which  $m$  agrees with  $P$  on  $\mathcal{F}_0$ ,
- signature morphisms in the same way as  $\mathbf{Mod}^{\mathbb{M}}$ .

These definitions lead to the following commutative diagram for a  $\Sigma'$ -model  $(\mathcal{F}, m)$  translated along  $\sigma$ :

$$\begin{array}{ccc} \Sigma^{mod} & \xrightarrow{\sigma^{mod}} & \Sigma'^{mod} \\ & \searrow \mathcal{F}_0 \curvearrowleft & \swarrow \\ m \circ \sigma^{mod} & P \downarrow & m \\ & \searrow & \swarrow \\ & \mathcal{F} & \end{array}$$

And these specializations indeed yield a logic:

**Theorem 8.** For arbitrary  $P : \mathcal{F}_0 \rightarrow \mathcal{F}$ , we obtain a logic

$$\mathbb{M}_P = (\mathbf{Sig}_{\mathcal{F}_0}^{\mathbb{M}}, \mathbf{Sen}^{\mathbb{M}}, \mathbf{Mod}_P^{\mathbb{M}}, \models^{\mathbb{M}}, \mathbf{Pf}^{\mathbb{M}}, \vdash^{\mathbb{M}})$$

(where  $\mathbf{Sen}^{\mathbb{M}}$ ,  $\models^{\mathbb{M}}$ ,  $\mathbf{Pf}^{\mathbb{M}}$ , and  $\vdash^{\mathbb{M}}$  are the appropriate restrictions according to  $\mathbf{Sig}_{\mathcal{F}_0}^{\mathbb{M}}$  and  $\mathbf{Mod}_P^{\mathbb{M}}$ ).

*Proof.* This follows immediately from Thm. 5. The only non-trivial aspect is to show that  $\mathbf{Mod}_P^{\mathbb{M}}(\sigma)$  is well-defined, and that follows from the commutativity of the above diagram.  $\square$

**Example 15 (Continued).** In Ex. 12, we have already used  $\mathbb{M}_P$ -models: We used  $\mathcal{F}_0 = \mathcal{F} = HOL$ , and  $P$  was the identity.

Now we can revisit Thm. 7, which stated that model theoretical theory morphisms  $\sigma : (\Sigma, \Theta) \rightarrow (\Sigma', \Theta')$  in  $\mathbb{M}$  are equivalent to LF-signature morphisms  $(\sigma, \vartheta)^{mod} : (\Sigma, \Theta)^{mod} \rightarrow (\Sigma', \Theta')^{mod}$ . If we restrict attention to  $\mathbb{M}_P$ , the situation is a bit more complicated:

**Theorem 9.** Assume an  $\mathbb{M}_P$ -signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$  and finite theories  $(\Sigma, \Theta)$  and  $(\Sigma', \Theta')$ . The following properties of  $\sigma$  are equivalent:

- $\sigma$  is a model theoretical theory morphism  $(\Sigma, \Theta) \rightarrow (\Sigma', \Theta')$  in the logic  $\mathbb{M}_P$ .
- For every LF-signature morphism  $m' : (\Sigma', \Theta')^{mod} \rightarrow \mathcal{F}$ , there is an LF-signature morphism  $m : (\Sigma, \Theta)^{mod} \rightarrow \mathcal{F}$  such that the following diagram commutes:

$$\begin{array}{ccc} \Sigma^{mod} & \xrightarrow{\sigma^{mod}} & \Sigma'^{mod} \\ \downarrow & & \downarrow \\ (\Sigma, \Theta)^{mod} & & (\Sigma', \Theta')^{mod} \\ m \swarrow & & \searrow m' \\ \mathcal{F} & & \end{array}$$

*Proof.* This follows immediately because  $\sigma$  is a model theoretical theory morphism iff  $\mathbf{Mod}_P^{\mathbb{M}}(\sigma)$  maps  $\Theta'$ -models to  $\Theta$ -models.  $\square$

Using Thm. 9, extending  $\sigma^{mod}$  to an LF-signature morphism  $(\sigma, \vartheta)^{mod} : (\Sigma, \Theta)^{mod} \rightarrow (\Sigma', \Theta')^{mod}$  is sufficient to make  $\sigma$  a model theoretical  $\mathbb{M}_P$ -theory morphism. But it is not a necessary condition. Counter-examples arise when  $\mathcal{F}_0$  is too weak and cannot prove all theorems of  $\mathcal{F}$ . In Horozal and Rabe (2011), we show that is a necessary condition in the special case  $\mathcal{F}_0 = \mathcal{F} = ZFC$ . The general case is open.

## 5. Defining and Encoding Logics

We can combine the results of Sect. 3 and 4 to define logics and logic translations in our framework. Logics are defined or represented using functors  $\Phi : \mathbf{Sig} \rightarrow \mathbf{Sig}^{\mathbb{M}}$ . But the formal details can be cumbersome in practice:  $\mathbb{M}$ -signatures are 6-tuples and morphisms 3-tuples with some commutativity constraints. Therefore, we introduce the notion of uniform logic representations, which are the typical situation in practice and which can be given conveniently.

Uniform logics are generated by a fixed  $\mathbf{Sig}^{\mathbb{M}}$  signature  $L$ . The intuition is that  $L^{syn}$  represents the logical symbols, and extensions  $L^{syn} \hookrightarrow \Sigma^{syn}$  add one declaration for each non-logical symbols. This is very similar to the use of “uniform” in Harper et al. (1994), from which we borrow the name. Then proof and model theory are given in general by  $L^{pf}$  and  $L^{mod}$ , and these induce extensions  $\Sigma^{pf}$  and  $\Sigma^{mod}$  for the proof and model theory of a specific choice of non-logical symbols. Similarly, translations interpret logical symbols in terms of logical symbols and non-logical ones in terms of non-logical ones.

### 5.1. Non-Logical Symbols

Recall from Sect. 2.2 that we write  $\mathbb{LF}$  for the category of LF signatures and LF signature morphisms. Because  $\mathbf{Sig}^{\mathbb{M}}$  morphisms are triples of  $\mathbb{LF}$  morphisms and composition is defined component-wise,  $\mathbf{Sig}^{\mathbb{M}}$  inherits inclusions and pushouts from  $\mathbb{LF}$ . In particular:

**Lemma 2.**  $\mathbf{Sig}^{\mathbb{M}}$  has inclusion morphisms in the following sense: The subcategory of  $\mathbf{Sig}^{\mathbb{M}}$  which contains

- all objects and
- only those morphisms  $(\sigma^{syn}, \sigma^{pf}, \sigma^{mod}) : \Sigma \rightarrow \Sigma'$  for which  $\sigma^{syn}$ ,  $\sigma^{pf}$ , and  $\sigma^{mod}$  are inclusions in  $\mathbb{LF}$

is a partial order. We write  $\Sigma \hookrightarrow \Sigma'$  for inclusion morphisms from  $\Sigma$  to  $\Sigma'$

*Proof.* Morphisms in  $\mathbf{Sig}^{\mathbb{M}}$  are triples, and composition is defined component-wise. Therefore, the properties of inclusions follow immediately from those of inclusions in  $\mathbb{LF}$ .  $\square$

**Remark 14.** More generally, the inclusions of  $\mathbb{LF}$  induce a weak inclusion system in the sense of Căzănescu and Roșu (1997). This property is inherited by  $\mathbf{Sig}^{\mathbb{M}}$  as well.

We will use inclusions  $L \hookrightarrow \Sigma$  to represent a logic  $L$  together with non-logical symbols given by  $\Sigma$ . When giving the non-logical symbols, it is not necessary to give  $\Sigma$ ; instead, we can just give an  $\mathbb{LF}$  inclusion  $L^{syn} \hookrightarrow \Sigma^{syn}$  and obtain the remaining components of  $\Sigma$  by pushout constructions. We make that precise in the following definitions:

**Definition 29.** An inclusion  $L^{syn} \hookrightarrow \Sigma^{syn}$  is called *compatible* with  $L$  if  $\Sigma^{syn}$  does not add declarations for names that are also declared in  $L^{pf}$  or  $L^{mod}$ .

**Definition 30.** We define the category  $\mathbb{M} + \mathbb{LF}$  as follows

- objects are the pairs  $(L, \Sigma^{syn})$  of an  $\mathbb{M}$  signature  $L$  and an  $\mathbb{LF}$  inclusion  $L^{syn} \hookrightarrow \Sigma^{syn}$  that is compatible with  $L$ ,
- morphisms from  $(L, \Sigma^{syn})$  to  $(L', \Sigma'^{syn})$  are the commuting rectangles

$$\begin{array}{ccc} L^{syn} & \longleftrightarrow & \Sigma^{syn} \\ \uparrow l^{syn} & & \uparrow \sigma^{syn} \\ L^{syn} & \longleftrightarrow & \Sigma^{syn} \end{array}$$

The restriction to compatible inclusions in Def. 30 is a technicality needed for Def. 31: If  $\Sigma^{syn}$  added a name that is also declared in  $L^{pf}$  or  $L^{mod}$ , the respective canonical pushout in  $\mathbb{LF}$  would not exist. This restriction is harmless in practice because namespaces can be used to ensure the uniqueness of names.

For compatible inclusions, the canonical pushouts exist and every  $\mathbb{M} + \mathbb{LF}$  object  $(L, \Sigma^{syn})$  induces an  $\mathbb{M}$  signature and every  $\mathbb{M} + \mathbb{LF}$  morphism  $(l, \sigma^{syn})$  induces an  $\mathbb{M}$  signature morphism:

**Definition 31.** We define a functor  $\overline{\cdot} : \mathbb{M} + \mathbb{LF} \rightarrow \mathbf{Sig}^{\mathbb{M}}$  as follows

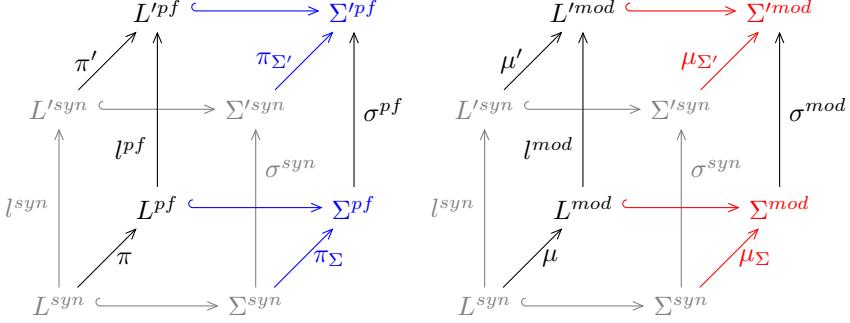
- For objects  $(L, \Sigma^{syn})$ , we define the  $\mathbb{M}$  signature

$$\overline{(L, \Sigma^{syn})} = (\Sigma^{syn}, \Sigma^{pf}, \Sigma^{mod}, base_{\Sigma}, \pi_{\Sigma}, \mu_{\Sigma})$$

such that the following diagram commutes and  $\Sigma^{pf}$  and  $\Sigma^{mod}$  are canonical pushouts in  $\mathbb{LF}$

$$\begin{array}{ccccc} & & L^{pf} & \xhookrightarrow{\quad} & \Sigma^{pf} \\ & & \nearrow \pi & & \nearrow \pi_{\Sigma} \\ Base & \xrightarrow{\quad base \quad} & L^{syn} & \xhookrightarrow{\quad} & \Sigma^{syn} \\ & & \searrow \mu & & \searrow \mu_{\Sigma} \\ & & & & L^{mod} \xhookrightarrow{\quad} \Sigma^{mod} \end{array}$$

- For morphisms  $(l, \sigma^{syn}) : (L, \Sigma^{syn}) \rightarrow (L', \Sigma'^{syn})$ , we define the  $\mathbb{M}$  signature morphism  $\overline{(l, \sigma^{syn})} = (\sigma^{syn}, \sigma^{pf}, \sigma^{mod}) : \overline{(L, \Sigma^{syn})} \rightarrow \overline{(L', \Sigma'^{syn})}$  such that  $\sigma^{pf}$  and  $\sigma^{mod}$  are the unique morphisms that make the following  $\mathbb{LF}$  diagrams commute



### 5.2. Uniform Logics

We obtain a uniform logic  $\mathbb{M}_P^L$  for every  $\mathbb{M}$  signature  $L$  by pairing  $L$  with all possible choices of non-logical symbols. In  $\mathbb{M}_P^L$ , we also fix a foundation for the model theory:

**Definition 32 (Uniform Logics).** Assume an LF signature morphism  $P : \mathcal{F}_0 \rightarrow \mathcal{F}$  and a  $\mathbf{Sig}_{\mathcal{F}_0}^{\mathbb{M}}$ -signature  $L$ . We write  $L + \mathbb{LF}$  for the subcategory of  $\mathbb{M} + \mathbb{LF}$  containing the objects  $(L, \Sigma^{syn})$  and the morphisms  $(id_L, \sigma^{syn})$ . Then we obtain a logic

$$\mathbb{M}_P^L = \mathbb{M}_P \circ \overline{\cdot}|_{L + \mathbb{LF}}.$$

Here  $\overline{\cdot}|_{L + \mathbb{LF}} : L + \mathbb{LF} \rightarrow \mathbf{Sig}^{\mathbb{M}}$  is the restriction of  $\overline{\cdot}$  to  $L + \mathbb{LF}$ .

Uniform logics are defined completely within LF – signatures, sentences, proofs, and models are given as syntactical entities of LF. We already know how to represent existing logics in LF, namely via Thm. 12. Uniform logic encodings arise when we use  $\mathbb{M}_P^L$  as the meta-logic.

**Example 16 (Modal Logic).** If we take all examples from Sect. 4 together, we obtain a uniform encoding of  $\mathbb{ML}$  using the  $\mathbb{M}$ -signature

$$ML = (ML^{syn}, ML^{pf}, ML^{mod}, incl, incl, \mu_{ML})$$

(where  $incl$  denotes inclusion morphisms). We define a functor  $\Phi^{\mathbb{ML}} : \mathbf{Sig}^{\mathbb{ML}} \rightarrow ML + \mathbb{LF}$  as follows. For a modal logic signature  $\Sigma = \{p_1, \dots, p_n\} \in |\mathbf{Sig}^{\mathbb{ML}}|$  and a signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$ , we define  $\Phi^{\mathbb{ML}}(\Sigma) = (ML, \Sigma^{syn})$  and  $\Phi^{\mathbb{ML}}(\sigma) = (id, \sigma^{syn})$  with

$$\begin{aligned} \Sigma^{syn} &= ML^{syn}, p_1 : \mathbf{form}, \dots, p_n : \mathbf{form} \\ \sigma^{syn} &= id_{ML^{syn}}, p_1 := \sigma(p_1), \dots, p_n := \sigma(p_n) \end{aligned}$$

This yields  $\overline{\Phi^{\mathbb{ML}}(\Sigma)} = (\Sigma^{syn}, \Sigma^{pf}, \Sigma^{mod}, incl, incl, \mu_{\Sigma})$  and  $\overline{\Phi^{\mathbb{ML}}(\sigma)} = (\sigma^{syn}, \sigma^{pf}, \sigma^{mod})$  with

$$\begin{aligned} \Sigma^{pf} &= ML^{pf}, p_1 : \mathbf{form}, \dots, p_n : \mathbf{form} \\ \Sigma^{mod} &= ML^{mod}, p_1 : tm\ worlds \rightarrow tm\ bool, \dots, p_n : tm\ worlds \rightarrow tm\ bool \\ \mu_{\Sigma} &= \mu_{ML}, p_1 := p_1, \dots, p_n := p_n \\ \sigma^{pf} &= id_{ML^{pf}}, p_1 := \sigma(p_1), \dots, p_n := \sigma(p_n) \\ \sigma^{mod} &= id_{ML^{mod}}, p_1 := \sigma(p_1), \dots, p_n := \sigma(p_n) \end{aligned}$$

In particular, we have

$$\begin{aligned}\overline{\Phi^{\text{ML}}(PQ)} &= (PQ^{syn}, PQ^{pf}, PQ^{mod}, incl, incl, \mu_{PQ}) \quad \text{and} \\ \overline{\Phi^{\text{ML}}(w)} &= (w^{syn}, w^{pf}, w^{mod})\end{aligned}$$

as defined throughout the running example.

Moreover it is straightforward to show that  $\Phi^{\text{ML}}$  can be extended to a logic comorphism  $(\Phi^{\text{ML}}, \alpha, \beta, \gamma)$  from  $\text{ML}$  to  $\text{ML}_P^{ML}$  where  $P$  is as in Ex. 15.  $\alpha_\Sigma$  is an obvious bijection.  $\beta_\Sigma$  maps every LF-signature morphism  $m : \Sigma^{mod} \rightarrow HOL$  to a Kripke-model of  $\Sigma$ . Finally,  $\gamma_\Sigma$  is a straightforward encoding of judgments as types and proofs as terms.

It is very easy to show the proof theoretical adequacy of  $\gamma$ . The discussion of the model theoretical adequacy of  $\beta$  is complicated as it depends on which sets are eligible as Kripke frames in ML-models (see Rem. 16). If  $HOL$  can express all sets that are eligible as Kripke frames, we obtain model theoretical adequacy.

**Remark 15 (Declaration Patterns).** Def. 32 uses all extensions of  $L^{syn}$  as the signatures of  $\text{ML}_P^L$ . However, often we want to work with a subcategory instead. For example, for modal logic, we would like to use only those extensions with declarations of the form  $p : \text{form}$ . This requires an extension of LF with what we call *declaration patterns* (similar to the *block* declarations already present in Twelf, Pfenning and Schürmann (1999)). Such an extension is currently designed in joint work with Fulya Horozal.

**Remark 16 (Model Theoretical Adequacy).** Ex. 16 raises the question when model theoretical adequacy holds. Let us call a mathematical object **definable** if it can be denoted by an expression of the foundational language. For example, since there can only be countably many definable objects, a set theory with uncountably many objects has undefinable objects. But even if we can prove the existence of undefinable objects, we can never actually give one. On the other hand, it is easy to see that a model can be represented as an LF signature morphism iff all its components are definable.

Therefore, the surjectivity of  $\beta_\Sigma$  is and thus model theoretical adequacy depend on the philosophical point of view we take.

The type theoretical school often rejects a commitment to set theory – recall that LF and thus the essence of  $\text{ML}$  can be developed using only formal languages and inference systems without appealing to a foundation of mathematics like set theory. Type theory does not object to model theory per se; but it would restrict attention to definable models. In fact, our models-as-signature-morphisms paradigm is an appealing way to define *model* from a type theoretical point of view. This is particularly elegant if we observe that the approach unifies the study of models of theories and that of implementations of specifications. In that case, of course all models are expressible as signature morphisms, and adequacy holds.

The set theoretical school does not only define individual models but also the class of all models. Moreover, they see the sets of LF signatures and signature morphisms as defined within set theory. Then adequacy means the existence of a certain between a class of models and a set of morphisms. But then a simple cardinality argument shows

that adequacy usually does not hold. In order to maintain Thm. 2, we have to study weaker notions of adequacy. We leave this to further work.

Besides their simplicity, uniform representations have the advantage, that the properties of soundness can be proved formally within the logical framework:

**Theorem 10.** A uniform logic  $\mathbb{M}_P^L$  is strongly sound if  $L$  is a sound  $\mathbb{M}$  signature.

*Proof.* Using Thm. 6, we only have to show that for every  $\Sigma^{syn}$  extending  $L^{syn}$  there is an  $\mathbb{LF}$ -morphism from  $\Sigma^{pf}$  to  $\Sigma^{mod}$  with a certain commutativity property. That follows immediately from the universal property of the pushout  $\Sigma^{pf}$  using the soundness of  $L$ .  $\square$

We sketch another example – a fragment of our encoding of first-order logic from Horozal and Rabe (2011) – in order to give an example of a logic translation below.

**Example 17 (First-Order Logic).** To define first-order logic  $\mathbb{FOL}$ , we give a uniform functor  $\Phi^{\mathbb{FOL}}$  over the  $\mathbb{M}$  signature  $FOL$ . The central declarations of  $FOL^{syn}$  are:

|           |   |                                   |             |
|-----------|---|-----------------------------------|-------------|
| $i$       | : | type                              | terms       |
| $o$       | : | type                              | formulas    |
| $\supset$ | : | $o \rightarrow o \rightarrow o$   | implication |
| $\forall$ | : | $(i \rightarrow o) \rightarrow o$ | universal   |
| $ded$     | : | $o \rightarrow type$              | truth       |

$FOL^{pf}$  adds natural deduction proof rules in a straightforward way. For simplicity, we assume that  $FOL^{mod}$  extends the same signature  $\mathcal{F}_0 = HOL$  as  $ML^{mod}$ . This extension includes a declaration  $univ : tp$  for the universe.  $base_{FOL}$  and  $\pi_{FOL}$  are inclusions, and  $\mu_{FOL}$  maps  $i$  to  $tm\ univ$ ,  $o$  to  $tm\ bool$ , and all formulas to their semantics in the usual way.

Assume a  $\mathbb{FOL}$  signature  $\Sigma = (\Sigma_f, \Sigma_p, arit)$  where  $\Sigma_f$  and  $\Sigma_p$  are the sets of function and predicate symbols with arity function  $arit : \Sigma_f \cup \Sigma_p \rightarrow \mathbb{N}$ . Then  $\Phi^{\mathbb{FOL}}(\Sigma)$  extends  $FOL^{syn}$  with declarations  $f : i \rightarrow \dots \rightarrow i \rightarrow i$  and  $p : i \rightarrow \dots \rightarrow i \rightarrow o$ , respectively, for each function or predicate symbol.

### 5.3. Uniform Logic Translations

Logic translations between uniform logics are already possible using the general methods of Thm. 4. We speak of uniform translations in the special case where a fixed  $\mathbb{M}$ -signature morphism is used to translate the logical symbols:

**Theorem 11 (Uniform Logic Translations).** For  $i = 1, 2$ , assume functors  $\Phi^i : \mathbf{Sig}^i \rightarrow L_i + \mathbb{LF}$ . Assume

- a functor  $\Phi : \mathbf{Sig}^1 \rightarrow \mathbf{Sig}^2$ ,
- a  $\mathbf{Sig}_{\mathcal{F}_0}^{\mathbb{M}}$ -morphism  $l = (l^{syn}, l^{pf}, l^{mod}) : L_1 \rightarrow L_2$ ,
- a natural transformation  $m : \Phi^1 \rightarrow \Phi^2 \circ \Phi$  (seen as functors  $\mathbf{Sig}^1 \rightarrow \mathbb{M} + \mathbb{LF}$ ) such that each  $m_\Sigma$  is of the form  $(l, \alpha_\Sigma)$ .

Then  $(\varphi, \mathbb{M}_P \circ \overline{\cdot} \circ m)$  is a logic comorphism from  $\mathbb{M}_P^{L_1} \circ \Phi^1$  to  $\mathbb{M}_P^{L_2} \circ \Phi^2$ .

*Proof.* Observe that  $\overline{\cdot} \circ m$  is a natural transformation  $\overline{\cdot} \circ \Phi^1 \rightarrow \overline{\cdot} \circ \Phi^2 \circ \Phi$  between functors  $\mathbf{Sig}^1 \rightarrow \mathbf{Sig}^{\mathbb{M}}$ , and also that  $\mathbb{M}_P^{L_i} \circ \Phi^i = \mathbb{M}_P \circ \overline{\cdot} \circ \Phi^i$ . Then the result follows immediately from Thm. 4.  $\square$

Finally we can give a uniform logic translation from modal logic to first-order logic:

**Example 18 (Translating ML to FOL).** Based on Ex. 16 and 17, we represent the well-known translation from modal logic to first-order logic that makes worlds explicit and relativizes quantifiers.

Firstly, the functor  $\Phi : \mathbf{Sig}^{\mathbb{ML}} \rightarrow \mathbf{Sig}^{\mathbb{FOL}}$  maps every modal logic signature  $\Sigma$  to the  $\mathbb{FOL}$  signature that contains a binary predicate symbol  $acc$  (for the accessibility relation) and one unary predicate symbol  $p$  for every propositional variable  $p \in \Sigma$ .

Secondly, we cannot give a  $\mathbb{M}$  signature morphism  $ML \rightarrow FOL$  because the translation requires the fixed predicate symbol  $acc$ , which is present in every  $\Phi(\Sigma)$  but not in  $FOL^{syn}$ . Therefore, we put  $L_1 = ML$  and  $L_2 = FOL' = \overline{(FOL, \Sigma_0)}$  where  $\Sigma_0$  is the LF signature  $FOL^{syn}$ ,  $acc : i \rightarrow i \rightarrow o$ . Then  $\Phi^{\mathbb{FOL}} \circ \Phi$  is indeed a functor  $\mathbf{Sig}^{\mathbb{ML}} \rightarrow FOL' + \mathbb{LF}$ .

Thirdly, we give a  $\mathbf{Sig}^{\mathbb{M}}$  morphism  $l : ML \rightarrow FOL'$  as follows.  $l^{syn}$  is given by

$$\begin{aligned} \mathbf{form} &:= i \rightarrow o \\ \supset &:= [f : i \rightarrow o] [g : i \rightarrow o] [x : i] (f x) \supset (g x) \\ \Box &:= [f : i \rightarrow o] [x : i] \forall[y : i] ((acc x y) \supset (f y)) \\ \mathbf{ded} &:= [f : i \rightarrow o] ded \forall[x : i] f x \end{aligned}$$

ML-sentences are mapped to FOL-formulas with a free variable (i.e., terms of type  $i \rightarrow o$ ). The intuition is that the free variable represents the world in which the ML-sentence is evaluated.  $\supset$  and  $\Box$  are translated in the expected way. Finally, the ML-truth judgment  $\mathbf{ded} F$  is mapped to the FOL-judgment  $ded \forall[x] l^{syn}(F) x$ ; note how  $\forall$  is used to bind the above-mentioned free variable and to obtain a sentence.

$l^{pf}$  extends  $l^{syn}$  with assignments that map every proof rule of modal logic to a proof in first-order logic. These are technical but straightforward.  $l^{mod}$  is the identity for all symbols of  $HOL$  and maps

$$\begin{aligned} \mathbf{worlds} &:= univ \\ acc &:= acc \end{aligned}$$

It is easy to establish the commutativity requirements on  $(l^{syn}, l^{pf}, l^{mod})$ .

Fourthly, we define the natural transformation  $m$ .  $m_\Sigma$  must be a  $\mathbb{M} + \mathbb{LF}$  morphism  $(l, \alpha_\Sigma)$  from  $\Phi^{\mathbb{FOL}}(\Phi(\Sigma))$  to  $\Phi^{\mathbb{ML}}(\Sigma)$ . For  $\Sigma = \{p_1, \dots, p_n\}$ , the domain of  $\alpha_\Sigma$  is

$$ML^{syn}, p_1 : \mathbf{form}, \dots, p_n : \mathbf{form},$$

and the codomain is

$$FOL^{syn}, acc : i \rightarrow i \rightarrow o, p_1 : i \rightarrow o, \dots, p_n : i \rightarrow o.$$

Thus, we put  $\alpha_\Sigma = l^{syn}$ ,  $p_1 := p_1, \dots, p_n := p_n$ .

**Remark 17.** In the typical case where  $\Phi^1$  and  $\Phi^2$  are injective, we can modify Def. 11 to use a functor  $\Phi' : L_1 + \mathbb{LF} \rightarrow L_2 + \mathbb{LF}$  such that  $\Phi^2 \circ \Phi = \Phi' \circ \Phi^1$ .  $\Phi'$  represents the same signature translation as  $\Phi$  but lives within the meta-logic  $\mathbb{M}$ .

Then in the case of Ex. 18,  $\Phi'$  and the natural transformation  $m$  are given simply by pushout along  $l$ . This yields a significantly simpler notion of uniform logic translation induced by  $l$  alone. However, not all signature translations can be obtained in this way. A counter-example is the translation from sorted to unsorted FOL translates single declarations in  $\Sigma$  to multiple declarations in  $\Phi(\Sigma)$ .

In general,  $\Phi'$  and  $m$  can be formalized using the declaration patterns from Rem. 15, but we leave the details to future work.

**Remark 18 (Non-simple Morphisms).** Note that  $l$  in Ex. 18 is not simple, which finally justifies the particular choice in Def. 18 explained in Rem. 9. The induced sentence translation  $\mathbf{Sen}^M(l)$  can be factored into two steps: First a compositional translation maps  $F$  of type `form` to  $l^{syn}(F)$  of type  $i \rightarrow o$ ; it is followed by a non-compositional step that is only applied once on toplevel, which maps  $l^{syn}(F)$  to  $\forall[x] l^{syn}(F) x$  of type  $o$ .

This is typical for non-trivial logic translations: If the semantics of  $\mathbb{I}_1$  is encoded using the syntax of  $\mathbb{I}_2$ , then  $\mathbb{I}_1$ -sentences are translated to some  $\mathbb{I}_2$ -expressions, which must be lifted to  $\mathbb{I}_2$ -sentences using a final toplevel step. Another example is the translation of many-valued propositional logic to first-order logic where formulas are translated compositionally to terms and a final toplevel step applies a unary predicate for designated truth values.

More generally, we can distinguish two kinds of sentence translation functions: the **within-logic** translations  $\mathbf{Sen}(\sigma) : \mathbf{Sen}(\Sigma) \rightarrow \mathbf{Sen}(\Sigma)$  translate along a signature morphism between two signatures of the same logic, and the **across-logic** translations  $\alpha_\Sigma : \mathbf{Sen}(\Sigma) \rightarrow \mathbf{Sen}'(\Phi(\Sigma))$  translate along a logic translation between two signatures of different logics. The corresponding observation applies to model and proof translations.

Within-logic translations are typically straightforward homomorphic mappings that can be represented as simple  $\mathbf{Sig}^M$ -morphisms. But across-logic translations may involve complex encoding steps that may not be easily expressible using homomorphic mappings. When using a Grothendieck institution as in Diaconescu (2002), within-logic and across-logic translations are unified, and the latter are those that employ a non-trivial institution comorphism.

But if we use a meta-logic like  $M$ , both within- and across-logic translations must be induced by  $\mathbf{Sig}^M$ -morphisms. Therefore, we need our more general definition of  $\mathbf{Sig}^M$ -morphisms that includes non-simple ones. This problem is a consequence of using a meta-logic in which a limited formal language is used to express translations. The problem does not arise when using institutions (as we do in Sect. 3) because both within-logic and across-logic translations are represented as mappings and functors, the most general notion of a translation operation.

## 6. Discussion

Our basic motivation was to provide a logical framework in which both model theoretical and proof theoretical logics as well as their combinations can be formulated and studied. But a logical framework – or any framework for that matter – can only serve as an auxiliary device: it helps get work done but does not do the work by itself. In fact, a

good logical framework should be general and weak in the sense of being foundationally uncommitted (see also [de Bruijn \(1991\)](#)).

Therefore, to evaluate an individual framework, we must ask how it supports the solution of which concrete problems. More concretely, we must ask (*i*) what logics can be represented in it, (*ii*) what applications become possible for represented logics, and (*iii*) whether the same could be achieved with other, more elegant frameworks. We will discuss these questions in Sect. [6.1](#), [6.2](#), and [6.3](#), respectively.

### *6.1. Coverage and Limitations*

**Logics** Our definitions are chosen carefully to subsume both the framework of institutions and the one of LF so that existing logic representations can be reused or – in case they only cover either proof or model theory – complemented. The available institution-based representations of model theoretical logics (e.g., [Borzyszkowski \(2000\)](#); [CoFI \(The Common Framework Initiative\) \(2004\)](#); [Goguen and Burstall \(1992\)](#); [Mossakowski et al. \(2007\)](#)) become logic representations in our sense once their syntax and proof theory are made explicit in LF. Similarly, the available LF-based proof theoretical logic representations (e.g., [Avron et al. \(1992, 1998\)](#); [Harper et al. \(1993, 1994\)](#); [Pfenning \(2000, 2001\)](#)) become uniform logic definitions in our sense after encoding their model theory. In many cases, existing institution and LF-based representations of the same logic can be paired to complement each other.

Our definition of logic covers a wide range of logics including propositional, first-order, or higher-order logics; logics based on untyped, simply typed, dependently typed, or polymorphic type theories; modal, temporal, and description logics; and logics with classical or intuitionistic semantics. Multiple truth values such as in three-valued logics are supported if some truth values are *designated* to obtain a two-valued satisfaction relation in the models (as we did in [Goguen et al. \(2007\)](#)), or if the syntax includes constants for the truth values.

The syntax of a logic can typically be expressed in our meta-logic  $\mathbb{M}$ . A limitation concerns logics where the notion of well-formed formulas is undecidable (such as in PVS, [Owre et al. \(1992\)](#)). This requires to merge  $\Sigma^{syn}$  and  $\Sigma^{pf}$  to axiomatize well-formedness and take a subtype of `base(form)` as the type of sentences. This is possible with the more general framework we used in [Rabe \(2008\)](#), but we have omitted this here for simplicity.

Regarding proof theories,  $\mathbb{M}$  inherits from LF the support of Hilbert, natural deduction, sequent, or tableaux calculi. Resolution-based proof theories cannot be represented as elegantly. Also, our notions of proof categories is biased against substructural logics, see Rem. [1](#). This is a bias shared with both institutions and LF, the latter has even been extended to a linear variant for that reason ([Cervesato and Pfenning \(2002\)](#)).

The representation of model theories in our meta-logic depends on two things. Firstly, the foundation must be expressible in LF; this is typically the case. Secondly, it must be adequate to represent models as morphisms; we discussed this in Rem. [16](#).

Our logical framework permits expressing signatures, sentences, models, and proofs of logics as LF objects. In addition, we identified two extensions of LF that we need to reach a fully comprehensive framework. Firstly, logical relations should be developed and used

to represent model morphisms (see Rem. 11). Secondly, a language of declaration patterns is needed to express the category of signatures as a whole as opposed to individual signatures (see Rem. 15).

**Logic Translations** For logic translations, the situation is more complicated. Firstly, logic translations are less well classified than logics, which complicates a systematic study of which classes of logic translations are covered. Secondly, the representation of logic translations is significantly harder than that of logics. In general, features present in covered syntax translations are expressing the model theory of one logic using the syntax of another, or coding sentences of one logic as terms of another logic.

A sentence translation is covered if it is compositional up to a final toplevel step as in Ex. 18.

A proof theory translation is covered if it translates inference rules to derivable rules. This excludes the non-compositional elimination of admissible rules such as cut elimination (where sentences and models are translated to themselves and proofs to cut-free proofs). Representing such translations requires a stronger notion of LF signature morphism that supports computation, e.g., along the lines of Twelf’s logic programs, Delphin ([Poswolsky and Schürmann \(2008\)](#)), or Beluga ([Pientka and Dunfield \(2010\)](#)). But the typing invariants of such signature morphisms would be a lot more complicated.

Model theory translations are typically covered if the model theories themselves are covered.

In Rem. 18, we have already discussed the need for non-simple  $\mathbf{Sig}^M$ -morphisms. We expect this to be a general phenomenon: While within-logic translations will always be represented by simple morphisms, the representation of more complex across-logic translations will require more complex notions of  $\mathbf{Sig}^M$ -morphisms. Consequently, we expect future work to further generalize the definition of  $\mathbf{Sig}^M$ -morphisms.

For example, some translations can only be encoded if the commutativity condition of Def. 18 is relaxed, especially for the lower rectangle dealing with the model translation. For example, the semantics of description logic interprets concepts as subsets of the universe. But the translation of description logic into first-order logic translates concepts to unary predicates, which the semantics of first-order logic interprets as functions from the universe to the booleans. Thus, the model translation commutes only up to the isomorphism between subsets and their characteristic functions. Recently a weaker condition than commutativity was shown to be sufficient to obtain logic comorphisms in [Sojakova \(2010\)](#), namely commutativity up to certain logical relations (see also Rem. 11).

Finally there are some limitations of logic comorphisms, which are inherited from institution comorphisms. Firstly, logic comorphisms are limited to total translations. This excludes partial translations such as a translation from higher-order to first-order logic that is undefined for expressions containing  $\lambda$ . Such translations are important in practice to borrow (semi-)automated theorem provers (e.g., the use of first-order provers in Isabelle) but difficult to describe in a logical framework. To overcome that, we recently designed an extension of LF that supports partial signature morphisms ([Dumbrava and Rabe \(2010\)](#)).

Secondly, logic comorphisms separate the signature and the sentence translation. But there are applications of borrowing where a different signature translation is used for different conjectures over the same signature. For example, the Leo-II prover ([Benzmüller et al. \(2008\)](#)) uses a translation from higher-order logic to sorted first-order logic by creating a new first-order sort for every function type that is mentioned in the higher-order conjecture. Such translations remain future work.

**Case Studies** In [Rabe and Kohlhase \(2011\)](#), we designed a generic module system based on theories and theory morphisms. In [Rabe and Schürmann \(2009\)](#), we extended the Twelf implementation of LF with the corresponding instance of this module system. We have used this implementation to conduct several large case studies where logics and logic translations are defined and machine-checked efficiently.

A large number of case studies has been conducted in the LATIN project [Codescu et al. \(2011\)](#). These include various incarnations of first-order logics, higher-order logics, modal and description logics as well as a number of set and type theoretical foundations. These are written as modular LF signatures and morphisms, and the overall size of the atlas exceeds 1000 modules.

The most extensive case studies were presented in [Horozal and Rabe \(2011\)](#) and [Iancu and Rabe \(2011\)](#). In [Horozal and Rabe \(2011\)](#), we give a comprehensive representation of first-order logic. It includes a formalization of ZFC set theory to formalize the model theory of FOL and a formalized soundness proof of FOL in the sense of Thm. 6.

The biggest investment when representing logics in M is the formalization of the foundation of mathematics. Therefore, we formalized three important foundations in [Iancu and Rabe \(2011\)](#): Zermelo Fraenkel set theory ([Fraenkel \(1922\)](#); [Zermelo \(1908\)](#)), the logical framework Isabelle and the higher-order logic Isabelle/HOL ([Nipkow et al. \(2002\)](#); [Paulson \(1994\)](#)), and the Mizar system for Tarski-Grothendieck set theory ([Trybulec and Blair \(1985\)](#)). These include translations between the foundations, which permits even the translation of models across foundations.

## 6.2. Applications

Our framework was designed in response to a number of specific problems we encountered in practice. In all cases, the lack of a comprehensive framework like ours had previously precluded general solutions.

**An Atlas of Logics** An important application of logical frameworks is to structure and relate the multitude of logics in use. This goal was a central motivation of institutions and has been pursued in LF as well ([Pfenning et al. \(2003\)](#)). And both in model and proof theoretical frameworks, there are substantial collections of presentations of model and proof theoretical logics, respectively. But a collection comprising both model and proof theory has so far been lacking.

Our framework provides the theoretical base of the LATIN project ([Codescu et al. \(2011\)](#), see also Sect. 6.1), which systematically builds such a collection of logic presentations. The LATIN atlas contains formalizations of logics and logic translations presented

as a diagram of  $\mathbb{M}$  signatures. It is made available as a web-based portal of documented logic definitions.

**Logic-Aware Machines** A central motivation of logical frameworks has been the goal to generically mechanize arbitrary logics: The implementation of one logical framework induces implementations of individual logics defined in it. This requires an abstract definition of logic and a machine-understandable language in which individual logics can be presented. Our framework provides both by presenting individual uniform logics as  $\mathbb{M}$ -signatures, i.e., LF spans. Therefore, we can now use implementations of LF – such as Twelf ([Pfenning and Schürmann \(1999\)](#)) – to mechanically process logic presentations.

We have given an example for this work flow in [Codescu et al. \(2012\)](#). The Hets system ([Mossakowski et al. \(2007\)](#)) implements the framework of institutions and acts as a mediator between parsers, static analyzers, and theorem provers for various object logics. But previously Hets could only work with a fixed number of institutions and comorphisms implemented individually in the underlying programming language. Our framework connects Hets to declarative logic presentations in LF, and Hets is now able to work with arbitrary uniform logics defined in  $\mathbb{M}$ .

**Rapidly Prototyping Logics** The definition of a new logic is typically the result of a long quest, and the evaluation of candidate logics often requires large case studies, which in turn usually require sophisticated machine support. Logical frameworks can support this process by permitting the swift implementation of candidate logics.

Using our framework, users can present candidate logics concisely and quickly, and Twelf immediately induces an implementation. In particular, Twelf provides type reconstruction and module system out of the box. Both features require a major investment to realize for individual logics but are indispensable in practice. Moreover, the module system can be used to build a library of logic components that serve as building blocks for large logics.

We employ this methodology in the LATIN atlas: Each logic feature (such as a connective or a type formation operator) is presented separately along with its proof and model theoretical semantics, and logics are composed using colimits. Therefore, the presentation of each logic only requires formalizing those features not already present in LATIN. Moreover, LATIN includes a library of modular formalizations of a variety of type and set theories so that logics can be built easily on top of and interpreted in a wide range of languages.

**Verified Heterogeneous Reasoning** At the moment mechanically verified formalizations of mathematics are restricted to individual proof theoretical logics, whose implementations find and verify theorems, e.g., Isabelle/HOL [Nipkow et al. \(2002\)](#) or Mizar [Trybulec and Blair \(1985\)](#). In this context, heterogeneous reasoning, i.e., the reuse of theorems along logic translations, is problematic because the translation engine would lie outside the verifying system. Applications are usually limited to translating and dynamically re-verifying a suite of theorems (e.g., [Krauss and Schropp \(2010\)](#), [Keller and Werner \(2010\)](#)).

On the other hand, model theoretical adequacy (in the sense of Def. 13) has been used very successfully in the institution community to borrow proof systems along a statically verified logic translation (Cerioli and Meseguer (1997)). But while there are systems (like Hets Mossakowski et al. (2007)) that permit presenting and applying such translations, there is no mechanized support for verifying them.

Our framework provides such support. For example, Sojakova (2010) uses it to prove the model theoretical adequacy of the translation from modal logic to first-order logic. Well-formedness and adequacy of the translation reduce to type checking in LF, which is verified mechanically by Twelf.

**Logical Knowledge Management** In Rabe and Kohlhase (2011), we have developed the MMT interface language for logic-related applications. The latter include both deduction systems such as theorem provers, proof assistants, or type checkers as well as management systems such as databases, browsers, IDEs, or search engines. MMT combines a modular representation format with a scalable knowledge management infrastructure (Kohlhase et al. (2010)) to provide an exchange format for logics, signatures, and theories as well as their morphisms, for foundations, and for expressions, proofs, and models.

For example, these objects can be presented using Twelf, exported as MMT, and imported by any other application. In particular, authors can use Twelf's human-oriented concrete syntax including the reconstruction of omitted terms and types, and the importing application can use fully reconstructed and thus easily machine-processable MMT format. This work flow is used in Codescu et al. (2012) to present logics in Twelf and use them in Hets.

### 6.3. Related Work

**Frameworks based on Model Theory** There are several closely related frameworks that define logics and logic comorphisms (possibly with different names) as certain tuples of categorical objects.

Our definitions of logics and logic comorphisms follow and subsume those of *institutions* and institution comorphisms (Goguen and Burstall (1992)). We obtain a forgetful functor from logics to institutions by dropping the proof theory from a logic and the proof translation from a logic comorphism. Similarly, if we drop the condition on proof translations in Def. 10, we recover institution comorphism modifications. These were introduced in Diaconescu (2002), albeit with a weaker condition; in Tarlecki (1996) a similar concept was called a representation map. Our Def. 10 extends the definition of institution comorphism modification given in Mossakowski (2005) in the expected way.

In Meseguer (1989), *general logics* are introduced as tuples  $(\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models, \vdash)$  where  $\vdash$  is an *entailment system* between the sentences. Entailment systems formalize the provability relation without formalizing proofs. Our use of  $\vdash_\Sigma F$  as a truth judgment is different from entailment systems, but our provability relation  $\Theta \vdash_\Sigma F$  is always an entailment system. In addition, Meseguer (1989) defines a proof calculus as a functor from theories to a fixed but arbitrary category  $\mathbf{Str}$ . Our definition of proof categories can be

seen as the special case  $\mathbf{Str} = \mathcal{PFCAT}$ ; in this special case, proofs can be represented as morphisms.

In Mossakowski et al. (2005) and Diaconescu (2006) *proof theoretic institutions* are introduced as essentially tuples  $(\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models, \mathbf{Pf})$ . Here the relation between sentences and objects of the proof category is predetermined because the objects of the proof categories are always the sets of sentences. Our definition of proof categories is more general and uses the truth judgment  $\vdash$  to relate sentences to objects in the proof category. In particular, the objects of our proof categories can be the multi-sets of sentences, which solves a problem discovered by us in an early revision of Diaconescu (2006): There the free generation of a proof system by a set of rules is restricted to logics with signature morphisms  $\sigma : \Sigma \rightarrow \Sigma'$  for which  $\mathbf{Sen}(\sigma)$  is injective. Otherwise,  $\sigma$  would not induce a canonical functor  $\mathbf{Pf}(\sigma)$  between proof categories. Later Lawvere theories were used to overcome this problem in the revised version of Mossakowski et al. (2005): There, no size restriction on the products is used, and proof categories are not small.

Mossakowski et al. (2005) also generalizes to  $\mathbb{C}/\mathbb{D}$ -*institutions* where  $\mathbb{C}$  and  $\mathbb{D}$  are the codomains of  $\mathbf{Pf}$  and  $\mathbf{Mod}$ , respectively, and  $\mathbf{Sen}$  is obtained by composing  $\mathbf{Pf}$  with a fixed functor  $\mathbb{C} \rightarrow \mathcal{SET}$ . Our logics are almost  $\mathcal{PFCAT}/\mathcal{CAT}$  institutions; the difference is again that we give  $\mathbf{Sen}$  and  $\mathbf{Pf}$  separately and relate them via  $\vdash$ .

In Fiadeiro and Sernadas (1988),  $\Pi$ -*institutions* are introduced as tuples  $(\mathbf{Sig}, \mathbf{Sen}, Cn)$  where  $Cn$  is a closure operator on sets of formulas defining consequence.  $Cn$  is treated proof theoretically, and theory morphisms are studied proof theoretically.

The idea of using a meta-logic like  $\mathbb{M}$  is well-known, and was studied for institutions in e.g., Tarlecki (1996). Our generalization to logic encodings is not surprising, and our notion of model theoretical adequacy is known as the model expansion property. The major novelty in our work is the use of LF as a concrete meta-logic.

In Martí-Oliet and Meseguer (1996), *rewriting logic* is proposed as another concrete meta-logic within an informal framework similar to general logics. Rewriting logic arises by combining sorted first-order logic with term rewriting, which makes it very different from LF. In rewriting logic, typically, the syntax of an object logic is represented as a sorted first-order theory, and the proof theory as a set of rewriting rules; the model theory is not represented syntactically but via the model theory of rewriting logic. The main advantage of rewriting logic over LF is the use of rewriting to simplify expressions; the main advantage of LF is the use of dependent type theory and of higher-order abstract syntax to represent proofs and variable binding as expressions.

The most advanced implementations of such frameworks are the Hets implementation of institutions (Mossakowski et al. (2007)) and the Maude implementation of a simplified variant of rewriting logic (Clavel et al. (1996)). Hets uses Haskell as an implementation language for the syntax of institutions and comorphisms. It maintains a graph of logics via which theories can be translated between logics and serves as middleware between implementations of individual logics. Maude uses theories of rewriting logic to represent the syntax and proof theory of object logics as theories of the meta-logic. It implements term rewriting to provide computation within the logical framework.

Hets implements an abstract framework (institutions) whereas Maude implements a

concrete meta-logic within a given framework (general logics). The latter corresponds to our approach: Twelf ([Pfenning and Schürmann \(1999\)](#)) implements LF and thus  $\mathbb{M}$  within the framework of our logics; in particular, logics are represented declaratively. The former is complementary to our approach, for example, we have used  $\mathbb{M}$  as a convenient way to add logics to Hets in [Codescu et al. \(2012\)](#). Unlike Hets and Maude, our approach also permits the mechanized representation of model theory.

**Frameworks based on Proof Theory** Our definition of  $\mathbb{M}$  subsumes the logical framework LF. The use of LF signatures  $L^{syn}$  and  $L^{pf}$  to obtain proof theoretical logic encodings are well-understood (see, e.g., [Pfenning \(2001\)](#)). The reasoning about adequacy of these encodings corresponds to our comorphisms  $(\Phi, \alpha, \gamma)$ . A difference is that in the proof theoretical community, such encodings are considered adequate if  $\alpha$  and  $\gamma$  are isomorphisms; we require only a weaker condition on  $\gamma$  here.

Isabelle ([Paulson \(1994\)](#)) is an alternative logical framework, which uses higher-order logic with shallow polymorphism ([Church \(1940\)](#)) instead of dependent type theory. The main advantage over LF are recursive and inductive computation and (semi-)automated reasoning support. Other type theories such as Martin-Löf type theory in Agda ([Martin-Löf \(1974\); Norell \(2005\)](#)) or the calculus of constructions in Coq ([Bertot and Castéran \(2004\); Coquand and Huet \(1988\)](#)) are sometimes used as logical frameworks as well and have similar advantages. The main advantage of LF are the use of higher-order abstract syntax, which facilitates adequacy proofs, and the support for signature morphisms in Twelf ([Pfenning and Schürmann \(1999\)](#)).

Our definition of  $\mathbb{M}$  and our results about it depend only on a few properties of LF. The main assumptions are that the signatures and for each signature the contexts form categories, and that there are pushouts along inclusions. This is the case for almost all type theories. Thus, our definitions can be generalized easily to most type theories subsuming LF such as Martin-Löf type theory or the calculus of constructions. Even though the signature *Base* uses dependent types, it is also easy to adapt the definitions to (simply-typed) higher-order logic. Then the signature *Base* contains `ded : form → prop` where `prop` is the type of propositions. This corresponds to logic encodings in Isabelle where `ded` is typically called `Trueprop`.

While logic translations play a major role in model theoretical frameworks, they have been studied less systematically for proof theoretical frameworks. The most comprehensive approach was the Logosphere project ([Pfenning et al. \(2003\)](#)), which employed LF as a framework for logic translations. These were implemented in two ways: firstly, using the operational semantics of Twelf based on logic programming, secondly using the specifically developed functional programming language Delphin ([Poswolsky and Schürmann \(2008\)](#)). Our use of LF signature morphisms differs because our translations are represented in a declarative language.

Other logic translations are typically represented ad-hoc, i.e., as implementations outside a logical framework, e.g., [Keller and Werner \(2010\); Krauss and Schropp \(2010\); McLaughlin \(2006\); Obua and Skalberg \(2006\)](#). Instead of appealing to general results about the framework, the correctness of such a translation can be guaranteed by veri-

fying the translated proofs. The disadvantage is that proofs must be produced, stored, translated, and verified for each translated theorem.

**Representing Models** Our representation of models as morphisms goes back to Lawvere's representation of models as functors ([Lawvere \(1963\)](#)) and the use of initial algebras as the semantics of theories ([Goguen et al. \(1978\)](#)). This approach has been applied most systematically in the area of categorical models of type theories (see, e.g., [Pitts \(2000\)](#)) and has been integrated into model theoretical frameworks (e.g., [Fiadeiro and Sernadas \(1988\)](#); [Goguen and Burstall \(1986\)](#); [Goguen et al. \(2007\)](#)). The representation of models as signature morphisms into some fixed signature has also been used to show the amalgamation property of given institutions (see, e.g., [Diaconescu \(2008\)](#)).

From a model theoretical perspective, the novelty of our approach is to use morphisms in a category – namely  $\text{LF}$  – whose objects and morphisms are given in terms of concrete syntax and do not depend on any foundation of mathematics. That makes it possible to represent models as expressions in a mechanized language. Moreover, by separating  $\Sigma^{syn}$  and  $\Sigma^{mod}$  and by using morphisms out of  $\Sigma^{mod}$  as models (rather than morphisms out of  $\Sigma^{syn}$ ), we can represent logics whose models have a different structure than the syntax. For example, Kripke models of modal logic use a set of worlds and an accessibility relation, which are not present in the syntax.

In type theories, model theory has been represented occasionally in a similar way. For example, in [Benton et al. \(2009\)](#); [Coquand and Dybjer \(1997\)](#) the syntax of formal languages is represented as an inductive data type and models as inductive functions out of it. Isabelle ([Paulson \(1994\)](#)) provides locales and interpretations, which behave very similarly to LF signatures and signature morphisms.

From a type theoretical perspective, the novelty of our approach is the systematic design of a logical framework on top of this intuition. Moreover, we abstract from the foundation by permitting the use of an arbitrary  $\text{LF}$  signature. Thus, we avoid the commitment to a specific foundation such as higher-order logic or Martin-Löf type theory that is often inherent in type theoretical representations of models.

## 7. Conclusion

**A Comprehensive Logical Framework** We gave a logical framework that permits comprehensive logic representations: The syntax, model theory, and proof theory of logics as well as their translations are represented within a formal logical framework. In particular, our approach combines the model theoretical and the proof theoretical perspectives. Despite the ontological and philosophical differences of these two perspectives on logic, we are able to preserve the flavor and advantages of either one.

Specifically, we extended the model theoretical framework of institutions ([Goguen and Burstall \(1992\)](#)) with the notions of judgments and proof categories. Our definitions preserve the elegance and abstraction of institutions while permitting a natural integration of proof theoretical logics, which can be seen as the continuation of work undertaken in [Meseguer \(1989\)](#) and [Mossakowski et al. \(2005\)](#).

Correspondingly, we extended the proof theoretical framework LF (Harper et al. (1993)) with the notions of homomorphisms and model categories. We use a logic based on LF as a meta-logic in which object logics are represented, a reply to a suggestion made in Tarlecki (1996). Using this meta-logic, all logical notions are defined as syntactic objects in the LF type theory and can thus be verified mechanically. This includes a special LF signature representing the foundation of mathematics so that we can represent model theory even though our meta-logic is foundationally uncommitted.

**Evaluation** Using the criteria we gave at the beginning of Sect. 6, we can conclude as follows. (i) As described in Sect. 6.1, our framework preserves the large collection of existing logic encodings in institutions and LF. Moreover, since these encodings usually covered either the model theory or the proof theory, in many cases we can now give comprehensive encodings for the first time. Regarding logic translations, our work errs on the side of simplicity in the simplicity-expressivity trade-off, and extensions remain future work.

(ii) In Sect. 6.2, we described a number of applications that are enabled by our framework, focusing on those that are already underway. The key feature here is that our framework provides both an abstract definition of logics and a simple declarative language, in which such logics can be presented concisely and easily.

(iii) Finally, our framework has been designed systematically as the simplest possible evolution of the existing frameworks which combines the above two properties. A special strength is the symmetric treatment of model theory ( $\mathbf{Mod}, \models$ ) and proof theory ( $\mathbf{Pf}, \vdash$ ). Moreover, the models-as-morphisms paradigm yields an elegant formalist representation of platonist model theory.

**Future Work** Future work will focus on three lines of research. Firstly, the theoretical framework and the tool support that is in place now can be leveraged in the practical applications outlined in Sect. 6.2. In particular, we have started the work on a logic atlas in the LATIN project (Codescu et al. (2011)).

Secondly, some of the limitations discussed in Sect. 6.1 can be overcome by using different underlying categories than  $\mathbb{LF}$ . For example, we can use a type theory with non-compositional or partial functions. Our results can be extended easily to such type theories.

Thirdly, our framework can be used for the theoretical analysis of a logic's meta-theory. This includes the mechanization of soundness and completeness proofs: Soundness will use Thm. 10; completeness is currently open, but we consider the approach along the lines of Rem. 13 very promising. A related application is the modular development of logics as envisioned in Harper et al. (1994) and now implemented in the LATIN project.

**Acknowledgments** This work was developed over the course of five years, during which the author was supported by PhD scholarships from Jacobs University Bremen, the German Academic Exchange Service, and the German Merit Foundation. Additional support was received from research grants by the US American National Science Foundation

(Logosphere grant, no. CCR-ITR-0325808) and the German Research Council (LATIN grant, no. KO-2428/9-1), and from IT University Copenhagen.

The author was given the idea central of combining institutions and LF by Michael Kohlhase. The definitions in Sect. 3 and 4 owe to discussions with Răzvan Diaconescu, Joseph Goguen, Till Mossakowski, Frank Pfenning, and Andrzej Tarlecki. The evaluation of the research presented here relied on collaboration with Carsten Schürmann on the implementation of the Twelf module system and with Fulya Horozal, Mihnea Iancu, and Kristina Sojakova on the conduction of large scale case studies.

## References

- Aiguier, M. and Diaconescu, R. (2007). Stratified institutions and elementary homomorphisms. *Information Processing Letters*, 103(1):5–13.
- Andrews, P. (1986). *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press.
- Avron, A., Honsell, F., Mason, I., and Pollack, R. (1992). Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9(3):309–354.
- Avron, A., Honsell, F., Miculan, M., and Paravano, C. (1998). Encoding modal logics in logical frameworks. *Studia Logica*, 60(1):161–208.
- Awodey, S. and Rabe, F. (2011). Kripke Semantics for Martin-Löf’s Extensional Type Theory. *Logical Methods in Computer Science*, 7(3).
- Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P., editors (2003). *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press.
- Barendregt, H. (1992). Lambda calculi with types. In Abramsky, S., Gabbay, D., and Maibaum, T., editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press.
- Barwise, J. (1977). An Introduction to First-Order Logic. In Barwise, J., editor, *Handbook of Mathematical Logic*, pages 5–46. North-Holland.
- Benton, N., Kennedy, A., and Varming, C. (2009). Some Domain Theory and Denotational Semantics in Coq. In Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M., editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 115–130. Springer.
- Benzmüller, C., Paulson, L., Theiss, F., and Fietzke, A. (2008). LEO-II - A Cooperative Automatic Theorem Prover for Classical Higher-Order Logic (System Description). In Armando, A., Baumgartner, P., and Dowek, G., editors, *Automated Reasoning*, pages 162–170. Springer.
- Bertot, Y. and Castéran, P. (2004). *Coq’Art: The Calculus of Inductive Constructions*. Springer.
- Béziau, J., editor (2005). *Logica Universalis*. Birkhäuser Verlag.
- Borzyszkowski, T. (2000). Higher-Order Logic and Theorem Proving for Structured Specifications. In Choppy, C., Bert, D., and Mosses, P., editors, *Workshop on Algebraic Development Techniques*, LNCS, pages 401–418.

- Bourbaki, N. (1974). *Algebra I*. Elements of Mathematics. Springer.
- Brachman, R. and Schmolze, J. (1985). An Overview of the KL-ONE Knowledge Representation Scheme. *Cognitive Science*, 9(2).
- Brouwer, L. (1907). *Over de grondslagen der wiskunde*. PhD thesis, Universiteit van Amsterdam. English title: On the Foundations of Mathematics.
- Cartmell, J. (1986). Generalized algebraic theories and contextual category. *Annals of Pure and Applied Logic*, 32:209–243.
- Căzănescu, V. and Roșu, G. (1997). Weak inclusion systems. *Mathematical Structures in Computer Science*, 7(2):195–206.
- Cerioli, M. and Meseguer, J. (1997). May I Borrow Your Logic? (Transporting Logical Structures along Maps). *Theoretical Computer Science*, 173:311–347.
- Cervesato, I. and Pfenning, F. (2002). A Linear Logical Framework. *Information and Computation*, 179(1):19–75.
- Church, A. (1940). A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68.
- Clavel, M., Eker, S., Lincoln, P., and Meseguer, J. (1996). Principles of Maude. In Meseguer, J., editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4, pages 65–89.
- Codecscu, M., Horozal, F., Kohlhase, M., Mossakowski, T., and Rabe, F. (2011). Project Abstract: Logic Atlas and Integrator (LATIN). In Davenport, J., Farmer, W., Rabe, F., and Urban, J., editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 287–289. Springer.
- Codecscu, M., Horozal, F., Kohlhase, M., Mossakowski, T., Rabe, F., and Sojakova, K. (2012). Towards Logical Frameworks in the Heterogeneous Tool Set Hets. In Mossakowski, T. and Kreowski, H., editors, *Recent Trends in Algebraic Development Techniques 2010*, volume 7137 of *Lecture Notes in Computer Science*, pages 139–159. Springer.
- CoFI (The Common Framework Initiative) (2004). *CASL Reference Manual*, volume 2960 of *LNCS*. Springer.
- Coquand, T. and Dybjer, P. (1997). Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7(1):75–94.
- Coquand, T. and Huet, G. (1988). The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120.
- Curry, H. and Feys, R. (1958). *Combinatory Logic*. North-Holland, Amsterdam.
- de Bruijn, N. (1970). The Mathematical Language AUTOMATH. In Laudet, M., editor, *Proceedings of the Symposium on Automated Demonstration*, volume 25 of *Lecture Notes in Mathematics*, pages 29–61. Springer.
- de Bruijn, N. (1991). A Plea for Weaker Frameworks. In Huet, G. and Plotkin, G., editors, *Logical Frameworks*, pages 40–67. Cambridge University Press.
- Diaconescu, R. (2002). Grothendieck institutions. *Applied Categorical Structures*, 10(4):383–402.
- Diaconescu, R. (2006). Proof systems for institutional logic. *Journal of Logic and Computation*, 16(3):339–357.
- Diaconescu, R. (2008). *Institution-independent Model Theory*. Birkhäuser.

- Dumbrava, S. and Rabe, F. (2010). Structuring Theories with Partial Morphisms. In *Workshop on Algebraic Development Techniques*.
- Farmer, W. (2010). Chiron: A set theory with types, undefinedness, quotation, and evaluation. SQRL Report 38, McMaster University.
- Farmer, W., Guttman, J., and Thayer, F. (1992). Little Theories. In Kapur, D., editor, *Conference on Automated Deduction*, pages 467–581.
- Fiadeiro, J. and Sernadas, A. (1988). Structuring theories on consequence. In *Recent Trends in Data Type Specification*, pages 44–72. Springer.
- Fraenkel, A. (1922). Zu den Grundlagen der Cantor-Zermeloschen Mengenlehre. *Mathematische Annalen*, 86:230–237. English title: On the Foundation of Cantor-Zermelo Set Theory.
- Gentzen, G. (1934). Untersuchungen über das logische Schließen. *Math. Z.*, 39. English title: Investigations into Logical Deduction.
- Girard, J. (1987). Linear Logic. *Theoretical Computer Science*, 50:1–102.
- Gödel, K. (1930). Die Vollständigkeit der Axiome des Logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik*, 37:349–360. English title: The Completeness of the Axioms of the Logical Calculus of Functions.
- Goguen, J. and Burstall, R. (1986). A study in the foundations of programming methodology: specifications, institutions, charters and parchments. In Pitt, D., Abramsky, S., Poigné, A., and Rydeheard, D., editors, *Workshop on Category Theory and Computer Programming*, pages 313–333. Springer.
- Goguen, J. and Burstall, R. (1992). Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146.
- Goguen, J., Mossakowski, T., de Paiva, V., Rabe, F., and Schröder, L. (2007). An Institutional View on Categorical Logic. *International Journal of Software and Informatics*, 1(1):129–152.
- Goguen, J. and Rosu, G. (2002). Institution morphisms. *Formal Aspects of Computing*, 13:274–307.
- Goguen, J., Thatcher, J., and Wagner, E. (1978). An initial algebra approach to the specification, correctness and implementation of abstract data types. In Yeh, R., editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice Hall.
- Gordon, M. (1988). HOL: A Proof Generating System for Higher-Order Logic. In Birtwistle, G. and Subrahmanyam, P., editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer-Academic Publishers.
- Haftmann, F. and Wenzel, M. (2006). Constructive Type Classes in Isabelle. In Altenkirch, T. and McBride, C., editors, *TYPES conference*, pages 160–174. Springer.
- Harper, R., Honsell, F., and Plotkin, G. (1993). A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184.
- Harper, R., Sannella, D., and Tarlecki, A. (1994). Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160.
- Harrison, J. (1996). HOL Light: A Tutorial Introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 265–269. Springer.
- Henkin, L. (1950). Completeness in the Theory of Types. *Journal of Symbolic Logic*, 15(2):81–91.

- Hilbert, D. (1926). Über das Unendliche. *Mathematische Annalen*, 95:161–90.
- Horozal, F. and Rabe, F. (2011). Representing Model Theory in a Type-Theoretical Logical Framework. *Theoretical Computer Science*, 412(37):4919–4945.
- Howard, W. (1980). The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press.
- Iancu, M. and Rabe, F. (2011). Formalizing Foundations of Mathematics. *Mathematical Structures in Computer Science*, 21(4):883–911.
- Keller, C. and Werner, B. (2010). Importing HOL Light into Coq. In Kaufmann, M. and Paulson, L., editors, *Interactive Theorem Proving*, pages 307–322. Springer.
- Kohlhase, M., Rabe, F., and Zhiludev, V. (2010). Towards MKM in the Large: Modular Representation and Scalable Software Architecture. In Autexier, S., Calmet, J., Delahaye, D., Ion, P., Rideau, L., Rioboo, R., and Sexton, A., editors, *Intelligent Computer Mathematics*, volume 6167 of *Lecture Notes in Computer Science*, pages 370–384. Springer.
- Krauss, A. and Schropp, A. (2010). A Mechanized Translation from Higher-Order Logic to Set Theory. In Kaufmann, M. and Paulson, L., editors, *Interactive Theorem Proving*, pages 323–338. Springer.
- Lambek, J. and Scott, P. (1986). *Introduction to Higher-Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press.
- Lawvere, F. (1963). *Functional Semantics of Algebraic Theories*. PhD thesis, Columbia University.
- Lawvere, W. (1969). Adjointness in Foundations. *Dialectica*, 23(3–4):281–296.
- Mac Lane, S. (1998). *Categories for the working mathematician*. Springer.
- Martí-Oliet, N. and Meseguer, J. (1996). Rewriting Logic as a Logical and Semantic Framework. In *Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 352–358.
- Martin-Löf, P. (1974). An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the '73 Logic Colloquium*, pages 73–118. North-Holland.
- Martin-Löf, P. (1996). On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):3–10.
- McLaughlin, S. (2006). An Interpretation of Isabelle/HOL in HOL Light. In Shankar, N. and Furbach, U., editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*. Springer.
- Meseguer, J. (1989). General logics. In Ebbinghaus, H.-D. et al., editors, *Proceedings, Logic Colloquium, 1987*, pages 275–329. North-Holland.
- Mossakowski, T. (2005). Heterogeneous Specification and the Heterogeneous Tool Set. Habilitation thesis, see <http://www.informatik.uni-bremen.de/~till/>.
- Mossakowski, T., Goguen, J., Diaconescu, R., and Tarlecki, A. (2005). What is a logic? In Béziau, J., editor, *Logica Universalis*, pages 113–133. Birkhäuser Verlag.
- Mossakowski, T., Maeder, C., and Lüttich, K. (2007). The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522.

- Mossakowski, T., Tarlecki, A., and Pawłowski, W. (1997). Combining and Representing Logical Systems. In Moggi, E. and Rosolini, G., editors, *Category Theory and Computer Science*, pages 177–196. Springer.
- Naumov, P., Stehr, M., and Meseguer, J. (2001). The HOL/NuPRL proof translator - a practical approach to formal interoperability. In *14th International Conference on Theorem Proving in Higher Order Logics*. Springer.
- Nipkow, T., Paulson, L., and Wenzel, M. (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer.
- Norell, U. (2005). The Agda WiKi. <http://wiki.portal.chalmers.se/agda>.
- Obua, S. and Skalberg, S. (2006). Importing HOL into Isabelle/HOL. In Shankar, N. and Furbach, U., editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*. Springer.
- Owre, S., Rushby, J., and Shankar, N. (1992). PVS: A Prototype Verification System. In Kapur, D., editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752. Springer.
- Paulson, L. (1994). *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer.
- Pfenning, F. (2000). Structural cut elimination: I. intuitionistic and classical logic. *Information and Computation*, 157(1-2):84–141.
- Pfenning, F. (2001). Logical frameworks. In *Handbook of automated reasoning*, pages 1063–1147. Elsevier.
- Pfenning, F. and Schürmann, C. (1999). System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206.
- Pfenning, F., Schürmann, C., Kohlhase, M., Shankar, N., and Owre, S. (2003). The Logosphere Project. <http://www.logosphere.org/>.
- Pientka, B. and Dunfield, J. (2010). A Framework for Programming and Reasoning with Deductive Systems (System description). In *International Joint Conference on Automated Reasoning*. To appear.
- Pitts, A. (2000). Categorical Logic. In Abramsky, S., Gabbay, D., and Maibaum, T., editors, *Handbook of Logic in Computer Science, Volume 5. Algebraic and Logical Structures*, chapter 2, pages 39–128. Oxford University Press.
- Poswolsky, A. and Schürmann, C. (2008). System Description: Delphin - A Functional Programming Language for Deductive Systems. In Abel, A. and Urban, C., editors, *International Workshop on Logical Frameworks and Metalanguages: Theory and Practice*, pages 135–141. ENTCS.
- Rabe, F. (2008). *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen. see <http://kwarc.info/frabe/Research/phdthesis.pdf>.
- Rabe, F. and Kohlhase, M. (2011). A Scalable Module System. see <http://arxiv.org/abs/1105.0548>.
- Rabe, F. and Schürmann, C. (2009). A Practical Module System for LF. In Cheney, J. and Felty, A., editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, volume LFMTP’09 of *ACM International Conference Proceeding Series*, pages 40–48. ACM Press.
- Robinson, A. (1950). On the application of symbolic logic to algebra. In *Proceedings of*

- the International Congress of Mathematicians, pages 686–694. American Mathematical Society.
- Seely, R. (1984). Locally cartesian closed categories and type theory. *Math. Proc. Cambridge Philos. Soc.*, 95:33–48.
- Smullyan, R. (1995). *First-Order Logic*. Dover, second corrected edition.
- Sojakova, K. (2010). Mechanically Verifying Logic Translations. Master’s thesis, Jacobs University Bremen.
- Tarlecki, A. (1996). Moving between logical systems. In Haveraaen, M., Owe, O., and Dahl, O.-J., editors, *Recent Trends in Data Type Specifications. 11th Workshop on Specification of Abstract Data Types*, volume 1130 of *Lecture Notes in Computer Science*, pages 478–502. Springer Verlag.
- Tarski, A. (1933). Pojęcie prawdy w językach nauk dedukcyjnych. *Prace Towarzystwa Naukowego Warszawskiego Wydział III Nauk Matematyczno-Fizycznych*, 34. English title: The concept of truth in the languages of the deductive sciences.
- Tarski, A. and Vaught, R. (1956). Arithmetical extensions of relational systems. *Compositio Mathematica*, 13:81–102.
- Trybulec, A. and Blair, H. (1985). Computer Assisted Reasoning with MIZAR. In Joshi, A., editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28.
- Whitehead, A. and Russell, B. (1913). *Principia Mathematica*. Cambridge University Press.
- Zermelo, E. (1908). Untersuchungen über die Grundlagen der Mengenlehre I. *Mathematische Annalen*, 65:261–281. English title: Investigations in the foundations of set theory I.

# Kripke Semantics for Martin-Löf’s Extensional Type Theory<sup>\*</sup>

Steve Awodey<sup>1</sup> and Florian Rabe<sup>2</sup>

<sup>1</sup> Carnegie Mellon University, Pittsburgh, USA

<sup>2</sup> Jacobs University Bremen, Germany

## Abstract

It is well-known that simple type theory is complete with respect to non-standard set-valued models. Completeness for standard models only holds with respect to certain extended classes of models, e.g., the class of cartesian closed categories. Similarly, dependent type theory is complete for locally cartesian closed categories. However, it is usually difficult to establish the coherence of interpretations of dependent type theory, i.e., to show that the interpretations of equal expressions are indeed equal. Several classes of models have been used to remedy this problem.

We contribute to this investigation by giving a semantics that is standard, coherent, and sufficiently general for completeness while remaining relatively easy to compute with. Our models interpret types of Martin-Löf’s extensional dependent type theory as sets indexed over posets or, equivalently, as fibrations over posets. This semantics can be seen as a generalization to dependent type theory of the interpretation of intuitionistic first-order logic in Kripke models. This yields a simple coherent model theory, with respect to which simple and dependent type theory are sound and complete.

## 1 Introduction and Related Work

Martin-Löf’s extensional type theory ([ML84], MLTT), is a dependent type theory. The main characteristic is that there are type-valued function symbols that take terms as input and return types as output. This is enriched with further type constructors such as dependent sum and product. The syntax of dependent type theory is significantly more complex than that of simple type theory because well-formed types and terms and both their equalities must be defined in a single joint induction.

---

<sup>\*</sup>The second author was partially supported by a fellowship for Ph.D. research of the German Academic Exchange Service.

The semantics of MLTT is similarly complicated. In [See84], the connection between MLTT and locally cartesian closed (LCC) categories was first established. LCC categories interpret contexts  $\Gamma$  as objects  $[\Gamma]$ , types in context  $\Gamma$  as objects in the slice category over  $[\Gamma]$ , substitution as pullback, and dependent sum and product as left and right adjoint to pullback. But there is a difficulty, namely that these three operations are not independent: Substitution of terms into types is associative and commutes with sum and product formation, which is not necessarily the case for the choices of pullbacks and their adjoints. This is known as the coherence or strictness problem and has been studied extensively. In incoherent models such as in [Cur89], equal types are interpreted as isomorphic but not necessarily equal objects. In [Car86], coherent models for MLTT are given using categories with attributes. And in [Hof94], a category with attributes is constructed for every LCC category. Several other model classes and their coherence properties have been studied in, e.g., [Str91] and [Jac90, Jac99]. In [Pit00], an overview is given.

These model classes all have in common that they are rather abstract and have a more complicated structure than general LCC categories. It is clearly desirable to have simpler, more concrete models. But it is a hard problem to equip a given LCC category with choices for pullbacks and adjoints that are both natural and coherent. Our motivation is to find a simple concrete class of LCC categories for which such a choice can be made, and which is still general enough to be complete for MLTT.

Mathematically, our main results can be summarized very simply: Using a theorem from topos theory, it can be shown that MLTT is complete with respect to — not necessarily coherent — models in the LCC categories of the form  $\mathcal{SET}^P$  for posets  $P$ , where  $\mathcal{SET}$  is the category of sets and mappings. This is equivalent to using presheaves on posets as models, which are often called Kripke models. They were also studied in [Hof97]. For these rather simple models, a solution to the coherence problem can be given.  $\mathcal{SET}$  can be equipped with a coherent choice of pullback functors, and hence the categories  $\mathcal{SET}^P$  can be as well. Deviating subtly from the well-known constructions, we can also make coherent choices for the required adjoints to pullback. Finally, rather than working in the various slices  $\mathcal{SET}^P/A$ , we use the isomorphism  $\mathcal{SET}^P/A \cong \mathcal{SET}^{f_P A}$ , where  $f_P A$  is the category of elements: Thus we can formulate the semantics of dependent types uniformly in terms of the simple categories of indexed sets  $\mathcal{SET}^Q$  for various posets  $Q$ .

In addition to being easy to work with, this has the virtue of capturing the idea that a dependent type  $S$  in context  $\Gamma$  is in some sense a type-valued function on  $\Gamma$ : Our models interpret  $\Gamma$  as a poset  $[\Gamma]$  and  $S$  as an indexed set  $[\Gamma|S] : [\Gamma] \rightarrow \mathcal{SET}$ . We speak of Kripke models because these models are a natural extension of the well-known Kripke models for intuitionistic first-order logic ([Kri65]). Such models are based on a poset  $P$  of worlds, and the universe is given as a  $P$ -indexed set (possibly equipped with  $P$ -indexed structure). This can be seen as the special case of our semantics when there is only one type.

In fact, our results are also interesting in the special case of simple type theory ([Chu40]). Contrary to Henkin models ([Hen50, MS89]), and the mod-

els given in [MM91], which like ours use indexed sets on posets, our models are standard: The interpretation  $\llbracket \Gamma | S \rightarrow S' \rrbracket$  of the function type is the exponential of  $\llbracket \Gamma | S \rrbracket$  and  $\llbracket \Gamma | S' \rrbracket$ . And contrary to the models in [Fri75, Sim95], our completeness result holds for theories with more than only base types and terms.

A different notion of Kripke-models for dependent type theory is given in [Lip92], which is related to [All87]. There, the MLTT types are translated into predicates in an untyped first-order language. The first-order language is then interpreted in a Kripke-model, i.e., there is one indexed universe of which all types are subsets. Such models correspond roughly to non-standard set-theoretical models.

We give the syntax of MLTT in Sect. 2 and some categorical preliminaries in Sect. 3. Then we derive the coherent functor choices in Sect. 4 and use them to define the interpretation in Sect. 5. We give our main results regarding the interpretation of substitution, soundness, and completeness in Sect. 6, 7, and 8. A preliminary version of this paper appeared as [AR09].

## 2 Syntax

### 2.1 Grammar

The basic syntax for MLTT expressions is given by the grammar in Fig. 1. The vocabulary of the syntax is declared in signatures and contexts: Signatures  $\Sigma$  declare globally accessible names  $c$  for constants of type  $S$  and names  $a$  for type-valued constants with a list  $\Gamma$  of argument types. Contexts  $\Gamma$  locally declare typed variables  $x$ .

Substitutions  $\gamma$  translate from a context  $\Gamma$  to  $\Gamma'$  by providing terms in context  $\Gamma'$  for the variables in  $\Gamma$ . Thus, a substitution from  $\Gamma$  to  $\Gamma'$  can be applied to expressions in context  $\Gamma$  and yields expressions in context  $\Gamma'$ . Relative to a signature  $\Sigma$  and a context  $\Gamma$ , there are two syntactical classes: types and typed terms.

The base types are the application  $a \gamma$  of a type-valued constant to a list of argument terms  $\gamma$  (which we write as a substitution for simplicity). The composed types are the unit type  $1$ , the identity types  $Id(s, s')$ , the dependent product types  $\Sigma_{x:S} T$ , and the dependent function types  $\Pi_{x:S} T$ . Terms are constants  $c$ , variables  $x$ , the element  $*$  of the unit type, the element  $refl(s)$  of the type  $Id(s, s)$ , pairs  $\langle s, s' \rangle$ , projections  $\pi_1(s)$  and  $\pi_2(s)$ ,  $\lambda$ -abstractions  $\lambda_{x:S} s$ , and function applications  $s s'$ . We do not need equality axioms  $s \equiv s'$  because they can be given as constants of type  $Id(s, s')$ . For simplicity, we omit equality axioms for types.

Our formulation of MLTT only uses types and terms. This is different from variants of dependent type theory with kinded type families as in [Bar92] and [HHP93]. In particular, in our formulation, the constants  $a$  are the only type families, and  $a$  itself is not a well-formed expression. All our results extend to the case with kinded type families (see [Rab08]).

|               |          |  |
|---------------|----------|--|
| Signatures    | $\Sigma$ | $::= \cdot \mid \Sigma, c:S \mid \Sigma, a:(\Gamma)\text{type}$  |
| Contexts      | $\Gamma$ | $::= \cdot \mid \Gamma, x:S$   |
| Substitutions | $\gamma$ | $::= \cdot \mid \gamma, x/s$   |
| Types         | $S$      | $::= a \gamma \mid 1 \mid Id(s, s') \mid \Sigma_{x:S} S' \mid \Pi_{x:S} S'$  |
| Terms         | $s$      | $::= c \mid x \mid * \mid refl(s) \mid \langle s, s' \rangle \mid \pi_1(s) \mid \pi_2(s) \mid \lambda_{x:S} s \mid s \ s'$ |

Figure 1: Basic Grammar

**Definition 1** (Substitution Application). The *application* of a substitution  $\gamma$  to a term, type, or substitution is defined as follows where  $\gamma^x$  abbreviates  $\gamma, x/x$ .

Substitution in terms:

$$\begin{aligned}
 \gamma(c) &:= c \\
 \gamma(x) &:= s && \text{for } x/s \text{ in } \gamma \\
 \gamma(*) &:= * \\
 \gamma(\text{refl}(s)) &:= \text{refl}(\gamma(s)) \\
 \gamma(\langle s, s' \rangle) &:= \langle \gamma(s), \gamma(s') \rangle \\
 \gamma(\pi_1(s)) &:= \pi_1(\gamma(s)) \\
 \gamma(\pi_2(s)) &:= \pi_2(\gamma(s)) \\
 \gamma(\lambda_{x:S} t) &:= \lambda_{x:\gamma(S)} \gamma^x(t) \\
 \gamma(f \ s) &:= \gamma(f) \ \gamma(s)
 \end{aligned}$$

Substitution in types:

$$\begin{aligned}
 \gamma(1) &:= 1 \\
 \gamma(Id(s, s')) &:= Id(\gamma(s), \gamma(s')) \\
 \gamma(\Sigma_{x:S} T) &:= \Sigma_{x:\gamma(S)} \gamma^x(T) \\
 \gamma(\Pi_{x:S} T) &:= \Pi_{x:\gamma(S)} \gamma^x(T) \\
 \gamma(a \ \gamma_0) &:= a \ \gamma(\gamma_0)
 \end{aligned}$$

Substitution in substitutions:

$$\begin{aligned}
 \gamma(\cdot) &:= \cdot \\
 \gamma(x_1/s_1, \dots, x_n/s_n) &:= x_1/\gamma(s_1), \dots, x_n/\gamma(s_n)
 \end{aligned}$$

Substitution in substitutions is the same as composition of substitutions, and we write  $\gamma \circ \delta$  instead of  $\gamma(\delta)$ .

## 2.2 Type System

The judgments defining well-formed syntax are listed in Fig. 2. The typing rules for these judgments are well-known. Our formulation follows roughly [See84], including the use of extensional identity types. The latter means that the equality judgment for the terms  $s$  and  $s'$  holds iff the type  $Id(s, s')$  is inhabited.

| Judgment  | Intuition   |
|---|---|
| $\vdash \Sigma \text{ Sig}$                           | $\Sigma$ is a well-formed signature   |
| $\vdash_{\Sigma} \Gamma \text{ Ctx}$                  | $\Gamma$ is a well-formed context over $\Sigma$                                 |
| $\vdash_{\Sigma} \gamma : \Gamma \rightarrow \Gamma'$ | $\gamma$ is a well-formed substitution over $\Sigma$ from $\Gamma$ to $\Gamma'$ |
| $\Gamma \vdash_{\Sigma} S : \text{type}$              | $S$ is a well-formed type over $\Sigma$ and $\Gamma$                            |
| $\Gamma \vdash_{\Sigma} S \equiv S'$                  | types $S$ and $S'$ are equal over $\Sigma$ and $\Gamma$                         |
| $\Gamma \vdash_{\Sigma} s : S$                        | term $s$ is well-formed with type $S$ over $\Sigma$ and $\Gamma$                |
| $\Gamma \vdash_{\Sigma} s \equiv s'$                  | terms $s$ and $s'$ are equal over $\Sigma$ and $\Gamma$                         |

Figure 2: Judgments

*Example 2.* The theory  $Cat$  of categories is given by declaring type-valued constants  $Ob$  and  $Mor$  and term-valued constants  $id$  and  $comp$  such that the following judgments hold

$$\begin{array}{lll}
 \cdot & \vdash_{Cat} Ob & : \text{type} \\
 x : Ob, y : Ob & \vdash_{Cat} Mor x y & : \text{type} \\
 x : Ob & \vdash_{Cat} id x & : Mor x x \\
 x : Ob, y : Ob, z : Ob, & & \\
 g : Mor y z, f : Mor x y & \vdash_{Cat} g \circ f & : Mor x z \\
 w : Ob, x : Ob, y : Ob, z : Ob, & & \\
 f : Mor w x, g : Mor x y, h : Mor y z & \vdash_{Cat} h \circ (g \circ f) \equiv (h \circ g) \circ f \\
 x : Ob, y : Ob, f : Mor x y & \vdash_{Cat} f \circ id x \equiv f \\
 x : Ob, y : Ob, f : Mor x y & \vdash_{Cat} id y \circ f \equiv f
 \end{array}$$

Here we have used two common abbreviations. (i)  $Mor$  is declared as  $Mor : (x : Ob, y : Ob)\text{type}$ , and we abbreviate the type application  $Mor x/s, y/t$  as  $Mor s t$ . (ii)  $\circ$  is declared as a constant

$$\circ : \Pi_{x:Ob} \Pi_{y:Ob} \Pi_{z:Ob} \Pi_{g:Mor y z} \Pi_{f:Mor x y} Mor x z$$

and we abbreviate  $\circ x y z g f$  as  $g \circ f$ . This is unambiguous because the values of the first three arguments can be inferred from the types of the last two arguments.

The axioms of a category are declared using the Curry-Howard equivalence ([CF58, How80]) of MLTT and intuitionistic first-order logic without negation ([See84]). For example, to obtain right-neutrality, we declare a constant

$$neutr : \Pi_{x:Ob} \Pi_{y:Ob} \Pi_{f:Mor x y} Id(f \circ id x, f)$$

Such a constant yields the corresponding equality judgment above using Rule  $e_{Id(-,-)}$  from Fig. 6.

The *rules* for signatures, contexts, and substitutions are given in Fig. 3. A *signature* is a list of declarations of type-valued constants  $a$  or term constants  $c$ . For example,  $a:(\Gamma)\text{type}$  means that  $a$  can be applied to arguments with types

given by  $\Gamma$  and returns a type. The domain of a signature is defined by  $\text{dom}(\cdot) = \emptyset$ ,  $\text{dom}(\Sigma, a:(\Gamma)\text{type}) = \text{dom}(\Sigma) \cup \{a\}$ , and  $\text{dom}(\Sigma, c:S) = \text{dom}(\Sigma) \cup \{c\}$ .

*Contexts* are similar to signatures except that they only declare variables ranging over terms. The domain of a context is defined as for signatures. A *substitution* from  $\Gamma$  to  $\Gamma'$  is a list of terms in context  $\Gamma'$  such that each term is typed by the corresponding type in  $\Gamma$ . Note that in a context  $x_1:S_1, \dots, x_n:S_n$ , the variable  $x_i$  may occur in  $S_{i+1}, \dots, S_n$ .

$$\begin{array}{c}
 \boxed{
 \begin{array}{c}
 \frac{}{\vdash \cdot \text{ Sig}} \Sigma. \quad \frac{\vdash \Sigma \text{ Sig} \quad \vdash_\Sigma S : \text{type} \quad c \notin \text{dom}(\Sigma)}{\vdash \Sigma, c:S \text{ Sig}} \Sigma_c \\
 \\ 
 \frac{\vdash \Sigma \text{ Sig} \quad \vdash_\Sigma \Gamma' \text{ Ctx} \quad a \notin \text{dom}(\Sigma)}{\vdash \Sigma, a:(\Gamma')\text{type} \text{ Sig}} \Sigma_a \\
 \\ 
 \frac{\vdash \Sigma \text{ Sig}}{\vdash \Sigma \cdot \text{ Ctx}} \Gamma. \quad \frac{\vdash_\Sigma \Gamma \text{ Ctx} \quad \Gamma \vdash_\Sigma S : \text{type} \quad x \notin \text{dom}(\Gamma)}{\vdash_\Sigma \Gamma, x:S \text{ Ctx}} \Gamma_x \\
 \\ 
 \frac{\vdash_\Sigma \Gamma' \text{ Ctx}}{\vdash_\Sigma \cdot : \cdot \rightarrow \Gamma'} \sigma. \quad \frac{\vdash_\Sigma \gamma : \Gamma \rightarrow \Gamma' \quad \Gamma \vdash_\Sigma S : \text{type} \quad \Gamma' \vdash_\Sigma s : \gamma(S)}{\vdash_\Sigma \gamma, x/s : \Gamma, x:S \rightarrow \Gamma'} \sigma_x
 \end{array}}
 \end{array}$$

Figure 3: Signatures, Contexts, Substitutions

$$\begin{array}{c}
 \boxed{
 \begin{array}{c}
 \frac{a:(\Gamma_0)\text{type} \text{ in } \Sigma \quad \vdash_\Sigma \gamma_0 : \Gamma_0 \rightarrow \Gamma}{\Gamma \vdash_\Sigma a \gamma_0 : \text{type}} T_{app} \\
 \\ 
 \frac{\vdash_\Sigma \Gamma \text{ Ctx}}{\Gamma \vdash_\Sigma 1 : \text{type}} T_1 \quad \frac{\Gamma \vdash_\Sigma s : S \quad \Gamma \vdash_\Sigma s' : S}{\Gamma \vdash_\Sigma Id(s, s') : \text{type}} T_{Id(-, -)} \\
 \\ 
 \frac{\Gamma, x:S \vdash_\Sigma T : \text{type}}{\Gamma \vdash_\Sigma \Sigma_{x:S} T : \text{type}} T_\Sigma \quad \frac{\Gamma, x:S \vdash_\Sigma T : \text{type}}{\Gamma \vdash_\Sigma \Pi_{x:S} T : \text{type}} T_\Pi
 \end{array}}
 \end{array}$$

Figure 4: Types

Fig. 4 gives the formation rules for *types*. In context  $\Gamma$ , an application  $a \gamma_0$  of a type constructor  $a:(\Gamma_0)\text{type}$  to a substitution  $\gamma_0$  from  $\Gamma_0$  into  $\Gamma$ , means that  $\gamma_0$  provides a list of terms as arguments to  $a$ .

|  |   |
|--|---|
| $\frac{c:S \text{ in } \Sigma \quad \vdash_{\Sigma} \Gamma \text{ Ctx}}{\Gamma \vdash_{\Sigma} c : S} t_c$   | $\frac{\vdash_{\Sigma} \Gamma \text{ Ctx} \quad x:S \text{ in } \Gamma}{\Gamma \vdash_{\Sigma} x : S} t_x$                      |
| $\frac{\vdash_{\Sigma} \Gamma \text{ Ctx}}{\Gamma \vdash_{\Sigma} * : 1} t_*$  | $\frac{\Gamma \vdash_{\Sigma} s : S}{\Gamma \vdash_{\Sigma} refl(s) : Id(s, s)} t_{refl(-)}$                                    |
| $\frac{\Gamma \vdash_{\Sigma} s : S \quad \Gamma, x:S \vdash_{\Sigma} T : \text{type} \quad \Gamma \vdash_{\Sigma} t : T[x/s]}{\Gamma \vdash_{\Sigma} \langle s, t \rangle : \Sigma_{x:S} T} t_{\langle -, - \rangle}$ |   |
| $\frac{\Gamma \vdash_{\Sigma} u : \Sigma_{x:S} T}{\Gamma \vdash_{\Sigma} \pi_1(u) : S} t_{\pi_1}$  | $\frac{\Gamma \vdash_{\Sigma} u : \Sigma_{x:S} T}{\Gamma \vdash_{\Sigma} \pi_2(u) : T[x/\pi_1(s)]} t_{\pi_2}$                   |
| $\frac{\Gamma, x:S \vdash_{\Sigma} t : T}{\Gamma \vdash_{\Sigma} \lambda_{x:S} t : \Pi_{x:S} T} t_{\lambda}$   | $\frac{\Gamma \vdash_{\Sigma} f : \Pi_{x:S} T \quad \Gamma \vdash_{\Sigma} s : S}{\Gamma \vdash_{\Sigma} f s : T[x/s]} t_{app}$ |

Figure 5: Terms

Fig. 5 gives the *term* formation rules. For the case where only one variable is to be substituted in an expression  $e$  in context  $\Gamma, x:S$ , we define

$$e[x/s] := (\text{id}_{\Gamma}, x/s)(e).$$

We have the following subexpression property:  $\Gamma \vdash_{\Sigma} s : S$  implies  $\Gamma \vdash_{\Sigma} S : \text{type}$  implies  $\vdash_{\Sigma} \Gamma \text{ Ctx}$  implies  $\vdash_{\Sigma} \text{Sig}$ .

Fig. 6 gives the congruence and conversion rules for the *equality of terms*.  $\eta$ -conversion, reflexivity, symmetry, transitivity, and congruence rules for the other term constructors are omitted because they are derivable or admissible. In particular,  $\eta$ -conversion is implied by functional extensionality  $e_{funcext}$ . The rules have extra premises ensuring well-formedness of subexpressions, but these are elided for ease of reading, i.e., we assume that all terms occurring in Fig. 6 are well-formed without making that explicit in the rules.

Finally, Fig. 7 gives a simple axiomatization of the equality of types. Note that equality of types is decidable iff the equality of terms is.

Parallel to Def. 1, we obtain the following basic property of substitutions by a straightforward induction on derivations:

**Lemma 3.** *Assume  $\vdash_{\Sigma} \gamma : \Gamma \rightarrow \Gamma'$ . Then:*

$$\begin{array}{lll} \text{if} & \vdash_{\Sigma} \delta : \Delta \rightarrow \Gamma & \text{then} \quad \vdash_{\Sigma} \gamma \circ \delta : \Delta \rightarrow \Gamma', \\ \text{if} & \Gamma \vdash_{\Sigma} S : \text{type} & \text{then} \quad \Gamma' \vdash_{\Sigma} \gamma(S) : \text{type}, \\ \text{if} & \Gamma \vdash_{\Sigma} s : S & \text{then} \quad \Gamma' \vdash_{\Sigma} \gamma(s) : \gamma(S). \end{array}$$

$$\boxed{
\begin{array}{c}
\frac{\Gamma \vdash_{\Sigma} v : Id(s, s') \ e_{Id(-, -)}}{\Gamma \vdash_{\Sigma} s \equiv s'} \quad \frac{\Gamma \vdash_{\Sigma} v : Id(s, s') \quad \Gamma \vdash_{\Sigma} v' : Id(s, s') \ e_{id-uniq}}{\Gamma \vdash_{\Sigma} v \equiv v'} \\ \\ 
\frac{\Gamma \vdash_{\Sigma} s : 1 \ e_*}{\Gamma \vdash_{\Sigma} s \equiv *} \quad \frac{}{\Gamma \vdash_{\Sigma} \langle \pi_1(u), \pi_2(u) \rangle \equiv u} \ e_{\langle -, - \rangle} \\ \\ 
\frac{}{\Gamma \vdash_{\Sigma} \pi_1(\langle s, s' \rangle) \equiv s} \ e_{\pi_1} \quad \frac{}{\Gamma \vdash_{\Sigma} \pi_1(\langle s, s' \rangle) \equiv s'} \ e_{\pi_2} \\ \\ 
\frac{\Gamma \vdash_{\Sigma} f : \Pi_{x:S} T \quad \Gamma \vdash_{\Sigma} f' : \Pi_{x:S} T \quad \Gamma, y:S \vdash_{\Sigma} f \ y \equiv f' \ y}{\Gamma \vdash_{\Sigma} f \equiv f'} \ e_{funcext} \\ \\ 
\frac{\Gamma \vdash_{\Sigma} s : S \quad \Gamma \vdash_{\Sigma} s \equiv s' \quad \Gamma \vdash_{\Sigma} S \equiv S'}{\Gamma \vdash_{\Sigma} s' : S'} \ e_{typing}
\end{array}
}$$

Figure 6: Equality of Terms

$$\boxed{
\begin{array}{c}
\frac{\gamma = x_1/s_1, \dots, x_n/s_n \quad \gamma' = x_1/s'_1, \dots, x_n/s'_n \quad \Gamma \vdash_{\Sigma} s_i \equiv s'_i \text{ for } i = 1, \dots, n}{\Gamma \vdash_{\Sigma} a \ \gamma \equiv a \ \gamma'} E_a \\ \\ 
\frac{}{\Gamma \vdash_{\Sigma} 1 \equiv 1} E_1 \quad \frac{\Gamma \vdash_{\Sigma} s_1 \equiv s'_1 \quad \Gamma \vdash_{\Sigma} s_2 \equiv s'_2}{\Gamma \vdash_{\Sigma} Id(s_1, s_2) \equiv Id(s'_1, s'_2)} E_{Id(-, -)} \\ \\ 
\frac{\Gamma \vdash_{\Sigma} S \equiv S' \quad \Gamma, x:S \vdash_{\Sigma} T \equiv T'}{\Gamma \vdash_{\Sigma} \Sigma_{x:S} T \equiv \Sigma_{x:S'} T'} E_{\Sigma} \quad \frac{\Gamma \vdash_{\Sigma} S \equiv S' \quad \Gamma, x:S \vdash_{\Sigma} T \equiv T'}{\Gamma \vdash_{\Sigma} \Pi_{x:S} T \equiv \Pi_{x:S'} T'} E_{\Pi}
\end{array}
}$$

Figure 7: Equality of Types

### 3 Categorical Preliminaries

In this section, we repeat some well-known definitions and results about indexed sets and fibrations over posets (see, e.g., [Joh02]). We assume the basic notions of category theory (see, e.g., [Mac98]). We use a set-theoretical pairing function  $(a, b)$  and define tuples as left-associatively nested pairs, i.e.,  $(a_1, a_2, \dots, a_n)$  abbreviates  $(\dots(a_1, a_2), \dots, a_n)$ .

**Definition 4** (Indexed Sets).  $\mathcal{POSET}$  denotes the category of partially ordered sets. We treat posets as categories and write  $p \leq p'$  for the uniquely determined morphism  $p \rightarrow p'$ . If  $P$  is a poset,  $\mathcal{SET}^P$  denotes the category of functors  $P \rightarrow \mathcal{SET}$  and natural transformations. These functors are also called  $P$ -indexed sets.

We denote the constant  $P$ -indexed set that maps each  $p \in P$  to  $\{\emptyset\}$  by  $1_P$ . It is often convenient to replace an indexed set  $A$  over  $P$  with a poset formed from the disjoint union of all sets  $A(p)$  for  $p \in P$ . This is a special case of the category of elements, a construction due to Mac Lane ([MM92]) that is sometimes also called the Grothendieck construction.

**Definition 5** (Category of Elements). For an indexed set  $A$  over  $P$ , we define a poset  $\int_P A := \{(p, a) \mid p \in P, a \in A(p)\}$  with

$$(p, a) \leq (p', a') \quad \text{iff} \quad p \leq p' \text{ and } A(p \leq p')(a) = a'.$$

We also write  $\int A$  instead of  $\int_P A$  if  $P$  is clear from the context.

Using the category of elements, we can work with sets indexed by indexed sets: We write  $P|A$  if  $A$  is an indexed set over  $P$ , and  $P|A|B$  if additionally  $B$  is an indexed set over  $\int_P A$ , etc.

**Definition 6.** Assume  $P|A|B$ . We define an indexed set  $P|(A \times B)$  by

$$(A \times B)(p) = \{(a, b) \mid a \in A(p), b \in B(p, a)\}$$

and

$$(A \times B)(p \leq p') : (a, b) \mapsto (a', B((p, a) \leq (p', a'))(b)) \quad \text{for } a' = A(p \leq p')(a).$$

And we define a natural transformation  $\pi_B : A \times B \rightarrow A$  by

$$(\pi_B)_p : (a, b) \mapsto a.$$

The following definition introduces discrete opfibrations; for brevity, we will refer to them as “fibrations” in the sequel. Using the axiom of choice, these are necessarily split.

**Definition 7** (Fibrations). A *fibration* over a poset  $P$  is a functor  $f : Q \rightarrow P$  for a poset  $Q$  with the following property: For all  $p' \in P$  and  $q \in Q$  such that  $f(q) \leq p'$ , there is a unique  $q' \in Q$  such that  $q \leq q'$  and  $f(q') = p'$ . We call  $f$  canonical iff  $f$  is the first projection of  $Q = \int_P A$  for some  $P|A$ .

For every indexed set  $A$  over  $P$ , the first projection  $f_P : \int_P A \rightarrow P$  is a (canonical) fibration. Conversely, every fibration  $f : Q \rightarrow P$  defines an indexed set over  $P$  by mapping  $p \in P$  to its preimage  $f^{-1}(p) \subseteq Q$  and  $p \leq p'$  to the obvious function. This leads to a well-known equivalence of indexed sets and fibrations over  $P$ . If we only consider canonical fibrations, we obtain an isomorphism as follows.

**Lemma 8.** *If we restrict the objects of  $\mathcal{POSET}/P$  to be canonical fibrations and the morphisms to be (arbitrary) fibrations, we obtain the full subcategory  $\text{Fib}(P)$  of  $\mathcal{POSET}/P$ . There are isomorphisms*

$$F(-) : \mathcal{SET}^P \rightarrow \text{Fib}(P) \quad \text{and} \quad I(-) : \text{Fib}(P) \rightarrow \mathcal{SET}^P.$$

*Proof.* It is straightforward to show that  $\text{Fib}(P)$  is a full subcategory: The identity in  $\mathcal{POSET}$  and the composition of two fibrations are fibrations. Thus, it only remains to show that if  $f \circ \varphi = f'$  in  $\mathcal{POSET}$  where  $f$  and  $f'$  are fibrations and  $\varphi$  is a morphism in  $\mathcal{POSET}$ , then  $\varphi$  is a fibration as well. This is easy.

For  $A : P \rightarrow \mathcal{SET}$ , we define the fibration  $F(A) : \int_P A \rightarrow P$  by  $(p, a) \mapsto p$ . And for a natural transformation  $\eta : A \rightarrow A'$ , we define the fibration  $F(\eta) : \int_P A \rightarrow \int_P A'$  satisfying  $F(A) \circ F(\eta) = F(A')$  by  $(p, a) \mapsto (p, \eta_p(a))$ .

For  $f : Q \rightarrow P$ , we obtain an indexed set using the fact that  $f$  is canonical. More concretely, we define  $I(f)(p) := \{a \mid f(p, a) = p\}$  and  $I(f)(p \leq p') : a \mapsto a'$  where  $a'$  is the uniquely determined element such that  $(p, a) \leq (p', a') \in Q$ . And for a morphism  $\varphi$  between fibrations  $f : Q \rightarrow P$  and  $f' : Q' \rightarrow P$ , we define a natural transformation  $I(\varphi) : I(f) \rightarrow I(f')$  by  $I(\varphi)_p : a \mapsto a'$  where  $a'$  is such that  $\varphi(p, a) = (p, a')$ .

Then it is easy to compute that  $I$  and  $F$  are mutually inverse functors.  $\square$

**Definition 9** (Indexed Elements). Assume  $P|A$ . The  $P$ -indexed elements of  $A$  are given by

$$\text{Elem}(A) := \{(a_p \in A(p))_{p \in P} \mid a_{p'} = A(p \leq p')(a_p) \text{ whenever } p \leq p'\}.$$

Then the indexed elements of  $A$  are in bijection with the natural transformations  $1_P \rightarrow A$ . For  $a \in \text{Elem}(A)$ , we will write  $F(a)$  for the fibration  $P \rightarrow \int A$  mapping  $p$  to  $(p, a_p)$ .  $F(a)$  is a section of  $F(A)$ , and indexed elements are also called global sections.

*Example 10.* We exemplify the introduced notions by Fig. 8.  $P$  is a totally ordered set visualized as a horizontal line with two elements  $p_1 \leq p_2 \in P$ . For  $P|A$ ,  $\int A$  becomes a blob over  $P$ . The sets  $A(p_i)$  correspond to the vertical lines in  $\int A$ , and  $a_i \in A(p_i)$ . The action of  $A(p \leq p')$  and the poset structure of  $\int A$  are horizontal: If we assume  $A(p_1 \leq p_2) : a_1 \mapsto a_2$ , then  $(p_1, a_1) \leq (p_2, a_2)$  in  $\int A$ . Finally, the action of  $F(A)$  is vertical:  $F(A)$  maps  $(p_i, a_i)$  to  $p_i$ . Note that our intuitive visualization is not meant to indicate that the sets  $A(p_i)$  must be in bijection or that the mapping  $A(p_1 \leq p_2)$  must be injective or surjective.

Similarly, for  $P|A|B$ ,  $\int B$  becomes a three-dimensional blob over  $\int A$ . The sets  $B(p_i, a_i)$  correspond to the dotted lines. Again the action of  $B((p_1, a_1) \leq (p_2, a_2))$  and the poset structure of  $\int B$  are horizontal:

$$b_i \in B(p_i, a_i) \quad \text{and} \quad B((p_1, a_1) \leq (p_2, a_2)) : b_1 \mapsto b_2$$

and  $F(B)$  projects vertically from  $\int B$  to  $\int A$ .

Similarly, we have

$$(a_i, b_i) \in (A \ltimes B)(p_i) \quad \text{and} \quad (A \ltimes B)(p_1 \leq p_2) : (a_1, b_1) \mapsto (a_2, b_2)$$

Thus, the sets  $(A \ltimes B)(p_i)$  correspond to the two-dimensional gray areas. The sets  $\int_P(A \ltimes B)$  and  $\int_{f_P A} B$  are isomorphic, and their elements differ only in the bracketing:

$$(p_i, (a_i, b_i)) \in \int_P(A \ltimes B) \quad \text{and} \quad ((p_i, a_i), b_i) \in \int_{f_P A} B.$$

Up to this isomorphism, the projection  $F(A \ltimes B)$  is the composite  $F(A) \circ F(B)$ .

Indexed elements  $a \in \text{Elem}(A)$  are families  $(a_p)_{p \in P}$  and correspond to horizontal curves through  $\int A$  such that  $F(a)$  is a section of  $F(A)$ . Indexed elements of  $B$  correspond to two-dimensional vertical areas in  $\int B$  (intersecting each line parallel to the dotted lines exactly once), and indexed elements of  $A \ltimes B$  correspond to horizontal curves in  $\int B$  (intersecting each area parallel to the gray areas exactly once).

Finally the condition that indexed elements are natural transformations can be visualized as follows: The indexed elements  $a \in \text{Elem}(A)$  are exactly those horizontal curves that arise if a line is drawn from  $(p, a)$  to  $(p', a')$  whenever  $(p, a) \leq (p', a')$ . There may be multiple such curves going through a point  $(p, a)$ , but they must coincide to the right of  $(p, a)$ . Moreover,  $(p, a) \leq (p', a')$  holds iff  $(p, a)$  is to the left of  $(p', a')$  on the same curve. In particular, if  $P$  has a least element  $p_0$ , we obtain exactly one such curve for every element of  $A(p_0)$ .

*Example 11.* Let  $\text{Sign}$  be the set of well-formed signatures of MLTT (or of any other type theory for that matter).  $\text{Sign}$  is a poset under inclusion  $\subseteq$  of signatures. Let  $\text{Con}(\Sigma)$  be the set of well-formed contexts over  $\Sigma$ , and let  $\text{Con}(\Sigma \subseteq \Sigma') : \text{Con}(\Sigma) \hookrightarrow \text{Con}(\Sigma')$  be an inclusion. Then  $\text{Sign}|\text{Con}$ , and the tuple assigning the empty context to every signature is an example of an indexed element of  $\text{Con}$ .

$\int_{\text{Sign}} \text{Con}$  is the set of pairs  $(\Sigma, \Gamma)$  such that  $\vdash_\Sigma \Gamma \text{ Ctx}$ , and  $(\Sigma, \Gamma) \leq (\Sigma', \Gamma')$  iff  $\Sigma \subseteq \Sigma'$  and  $\Gamma = \Gamma'$ . Let  $\text{Typ}(\Sigma, \Gamma)$  be the set of types  $S$  such that  $\Gamma \vdash_\Sigma S : \text{type}$ .  $\text{Typ}$  becomes an indexed set  $\text{Sign}|\text{Con}|\text{Typ}$  by defining  $\text{Typ}((\Sigma, \Gamma) \leq (\Sigma', \Gamma'))$  to be an inclusion. The tuple assigning 1 to every pair  $(\Sigma, \Gamma)$  is an example of an indexed element of  $\text{Typ}$ .

We will use Lem. 8 frequently to switch between indexed sets and fibrations, as convenient. In particular, we will use the following two corollaries.

**Lemma 12.** *Assume  $P|A$ . Then*

$$\text{Elem}(A) \cong \text{Hom}_{\text{Fib}(P)}(\text{id}_P, F(A)) = \{f : P \rightarrow \int_P A \mid F(A) \circ f = \text{id}_P\}.$$

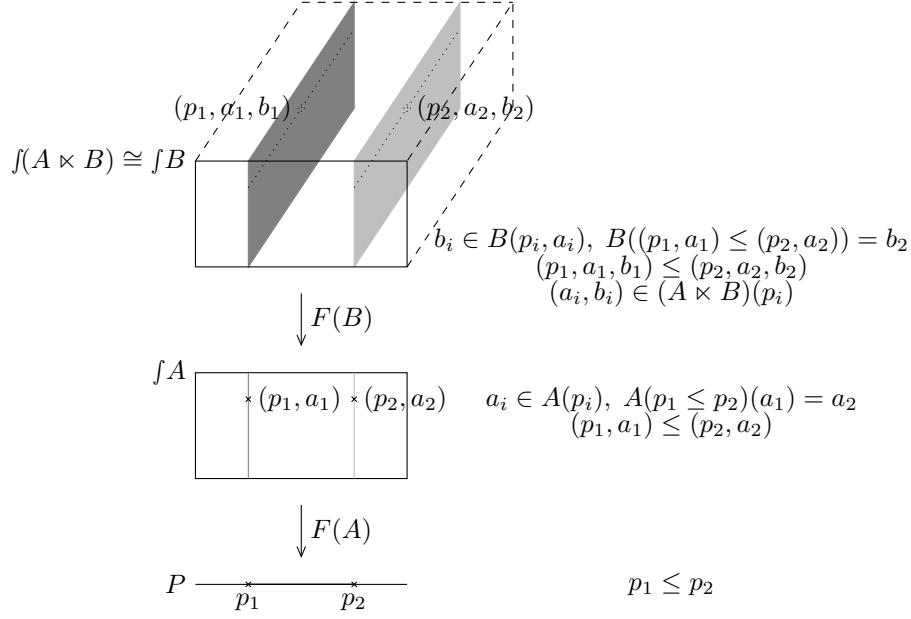


Figure 8: Indexed Sets and Fibrations

and

$$\mathcal{SET}^P/A \cong \mathcal{SET}^{fA}$$

*Proof.* Both claims follow from Lem. 8 by using  $\text{Elem}(A) \cong \text{Hom}_{\mathcal{SET}^P}(1_P, A)$  as well as  $\text{Fib}(P)/F(A) \cong \text{Fib}(f_P A)$ , respectively.  $\square$

Finally, as usual, we say that a category is *locally cartesian closed* (LCC) if it and all of its slice categories are cartesian closed (in particular, it has a terminal object). Then we have the following well-known result.

**Lemma 13.**  $\mathcal{SET}^P$  is LCC.

*Proof.* The terminal object is given by  $1_P$ . The product is taken pointwise:  $A \times B : p \mapsto A(p) \times B(p)$  and similarly for morphisms. The exponential object is given by:  $B^A : p \mapsto \text{Hom}_{\mathcal{SET}^{P^P}}(A^p, B^p)$  where  $A^p$  and  $B^p$  are as  $A$  and  $B$  but restricted to  $P^p := \{p' \in P \mid p \leq p'\}$ .  $B^A(p \leq p')$  maps a natural transformation, which is a family of mappings over  $P^p$ , to its restriction to  $P^{p'}$ . This proves that  $\mathcal{SET}^P$  and so also  $\text{Fib}(P)$  is cartesian closed for any  $P$ . By Lem. 12, we obtain the same for all slice categories.  $\square$

## 4 Operations on Indexed Sets

Because  $\mathcal{SET}^P$  is LCC, we know that it has pullbacks and that the pullback along a fixed natural transformation has left and right adjoints (see, e.g.,

[Joh02]). However, these functors are only unique up to isomorphism, and it is non-trivial to pick coherent choices for them.

**Pullbacks** Assume  $P|A_1$  and  $P|A_2$  and a natural transformation  $h : A_2 \rightarrow A_1$ . The pullback along  $h$  is a functor  $\mathcal{SET}^P/A_1 \rightarrow \mathcal{SET}^P/A_2$ . Using Lem. 12, we can avoid dealing with slice categories of  $\mathcal{SET}^P$  and instead give a functor

$$h^* : \mathcal{SET}^{fA_1} \rightarrow \mathcal{SET}^{fA_2},$$

which we also call the pullback along  $h$ . The functor  $h^*$  is given by precomposition:

**Definition 14.** Assume  $A_1$  and  $A_2$  indexed over  $P$ , and a natural transformation  $h : A_2 \rightarrow A_1$ . Then for  $B \in \mathcal{SET}^{fA_1}$ , we put

$$h^*B := B \circ F(h) \in \mathcal{SET}^{fA_2},$$

where, as in Lem. 8,  $F(h) : f_P A_2 \rightarrow f_P A_1$ . The action of  $h^*$  on morphisms is defined similarly by composing a natural transformation  $\beta : B \rightarrow B'$  with the functor  $F(h)$ :  $h^*\beta := \beta \circ F(h)$ . Finally, we define a natural transformation between  $P$ -indexed sets by

$$h \ltimes B : A_2 \ltimes h^*B \rightarrow A_1 \ltimes B, \quad (h \ltimes B)_p : (a_2, b) \mapsto (h_p(a_2), b).$$

The application of  $h \ltimes B$  is independent of  $B$ , which is only needed in the notation to determine the domain and codomain of  $h \ltimes B$ .

**Lemma 15** (Pullbacks). *In the situation of Def. 14, the following is a pullback in  $\mathcal{SET}^P$ .*

$$\begin{array}{ccc} A_2 \ltimes h^*B & \xrightarrow{h \ltimes B} & A_1 \ltimes B \\ \pi_{h^*B} \downarrow & & \downarrow \pi_B \\ A_2 & \xrightarrow{h} & A_1 \end{array}$$

Furthermore, we have the following coherence properties for every natural transformation  $g : A_3 \rightarrow A_2$ :

$$\begin{aligned} (\text{id}_{A_1})^*B &= B, & \text{id}_{A_1} \ltimes B &= \text{id}_{A_1 \ltimes B}, \\ (h \circ g)^*B &= g^*(h^*B), & (h \circ g) \ltimes B &= (h \ltimes B) \circ (g \ltimes h^*B). \end{aligned}$$

*Proof.* The following is a pullback in  $\mathcal{POSET}$ :

$$\begin{array}{ccc} f_A_2 \ltimes h^*B & \xrightarrow{F(h \ltimes B)} & f_A_1 \ltimes B & (p, (a_2, b)) \xrightarrow{F(h \ltimes B)} & (p, (h_p(a_2), b)) \\ \downarrow F(\pi_{h^*B}) & \quad \downarrow F(\pi_B) & & \downarrow F(\pi_{h^*B}) & \quad \downarrow F(\pi_B) \\ f_A_2 & \xrightarrow{F(h)} & f_A_1 & (p, a_2) \xrightarrow{F(h)} & (p, h_p(a_2)) \end{array}$$

If we turn this square into a cocone on  $P$  by adding the canonical projections  $F(A_2)$  and  $F(A_1)$ , it becomes a pullback in  $\text{Fib}(P)$ . Then the result follows by Lem. 8. The coherence properties can be verified by simple computations.  $\square$

Equivalently, using the terminology of [Pit00], we can say that for every  $P$  the tuple

$$(\mathcal{SET}^P, \mathcal{SET}^{\int A}, A \times B, \pi_B, h^*B, h \times B)$$

forms a type category (where  $A, B, h$  indicate arbitrary arguments). Then giving coherent adjoints to the pullback functor shows that this type category admits dependent sums and products.

**Adjoints** To interpret MLTT, the adjoints to  $h^*$ , where  $h : A_2 \rightarrow A_1$ , are only needed if  $h$  is a projection, i.e.,  $A_1 := A$ ,  $A_2 := A \times B$ , and  $h := \pi_B$  for some  $P|A|B$ . We only give adjoint functors for this special case because we use this restriction when defining the right adjoint. Thus, we give functors

$$\mathcal{L}_B, \mathcal{R}_B : \mathcal{SET}^{\int A \times B} \rightarrow \mathcal{SET}^{\int A} \text{ such that } \mathcal{L}_B \dashv \pi_B^* \dashv \mathcal{R}_B$$

in Def. 16 and 19, respectively. These functors will satisfy the coherence properties

$$g^*(\mathcal{L}_B C) = \mathcal{L}_{g^*B}(g \times B)^*C \quad \text{and} \quad g^*(\mathcal{R}_B C) = \mathcal{R}_{g^*B}(g \times B)^*C$$

for every  $g : A' \rightarrow A$ , which we prove in Lem. 17 and 20, respectively.

**Definition 16.** We define the functor  $\mathcal{L}_B$  as follows. For an object  $C$ , we put  $\mathcal{L}_B C := B \times (C \circ \text{assoc})$  where  $\text{assoc}$  maps elements  $((p, a), b) \in \int B$  to  $(p, (a, b)) \in \int A \times B$ ; and for a morphism, i.e., a natural transformation  $\eta : C \rightarrow C'$ , we put

$$(\mathcal{L}_B \eta)_{(p, a)} : (b, c) \mapsto (b, \eta_{(p, (a, b))}(c)) \quad \text{for } (p, a) \in \int A.$$

**Lemma 17** (Left Adjoint).  *$\mathcal{L}_B$  is left adjoint to  $\pi_B^*$ . Furthermore, for any natural transformation  $g : A' \rightarrow A$ , we have the following coherence property (the Beck-Chevalley condition)*

$$g^*(\mathcal{L}_B C) = \mathcal{L}_{g^*B}(g \times B)^*C.$$

*Proof.* It is easy to show that  $\mathcal{L}_B$  is isomorphic to composition along  $\pi_B$ , for which the adjointness is well-known. In particular, we have the following diagram in  $\mathcal{SET}^P$ :

$$\begin{array}{ccc}
(A \ltimes B) \ltimes C & \xleftarrow{\cong} & A \ltimes \mathcal{L}_B C \\
\pi_C \downarrow & & \swarrow \pi_{\mathcal{L}_B C} \\
A \ltimes B & & \\
\pi_B \downarrow & & \\
A & &
\end{array}$$

The coherence can be verified by direct computation.  $\square$

The right adjoint is more complicated. Intuitively,  $\mathcal{R}_B C$  must represent the dependent functions from  $B$  to  $C$ . The naive candidate for this is  $\text{Elem}(C) \cong \text{Hom}(1_{fB}, C)$  (i.e.,  $\text{Hom}(B, C)$  in the simply-typed case), but this is not a  $fA$ -indexed set. There is a well-known construction to remedy this, but we use a subtle modification to achieve coherence, i.e., the corresponding Beck-Chevalley condition. To do that, we need an auxiliary definition.

**Definition 18.** Assume  $P|A|B$ ,  $P|(A \ltimes B)|C$ , and an element  $x := (p, a) \in fA$ . Let  $A^x \in \mathcal{SET}^P$  and a natural transformation  $i^x : A^x \rightarrow A$  be given by

$$A^x(p') = \begin{cases} \{\emptyset\} & \text{if } p \leq p' \\ \emptyset & \text{otherwise} \end{cases} \quad i_p^x : \emptyset \mapsto A(p \leq p')(a).$$

Then we define indexed sets  $P|A^x|B^x$  and  $P|(A^x \ltimes B^x)|C^x$  by:

$$B^x := i^{x*} B, \quad C^x := (i^x \ltimes B)^* C$$

and put  $d^x := fA^x \ltimes B^x$  for the domain of  $C^x$ .

Note that  $A^x$  is the Yoneda embedding of  $p$  in  $\mathcal{SET}^P$ . The left diagram in Fig. 9 shows the involved  $P$ -indexed sets, the right one gives the actions of the natural transformations for an element  $p' \in P$  with  $p \leq p'$ . Below it will be crucial for coherence that  $B^x$  and  $C^x$  contain tuples in which  $a'$  is replaced with  $\emptyset$ .

**Definition 19.** Assume  $P|A|B$ . Then we define the functor  $\mathcal{R}_B : \mathcal{SET}^{fA \times B} \rightarrow \mathcal{SET}^{fA}$  as follows. Firstly, for an object  $C$ , we put for  $x \in fA$

$$(\mathcal{R}_B C)(x) := \text{Elem}(C^x).$$

In particular,  $f \in (\mathcal{R}_B C)(x)$  is a family  $(f_y)_{y \in d^x}$  with  $f_y \in C^x(y)$ . For  $x \leq x' \in fA$ , we have  $d^x \supseteq d^{x'}$  and put

$$(\mathcal{R}_B C)(x \leq x') : (f_y)_{y \in d^x} \mapsto (f_y)_{y \in d^{x'}}.$$

$$\begin{array}{ccccc}
& & (\mathbf{i}^x \ltimes B) \ltimes C & & \\
(A^x \ltimes B^x) \ltimes C^x & \longrightarrow & (A \ltimes B) \ltimes C & (\emptyset, b', c') \mapsto (a', b', c') & \\
\downarrow \pi_{C^x} & & \downarrow \pi_C & & \downarrow \\
A^x \ltimes B^x & \xrightarrow{\mathbf{i}^x \ltimes B} & A \ltimes B & (\emptyset, b') \mapsto (a', b') & \\
\downarrow \pi_{B^x} & & \downarrow \pi_B & & \downarrow \\
A^x & \xrightarrow{\mathbf{i}^x} & A & \emptyset \mapsto a' & \\
& & & \downarrow x := (p, a) & \\
& & & \downarrow a' := A(p \leq p')(a) &
\end{array}$$

Figure 9: The Situation of Def. 18

Secondly, for a morphism, i.e., a natural transformation  $\eta : C \rightarrow C'$ , we define  $\mathcal{R}_B \eta : \mathcal{R}_B C \rightarrow \mathcal{R}_B C'$  as follows: For  $x := (p, a) \in \int A$  and  $f \in (\mathcal{R}_B C)(x)$ , we define  $f' := (\mathcal{R}_B \eta)_x(f) \in (\mathcal{R}_B C')(x)$  by

$$f'_{(p', (\emptyset, b'))} := \eta_{(p', (\emptyset, b'))}(f_{(p', (\emptyset, b'))}) \quad \text{for } (p', (\emptyset, b')) \in d^x \text{ and } a' := A(p \leq p')(a).$$

**Lemma 20** (Right Adjoint).  $\mathcal{R}_B$  is right adjoint to  $\pi_B^*$ . Furthermore, for every natural transformation  $g : A' \rightarrow A$ , we have the following coherence property

$$g^*(\mathcal{R}_B C) = \mathcal{R}_{g^* B}(g \ltimes B)^* C.$$

*Proof.* Assume  $P|A|B$ ,  $P|A \ltimes B|C$ , and  $x = (p, a) \in \int A$ . Let  $y(x) \in \mathcal{SET}^{\int A}$  be the covariant representable functor of  $x$  mapping  $x' \in \int A$  to a singleton iff  $x \leq x'$  and to the empty set otherwise. Since we know the right adjoint exists, we can use the Yoneda lemma for covariant functors to derive sufficient and necessary constraints for  $\mathcal{R}_B$  to be a right adjoint:

$$\begin{aligned}
(\mathcal{R}_B C)(x) &\cong \text{Hom}_{\mathcal{SET}^{\int A}}(y(x), \mathcal{R}_B C) \cong \text{Hom}_{\mathcal{SET}^{\int A \ltimes B}}(\pi_B^* y(x), C) \\
&\cong \text{Hom}_{\text{Fib}(\int A \ltimes B)}(F(\pi_B^* y(x)), F(C)).
\end{aligned}$$

Let  $\mathbf{i}^x$  be as in Def. 18. Let  $\text{Fib}'(Q)$  be the category of (not necessarily canonical) fibrations on  $Q$ . Then it is easy to check that  $F(\mathbf{i}^x \ltimes B)$  seen as a fibration with domain  $d^x$  and  $F(\pi_B^* y(x))$  are isomorphic in  $\text{Fib}'(\int A \ltimes B)$ . (They are not isomorphic in  $\text{Fib}(\int B)$  because the former is not canonical and thus not an object of  $\text{Fib}(\int B)$ .) Using the fullness of  $\text{Fib}(Q)$ , we obtain

$$\begin{aligned}
(\mathcal{R}_B C)(x) &\cong \text{Hom}_{\text{Fib}'(\int A \ltimes B)}(F(\mathbf{i}^x \ltimes B), F(C)) \\
&= \{f : d^x \rightarrow \int C \mid F(C) \circ f = F(\mathbf{i}^x \ltimes B)\}.
\end{aligned}$$

And using the definition of  $C^x$  as a pullback, we obtain

$$(\mathcal{R}_B C)(x) \cong \{f : d^x \rightarrow \int C^x \mid F(C^x) \circ f = \text{id}_{d^x}\} \cong \text{Elem}(C^x).$$

And this is indeed how  $\mathcal{R}_B C$  is defined. The value of  $\mathcal{R}_B C$  on morphisms is verified similarly.

To show the coherence property, we assume  $P|A'$ ,  $g : A' \rightarrow A$ , and  $x' := (p, a') \in fA'$ . We abbreviate as follows:  $a := g_p(a')$ ,  $x := (p, a)$ ,  $B' := g^*B$ , and  $C' := (g \times B)^*C$ . Furthermore, we write  $i^{x'}$ ,  $A'^{x'}$ ,  $B'^{x'}$ , and  $C'^{x'}$  according to Def. 18. Note that  $A'^{x'} = A^x$ .

Now coherence requires  $g^*\mathcal{R}_B C = \mathcal{R}_{B'} C'$ . And that follows if we show that

$$B'^{x'} = B^x \quad \text{and} \quad C'^{x'} = C^x.$$

Using Lem. 15, this follows from  $g \circ i^{x'} = i^x$ , which is an equality between natural transformations from  $A^x = A'^{x'}$  to  $A$  in  $\mathcal{SET}^P$ . And to verify the latter, assume  $o \in P$ . The maps  $g_o \circ i_o^{x'}$  and  $i_o^x$  have domain  $\emptyset$  or  $\{\emptyset\}$ . In the former case, there is nothing to prove. In the latter case, put

$$a'_o := i_o^{x'}(\emptyset) = A'(p \leq o)(a') \quad \text{and} \quad a_o := i_o^x(\emptyset) = A(p \leq o)(a).$$

Then we need to show  $g_o(a'_o) = a_o$ . And that is indeed the case because of the naturality of  $g$  as indicated in

$$\begin{array}{ccc} & A'(p \leq o) & \\ a' \mapsto & \xrightarrow{\quad} & a'_o \\ \downarrow g_p & & \downarrow g_o \\ a \mapsto & \xrightarrow{\quad} & a_o \end{array}$$

□

*Example 21* (Continuing Ex. 11). The *Sign*-indexed set  $Con \times Typ$  maps every MLTT-signature  $\Sigma$  to the set of pairs  $(\Gamma, S)$  such that  $\Gamma \vdash_{\Sigma} S : \text{type}$ . The projection  $\pi_{Typ}$  is a natural transformation  $Con \times Typ \rightarrow Con$  such that  $(\pi_{Typ})_{\Sigma} : (\Gamma, S) \mapsto \Gamma$ .

We define  $Tm$  such that  $Sign|Con \times Typ|Tm$ : The set  $Tm(\Sigma, (\Gamma, S))$  contains the terms  $s$  such that  $\Gamma \vdash_{\Sigma} s : S$ .  $Tm((\Sigma, (\Gamma, S))) \leq (\Sigma', (\Gamma, S))$  is an inclusion.

Then we have  $Sign|Con|\mathcal{L}_{Typ} Tm$ , and  $\mathcal{L}_{Typ} Tm$  maps  $(\Sigma, \Gamma)$  to the set of pairs  $(S, s)$  such that  $\Gamma \vdash_{\Sigma} s : S$ .

To exemplify Def. 18, fix an element  $x = (\Sigma, \Gamma) \in f_{Sign} Con$ . Then we have  $i_{\Sigma'}^x(\emptyset) = \Gamma$  for every  $\Sigma \subseteq \Sigma'$ .  $Typ^x$  maps the pair  $(\Sigma', \emptyset)$  where  $\Sigma \subseteq \Sigma'$  to  $Typ(\Sigma', i_{\Sigma'}^x(\emptyset)) = Typ(\Sigma', \Gamma)$ . If  $S \in Typ(\Sigma', i_{\Sigma'}^x(\emptyset))$ , then  $Tm^x$  maps  $(\Sigma', (\emptyset, S))$  to the set  $Tm(\Sigma', (i_{\Sigma'}^x(\emptyset), S))$ .

Now we have  $Sign|Con|\mathcal{R}_{Typ} Tm$ , and  $\mathcal{R}_{Typ} Tm$  maps  $(\Sigma, \Gamma)$  to the set of indexed elements of  $Tm^x$ . Those are the families that assign to every  $(\Sigma', (\emptyset, S))$  a term  $s_{(\Sigma', (\emptyset, S))} \in Tm^x(\Sigma', (\emptyset, S)) = Tm(\Sigma', (\Gamma, S))$  such that  $s_{(\Sigma', (\emptyset, S))} = s_{(\Sigma'', (\emptyset, S))}$  whenever  $\Sigma' \subseteq \Sigma''$ .

Above, we called  $\text{Elem}(C)$  the naive candidate for the right adjoint, and indeed the adjointness implies  $\text{Elem}(\mathcal{R}_B C) \cong \text{Elem}(C)$ . We define the isomorphisms explicitly because we will use them later on:

**Lemma 22.** *Assume  $P|A|B$  and  $P|(A \times B)|C$ . For  $t \in \text{Elem}(C)$  and  $x := (p, a) \in \int A$ , let  $t^x \in \text{Elem}(C^x)$  be given by*

$$(t^x)_{(p', (\emptyset, b'))} = t_{(p', (a', b'))} \quad \text{where } a' := A(p \leq p')(a).$$

*And for  $f \in \text{Elem}(\mathcal{R}_B C)$  and  $x := (p, (a, b)) \in \int A \times B$ , we have  $f_{(p, a)} \in \text{Elem}(C^x)$ ; thus, we can put*

$$f^x := (f_{(p, a)})_{(p, (\emptyset, b))} \in C(p, (a, b)).$$

*Then the sets  $\text{Elem}(C)$  and  $\text{Elem}(\mathcal{R}_B C)$  are in bijection via*

$$\text{Elem}(C) \ni t \xrightarrow{\text{sp}(-)} (t^x)_{x \in \int A} \in \text{Elem}(\mathcal{R}_B C)$$

*and*

$$\text{Elem}(\mathcal{R}_B C) \ni f \xrightarrow{\text{am}(-)} (f^x)_{x \in \int A \times B} \in \text{Elem}(C).$$

*Proof.* This follows from the right adjointness by easy computations.  $\square$

Intuitively,  $\text{sp}(t)$  turns  $t \in \text{Elem}(C)$  into a  $\int A$ -indexed set by splitting it into components. And  $\text{am}(f)$  amalgamates such a tuple of components back together. Syntactically, these operations correspond to currying and uncurrying, respectively.

Then we need one last notation. For  $P|A$ , indexed elements  $a \in \text{Elem}(A)$  behave like mappings with domain  $P$ . We can precompose such indexed elements with fibrations  $f : Q \rightarrow P$  to obtain  $Q$ -indexed elements of  $\text{Elem}(A \circ f)$ .

**Definition 23.** Assume  $P|A$ ,  $f : Q \rightarrow P$ , and  $a \in \text{Elem}(A)$ .  $a * f \in \text{Elem}(A \circ f)$  is defined by:  $(a * f)_q := a_{f(q)}$  for  $q \in Q$ .

## 5 Semantics

Using the LCC structure developed in Sect. 4, the definition of the semantics is straightforward and well-known. To demonstrate its simplicity, we spell it out in an elementary way. The semantics is defined by induction on the derivations of the judgments listed in Fig. 2.

Firstly, for every signature  $\vdash \Sigma \text{ Sig}$ , we define models  $I$ , which provide interpretations  $\llbracket c \rrbracket^I$  and  $\llbracket a \rrbracket^I$  for all symbols declared in  $\Sigma$ . The models are Kripke-models, i.e., a  $\Sigma$ -model  $I$  is based on a poset  $P^I$  of worlds.

Secondly,  $I$  extends to an interpretation function  $\llbracket - \rrbracket^I$ , which interprets all  $\Sigma$ -expressions. We will omit the index  $I$  if no confusion is possible.  $\llbracket - \rrbracket$  is such that

- if  $\vdash_\Sigma \Gamma \text{ Ctx}$ , then  $\llbracket \Gamma \rrbracket$  is a poset (which has a canonical projection to  $P$ ),

- if  $\vdash_{\Sigma} \gamma : \Gamma \rightarrow \Gamma'$ , then  $\llbracket \gamma \rrbracket : \llbracket \Gamma' \rrbracket \rightarrow \llbracket \Gamma \rrbracket$  is a monotone function,
- if  $\Gamma \vdash_{\Sigma} S : \text{type}$ , then  $\llbracket \Gamma | S \rrbracket$  is an indexed set on  $\llbracket \Gamma \rrbracket$ ,
- if  $\Gamma \vdash_{\Sigma} s : S$ , then  $\llbracket \Gamma | s \rrbracket$  is an indexed element of  $\llbracket \Gamma | S \rrbracket$ .

Thirdly, the judgments  $\Gamma \vdash_{\Sigma} S \equiv S'$  and  $\Gamma \vdash_{\Sigma} s \equiv s'$  correspond to a soundness result, which we will prove in Sect. 7.

The poset  $P$  of worlds plays the same role as the various posets  $\llbracket \Gamma \rrbracket$  — it interprets the empty context. In this way,  $P$  can be regarded as interpreting an implicit or relative context. This is in keeping with the practice of type theory (and category theory), according to which closed expressions may be considered relative to some fixed but unspecified context (respectively, base category).

For a typed term  $\Gamma \vdash_{\Sigma} s : S$ , both  $\llbracket \Gamma | s \rrbracket$  and  $\llbracket \Gamma | S \rrbracket$  are indexed over  $\llbracket \Gamma \rrbracket$ . If  $\Gamma = x_1 : S_1, \dots, x_n : S_n$ , an element of  $\llbracket \Gamma \rrbracket$  has the form  $(p, (a_1, \dots, a_n))$  where  $p \in P$  and  $a_i \in [x_1 : S_1, \dots, x_{i-1} : S_{i-1} | S_i](p, (a_1, \dots, a_{i-1}))$ . Intuitively,  $a_i$  is an assignment to the variable  $x_i$  in world  $p$ . And if an assignment  $(p, \alpha)$  is given, the interpretations of  $s$  and  $S$  satisfy  $\llbracket \Gamma | s \rrbracket_{(p, \alpha)} \in \llbracket \Gamma | S \rrbracket(p, \alpha)$ . This is illustrated in the left diagram in Fig. 10.

If  $\gamma$  is a substitution  $\Gamma \rightarrow \Gamma'$ , then  $\llbracket \gamma \rrbracket$  maps assignments  $(p, \alpha') \in \llbracket \Gamma' \rrbracket$  to assignments  $(p, \alpha) \in \llbracket \Gamma \rrbracket$ . And a substitution in types and terms is interpreted by pullback, i.e., composition. This is illustrated in the right diagram in Fig. 10, whose commutativity expresses the coherence. We will state this more precisely in Sect. 6.

Sum types are interpreted naturally as the dependent sum of indexed sets given by the left adjoint. And pairing and projections have their natural semantics. Product types are interpreted as exponentials using the right adjoint. A  $\lambda$ -abstraction  $\lambda_{x:S} t$  is interpreted by first interpreting  $t$  and then splitting it as in Lem. 22. And an application  $f s$  is interpreted by amalgamating the interpretation of  $f$  as in Lem. 22 and using the composition from Def. 23.

$$\begin{array}{ccccc}
& & \llbracket \Gamma | S \rrbracket & & \\
& \nearrow F(\llbracket \Gamma | s \rrbracket) & \downarrow F(\llbracket \Gamma | S \rrbracket) & \nearrow \llbracket \gamma \rrbracket & \\
\llbracket \Gamma \rrbracket & \xrightarrow{\text{id}} & \llbracket \Gamma \rrbracket & \llbracket \Gamma' \rrbracket & \searrow \llbracket \Gamma | S \rrbracket \\
& & & \llbracket \Gamma' | \gamma(S) \rrbracket & \\
& & & \searrow & \\
& & & & \mathcal{SET}
\end{array}$$

Figure 10: Semantics of Terms, Types, and Substitution

**Definition 24** (Models). For a signature  $\Sigma$ ,  $\Sigma$ -models are defined as follows:

- A model  $I$  for the empty signature  $\cdot$  is a poset  $P^I$ .
- A model  $I$  for the signature  $\Sigma, c : S$  consists of a  $\Sigma$ -model  $I_{\Sigma}$  and an indexed element  $\llbracket c \rrbracket^I \in \text{Elem}(\llbracket \cdot | S \rrbracket^{I_{\Sigma}})$ .

- A model  $I$  for the signature  $\Sigma, a:(\Gamma_0)\text{type}$  consists of a  $\Sigma$ -model  $I_\Sigma$  and an indexed set  $\llbracket a \rrbracket^I$  over  $\llbracket \Gamma_0 \rrbracket^{I_\Sigma}$ .

**Definition 25** (Model Extension). The extension of a model is defined by induction on the typing derivations. Therefore, we can assume in each case that all occurring expressions are well-formed. For example in the case for  $\llbracket \Gamma | f s \rrbracket$ ,  $f$  has type  $\Pi_{x:S} T$  and  $s$  has type  $S$ .

- Contexts: The elements of the poset  $\llbracket x_1 : S_1, \dots, x_n : S_n \rrbracket$  are the tuples  $(p, (a_1, \dots, a_n))$  such that

$$\begin{aligned} p &\in P \\ a_1 &\in \llbracket \cdot | S_1 \rrbracket(p, \emptyset) \\ &\vdots \\ a_n &\in \llbracket x_1 : S_1, \dots, x_{n-1} : S_{n-1} | S_n \rrbracket(p, (a_1, \dots, a_{n-1})) \end{aligned}$$

In particular  $\llbracket \cdot \rrbracket = P \times \{\emptyset\}$ . The ordering of this poset is inherited from the  $n$ -times iterated category of elements, to which it is canonically isomorphic. The first projection from  $\llbracket \Gamma \rrbracket$  is a canonical fibration, and we write  $I(\llbracket \Gamma \rrbracket)$  for the corresponding indexed set.

- Substitutions  $\gamma = x_1/s_1, \dots, x_n/s_n$  from  $\Gamma$  to  $\Gamma'$ :

$$\llbracket \gamma \rrbracket : (p, \alpha') \mapsto (p, (\llbracket \Gamma' | s_1 \rrbracket_{(p, \alpha')}, \dots, \llbracket \Gamma' | s_n \rrbracket_{(p, \alpha')})) \quad \text{for } (p, \alpha') \in \llbracket \Gamma' \rrbracket$$

We write  $I(\llbracket \gamma \rrbracket)$  for the induced natural transformation  $I(\llbracket \Gamma' \rrbracket) \rightarrow I(\llbracket \Gamma \rrbracket)$ .

- Basic types:

$$\llbracket \Gamma | a \gamma_0 \rrbracket := \llbracket a \rrbracket \circ \llbracket \gamma_0 \rrbracket$$

- Complex types:

$$\begin{aligned} \llbracket \Gamma | 1 \rrbracket(p, \alpha) &:= \{\emptyset\} \\ \llbracket \Gamma | Id(s, s') \rrbracket(p, \alpha) &:= \begin{cases} \{\emptyset\} & \text{if } \llbracket \Gamma | s \rrbracket_{(p, \alpha)} = \llbracket \Gamma | s' \rrbracket_{(p, \alpha)} \\ \emptyset & \text{otherwise} \end{cases} \\ \llbracket \Gamma | \Sigma_{x:S} T \rrbracket &:= \mathcal{L}_{\llbracket \Gamma | S \rrbracket} \llbracket \Gamma, x:S | T \rrbracket \\ \llbracket \Gamma | \Pi_{x:S} T \rrbracket &:= \mathcal{R}_{\llbracket \Gamma | S \rrbracket} \llbracket \Gamma, x:S | T \rrbracket \end{aligned}$$

$\llbracket \Gamma | 1 \rrbracket$  and  $\llbracket \Gamma | Id(s, s') \rrbracket$  are only specified for objects; their extension to morphisms is uniquely determined.

- Basic terms:

$$\llbracket \Gamma | c \rrbracket_{(p, \alpha)} := \llbracket c \rrbracket_p, \quad \llbracket x_1 : S_1, \dots, x_n : S_n | x_i \rrbracket_{(p, (a_1, \dots, a_n))} := a_i$$

- Complex terms:

$$\begin{aligned}
\llbracket \Gamma | * \rrbracket_{(p,\alpha)} &:= \emptyset \\
\llbracket \Gamma | refl(s) \rrbracket_{(p,\alpha)} &:= \emptyset \\
\llbracket \Gamma | \langle s, s' \rangle \rrbracket_{(p,\alpha)} &:= (\llbracket \Gamma | s \rrbracket_{(p,\alpha)}, \llbracket \Gamma | s' \rrbracket_{(p,\alpha)}) \\
\llbracket \Gamma | \pi_i(u) \rrbracket_{(p,\alpha)} &:= a_i \quad \text{where } \llbracket \Gamma | u \rrbracket_{(p,\alpha)} = (a_1, a_2) \\
\llbracket \Gamma | \lambda_{x:S} t \rrbracket &:= \text{sp}(\llbracket \Gamma, x:S | t \rrbracket) \\
\llbracket \Gamma | f s \rrbracket &:= \text{am}(\llbracket \Gamma | f \rrbracket) * (\text{assoc} \circ F(\llbracket \Gamma | s \rrbracket))
\end{aligned}$$

Here *assoc* maps  $((p, \alpha), a)$  to  $(p, (\alpha, a))$ .

Since the same expression may have more than one well-formedness derivation, the well-definedness of Def. 25 must be proved in a joint induction with the proof of Thm. 30 below (see also [Str91]). And because of the use of substitution, e.g., for application of function terms, the induction must be intertwined with the proof of Thm. 27 as well.

*Example 26* (Continuing Ex. 2). A model of the signature *Cat* over an indexing poset  $P$  is the same thing as a functor from  $P$  into  $\mathcal{CAT}$ , the category of (small) categories. In more detail, assume a poset  $P$  and a functor  $F : P \rightarrow \mathcal{CAT}$ . Then we obtain a model of the signature *Cat* as follows:

- The underlying poset is  $P$ .
- $\llbracket Ob \rrbracket$  is the indexed set over  $P$  mapping
  - every  $p \in P$  to the set of objects of  $F(p)$ ,
  - every morphism  $p \leq p'$  to the object-part of  $F(p \leq p')$ .
- $\llbracket x : Ob, y : Ob \rrbracket$  is a poset containing tuples  $(p, (a, b))$  for  $a, b \in F(p)$ . We obtain  $(p, (a, b)) \leq (p', (a', b'))$  iff  $p \leq p'$  and  $a' = F(p \leq p')(a)$  and  $b' = F(p \leq p')(b)$ . Then  $\llbracket Mor \rrbracket$  is the indexed set over  $\llbracket x : Ob, y : Ob \rrbracket$  mapping
  - every  $(p, (a, b))$  to the set  $\text{Hom}_{F(p)}(a, b)$ ,
  - every  $(p, (a, b)) \leq (p', (a', b'))$  to the morphism part of  $F(p \leq p')$  restricted to a map from  $\text{Hom}_{F(p)}(a, b)$  to  $\text{Hom}_{F(p')}(a', b')$ .
- Next we define  $\llbracket id \rrbracket \in \text{Elem}(\llbracket \cdot | \prod_{x:Ob} Mor x x \rrbracket)$  as  $\text{sp}(e)$  (using Lem. 22) where  $e \in \text{Elem}(\llbracket x:Ob | Mor x x \rrbracket)$  is defined as follows.  $\llbracket x:Ob | Mor x x \rrbracket$  maps  $(p, a)$  for  $a \in \llbracket \cdot | Ob \rrbracket(p)$  to the set  $\text{Hom}_{F(p)}(a, a)$ , and we put  $e_{(p,a)} := \text{id}_a$ .

Because  $F$  is a functor, we have

$$\llbracket x:Ob | Mor x x \rrbracket((p, a) \leq (p', a'))(\text{id}_a) = \text{id}_{a'}.$$

Therefore,  $e$  is indeed an indexed element.

- *comp* is interpreted as composition in  $F(p)$  in the same manner as *id* applying Lem. 22 five times.

- The interpretations of the constants representing axioms such as *neutr* are uniquely determined. And they exist because all  $F(p)$  are categories.

## 6 Substitution Lemma

Parallel to Lem. 3, we obtain the following central result about the semantics of substitutions. It expresses the coherence of our models.

**Theorem 27** (Substitution). *Assume  $\vdash_{\Sigma} \gamma : \Gamma \rightarrow \Gamma'$ . Then:*

$$\begin{array}{lll} \text{if } \vdash_{\Sigma} \delta : \Delta \rightarrow \Gamma & \text{then} & \llbracket \gamma \circ \delta \rrbracket = \llbracket \delta \rrbracket \circ \llbracket \gamma \rrbracket, \\ \text{if } \Gamma \vdash_{\Sigma} S : \text{type} & \text{then} & \llbracket \Gamma' | \gamma(S) \rrbracket = \llbracket \Gamma | S \rrbracket \circ \llbracket \gamma \rrbracket, \\ \text{if } \Gamma \vdash_{\Sigma} s : S & \text{then} & \llbracket \Gamma' | \gamma(s) \rrbracket = \llbracket \Gamma | s \rrbracket * \llbracket \gamma \rrbracket. \end{array}$$

Before we give the proof of Thm. 27, we establish some auxiliary results:

**Lemma 28.** *Assume  $\vdash_{\Sigma} \gamma : \Gamma \rightarrow \Gamma'$  and  $\Gamma \vdash_{\Sigma} S : \text{type}$  and thus also*

$$\vdash_{\Sigma} \gamma, x/x : \Gamma, x:S \rightarrow \Gamma', x:\gamma(S).$$

*Furthermore, assume the induction hypothesis of Thm. 27 for the involved expressions. Then we have:*

$$\llbracket \gamma, x/x \rrbracket = F(I(\llbracket \gamma \rrbracket) \ltimes \llbracket \Gamma | S \rrbracket).$$

*Proof.* This follows by direct computation. □

**Lemma 29.** *Assume  $P|A|B$ ,  $P|A \ltimes B|C$ ,  $P|A'$ , a natural transformation  $g : A' \rightarrow A$ , and  $t \in \text{Elem}(C)$ . Then for  $x' \in fA'$ :*

$$\text{sp}(t * F(g \ltimes B))_{x'} = \text{sp}(t)_{F(g)(x')}.$$

*Proof.* This follows by direct computation. □

*Proof of Thm. 27.* The proofs of all subtheorems are intertwined in an induction on the typing derivations; in addition, the induction is intertwined with the proof of Thm. 30.

The case of an empty substitution  $\delta$  is trivial. For the remaining cases, assume  $\delta = x_1/s_1, \dots, x_n/s_n$  and  $(p, \alpha') \in \llbracket \Gamma' \rrbracket$ . Then applying the composition of substitutions, the semantics of substitutions, the induction hypothesis for terms, and the semantics of substitutions, respectively, yields:

$$\begin{aligned} \llbracket \gamma \circ \delta \rrbracket(p, \alpha') &= \llbracket x_1/\gamma(s_1), \dots, x_n/\gamma(s_n) \rrbracket(p, \alpha') = (p, (\llbracket \Gamma' | \gamma(s_i) \rrbracket)_{(p, \alpha')} )_{i=1, \dots, n}) \\ &= (p, (\llbracket \Gamma | s_i \rrbracket)_{\llbracket \gamma \rrbracket(p, \alpha')})_{i=1, \dots, n}) = (\llbracket \delta \rrbracket \circ \llbracket \gamma \rrbracket)(p, \alpha') \end{aligned}$$

The cases for *types* are as follows:

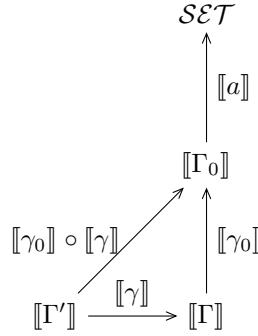
- $a \gamma_0$ : Using the definition of substitution and the semantics of application, we obtain:

$$\llbracket \Gamma' | \gamma(a \gamma_0) \rrbracket = \llbracket \Gamma' | a (\gamma_0 \circ \gamma) \rrbracket = \llbracket a \rrbracket \circ \llbracket \gamma_0 \circ \gamma \rrbracket$$

And similarly we obtain:

$$\llbracket \Gamma | a \gamma_0 \rrbracket = \llbracket a \rrbracket \circ \llbracket \gamma_0 \rrbracket$$

Then the needed equality follows from the induction hypothesis for  $\gamma_0$ .



- 1: Trivial.
- $Id(s, s')$ : This follows directly from the induction hypothesis for  $s$  and  $s'$ .
- $\Sigma_{x:S} T$ : This follows directly by combining the induction hypothesis as well as Lem. 17 and 28.
- $\Pi_{x:S} T$ : This follows directly by combining the induction hypothesis as well as Lem. 20 and 28.

For the cases of a term  $s$ , let us assume a fixed  $(p, \alpha') \in \llbracket \Gamma' \rrbracket$  and  $(p, \alpha) := \llbracket \gamma \rrbracket(p, \alpha')$ . Then we need to show

$$\llbracket \Gamma' | \gamma(s) \rrbracket_{(p, \alpha')} = \llbracket \Gamma | s \rrbracket_{(p, \alpha)}.$$

- $c$ : Clear because  $\gamma(c) = c$ .
- $x$ : Assume  $x$  occurs in position  $i$  in  $\Gamma$ , and let  $x/s$  be in  $\gamma$ . Further, assume  $\alpha' = (a'_1, \dots, a'_n)$  and  $\alpha = (a_1, \dots, a_n)$ . Then by the properties of substitutions:  $\llbracket \Gamma' | \gamma(x) \rrbracket_{(p, \alpha')} = \llbracket \Gamma' | s \rrbracket_{(p, \alpha')} = a_i$ . And that is equal to  $\llbracket \Gamma | x \rrbracket_{(p, \alpha)}$ .
- $refl(s)$ : Trivial.
- $*$ : Trivial.

- $\langle s, s' \rangle$ : Because  $\gamma(\langle s, s' \rangle) = \langle \gamma(s), \gamma(s') \rangle$ , this case follows immediately from the induction hypothesis.
- $\pi_i(u)$  for  $i = 1, 2$ : Because  $\gamma(\pi_i(s)) = \pi_i(\gamma(s))$ , this case follows immediately from the induction hypothesis.
- $\lambda_{x:S} t$ : By the definition of substitution, the semantics of  $\lambda$ -abstraction, the induction hypothesis, and Lem. 28, respectively, we obtain:

$$\begin{aligned} \llbracket \Gamma' | \gamma(\lambda_{x:S} t) \rrbracket &= \llbracket \Gamma' | \lambda_{x:\gamma(S)} \gamma^x(t) \rrbracket = \text{sp}(\llbracket \Gamma', x:\gamma(S) | \gamma^x(t) \rrbracket) \\ &= \text{sp}(\llbracket \Gamma, x:S | t \rrbracket * \llbracket \gamma, x/x \rrbracket) \\ &= \text{sp}(\llbracket \Gamma, x:S | t \rrbracket * F(I(\llbracket \gamma \rrbracket) \ltimes \llbracket \Gamma | S \rrbracket)). \end{aligned}$$

Furthermore, we have  $\llbracket \Gamma | \lambda_{x:S} t \rrbracket = \text{sp}(\llbracket \Gamma, x:S | t \rrbracket)$ . Then the result follows by using Lem. 29 and  $F(I(\llbracket \gamma \rrbracket)) = \llbracket \gamma \rrbracket$ .

- $f s$ : We evaluate both sides of the needed equation. Firstly, on the left-hand side, we obtain by the definition of substitution, the semantics of application, and the induction hypothesis, respectively:

$$\begin{aligned} \llbracket \Gamma' | \gamma(f s) \rrbracket &= \llbracket \Gamma' | \gamma(f) \gamma(s) \rrbracket = \text{am}(\llbracket \Gamma' | \gamma(f) \rrbracket) * (\text{assoc} \circ F(\llbracket \Gamma' | \gamma(s) \rrbracket)) \\ &= \text{am}(\llbracket \Gamma | f \rrbracket * \llbracket \gamma \rrbracket) * (\text{assoc} \circ F(\llbracket \Gamma | s \rrbracket * \llbracket \gamma \rrbracket)). \end{aligned}$$

To compute the value at  $(p, \alpha')$  of this indexed element, we first compute  $(\llbracket \Gamma | s \rrbracket * \llbracket \gamma \rrbracket)_{(p, \alpha')}$ , say we obtain  $b$ . Then we can compute  $\text{am}(\llbracket \Gamma | f \rrbracket * \llbracket \gamma \rrbracket)_{(p, (\alpha', b))}$ . Using the notation from Lem. 22, the left-hand side evaluates to

$$(\llbracket \Gamma | f \rrbracket * \llbracket \gamma \rrbracket)^{(p, (\alpha', b))} = (\llbracket \Gamma | f \rrbracket)_{(p, \alpha)}.$$

Secondly, on the right-hand side, we have by the semantics of application:

$$\llbracket \Gamma | f s \rrbracket = \text{am}(\llbracket \Gamma | f \rrbracket) * (\text{assoc} \circ F(\llbracket \Gamma | s \rrbracket)).$$

When computing the value at  $(p, \alpha)$  of this indexed element, we obtain in a first step  $\text{am}(\llbracket \Gamma | f \rrbracket)_{(p, (\alpha, b))}$ . And evaluating further, this yields  $(\llbracket \Gamma | f \rrbracket)_{(p, \alpha)}$ .

Thus, the equality holds as needed. □

## 7 Soundness

We have already mentioned the soundness result, which states that the interpretation takes the syntactic judgments for equality of terms and types to corresponding semantic judgments:

**Theorem 30** (Soundness). *Assume a signature  $\Sigma$ , and a context  $\Gamma$ . If  $\Gamma \vdash_{\Sigma} S \equiv S'$  for two well-formed types  $S, S'$ , then in every  $\Sigma$ -model:*

$$\llbracket \Gamma | S \rrbracket = \llbracket \Gamma | S' \rrbracket \in \mathcal{SET}^{\llbracket \Gamma \rrbracket}.$$

*And if  $\Gamma \vdash_{\Sigma} s \equiv s'$  for two well-formed terms  $s, s'$  of type  $S$ , then in every  $\Sigma$ -model:*

$$\llbracket \Gamma | s \rrbracket = \llbracket \Gamma | s' \rrbracket \in \text{Elem}(\llbracket \Gamma | S \rrbracket).$$

*Proof.* The soundness is proved by induction over all derivations; the induction is intertwined with the proof of Thm. 27. An instructive example is the rule  $e_{typing}$ . Its soundness states the following: If  $\llbracket \Gamma | s \rrbracket \in \text{Elem}(\llbracket \Gamma | S \rrbracket)$  and  $\llbracket \Gamma | s \rrbracket = \llbracket \Gamma | s' \rrbracket$  and  $\llbracket \Gamma | S \rrbracket = \llbracket \Gamma | S' \rrbracket$ , then also  $\llbracket \Gamma | s' \rrbracket \in \text{Elem}(\llbracket \Gamma | S' \rrbracket)$ . And this clearly holds.

Among the remaining rules for terms, the soundness of some rules is an immediate consequence of the semantics. These are: all rules from Fig. 5 except for  $t_{\lambda}$  and  $t_{app}$ , and from Fig. 6 the rules  $e_{Id(-,-)}$ ,  $e_{id-uniq}$ ,  $e_*$ ,  $e_{\langle -, - \rangle}$ ,  $e_{\pi_1}$ ,  $e_{\pi_2}$ , and  $e_{app}$ .

The soundness of the rules  $t_{\lambda}$  and  $t_{app}$  follows by applying the semantics and Lem. 22. That leaves the rules  $e_{\beta}$  and  $e_{funcext}$ , the soundness of which we will prove in detail.

For  $e_{\beta}$ , we interpret  $(\lambda_{x:S} t) s$  by applying the definition:

$$\begin{aligned} \llbracket \Gamma | (\lambda_{x:S} t) s \rrbracket &= \text{am}(\llbracket \Gamma | \lambda_{x:S} t \rrbracket) * (\text{assoc} \circ F(\llbracket \Gamma | s \rrbracket)) \\ &= \text{am}(\text{sp}(\llbracket \Gamma, x:S | t \rrbracket)) * (\text{assoc} \circ F(\llbracket \Gamma | s \rrbracket)) \end{aligned}$$

$\text{am}(\text{sp}(\llbracket \Gamma, x:S | t \rrbracket))$  is equal to  $\llbracket \Gamma, x:S | t \rrbracket$  by Lem. 22. Furthermore, we have  $t[x/s] = \gamma(t)$  where  $\gamma = \text{id}_{\Gamma}, x/s$  is a substitution from  $\Gamma, x:S$  to  $\Gamma$ . And interpreting  $\gamma$  yields  $\llbracket \gamma \rrbracket(p, \alpha) = (p, (\alpha, \llbracket \Gamma | s \rrbracket_{(p, \alpha)}))$ , i.e.,  $\llbracket \gamma \rrbracket = \text{assoc} \circ F(\llbracket \Gamma | s \rrbracket)$ . Therefore, using Thm. 27 for terms yields

$$\llbracket \Gamma | t[x/s] \rrbracket = \llbracket \Gamma, x:S | t \rrbracket * (\text{assoc} \circ F(\llbracket \Gamma | s \rrbracket)),$$

which concludes the soundness proof for  $e_{\beta}$ .

To understand the soundness of  $e_{funcext}$ , let us look at the interpretations of  $f$  in the contexts  $\Gamma$  and  $\Gamma, y:S$ :

$$\text{am}(\llbracket \Gamma | f \rrbracket) \in \text{Elem}(\llbracket \Gamma, x:S | T \rrbracket), \quad \text{am}(\llbracket \Gamma, y:S | f \rrbracket) \in \text{Elem}(\llbracket \Gamma, y:S, x:S | T \rrbracket).$$

Let  $\gamma$  be the inclusion substitution from  $\Gamma$  to  $\Gamma, y:S$ . Then  $\llbracket \gamma \rrbracket$  is the projection  $\llbracket \Gamma, y:S \rrbracket \rightarrow \llbracket \Gamma \rrbracket$  mapping elements  $(p, (\alpha, a))$  to  $(p, \alpha)$ . Applying Thm. 27 yields for arbitrary  $(p, \alpha) \in \llbracket \Gamma \rrbracket$  and  $a', a \in \llbracket \Gamma | S \rrbracket(p, \alpha)$ :

$$\text{am}(\llbracket \Gamma, y:S | f \rrbracket)_{(p, (\alpha, a'))} = \text{am}(\llbracket \Gamma | f \rrbracket)_{(p, (\alpha, a))}.$$

And we have

$$[\Gamma, y:S|y]_{(p,(\alpha,a'))} = a', \quad \text{and} \quad F([\Gamma, y:S|y])(p, (\alpha, a')) = (p, (\alpha, a'), a').$$

Putting these together yields

$$\begin{aligned} [\Gamma, y:S|f\ y]_{(p,(\alpha,a'))} &= (\text{am}([\Gamma, y:S|f]) * (\text{assoc} \circ F([\Gamma, y:S|y])))_{(p,(\alpha,a'))} \\ &= \text{am}([\Gamma, y:S|f])_{(p,(\alpha,a',a'))} = \text{am}([\Gamma|f])_{(p,(\alpha,a'))} \end{aligned}$$

Therefore, the induction hypothesis applied to  $\Gamma, y:S \vdash_\Sigma f\ y \equiv f'\ y$  yields

$$\text{am}([\Gamma|f]) = \text{am}([\Gamma|f']).$$

And then Lem. 22 yields

$$[\Gamma|f] = [\Gamma|f']$$

concluding the soundness proof for  $e_{funcext}$ .

Regarding the rules for types in Fig. 4 and Fig. 7, the soundness proofs are straightforward.  $\square$

## 8 Completeness

According to the propositions-as-types interpretation — also known as the Curry-Howard correspondence — a type  $S$  holds in a model if its interpretation  $[S]$  is inhabited, i.e., the indexed set  $[S]$  has an indexed element. A type is valid if it holds in all models. Then soundness implies: If there is a term  $s$  of type  $S$  in context  $\Gamma$ , then in every  $\Sigma$ -model there is an indexed element of  $[\Gamma|S]$ , namely  $[\Gamma|s]$ . The converse is completeness: A type that has an indexed element in every model is inhabited. Observe that the presence of (extensional) identity types then implies also the completeness of the equational term calculus because two terms are equal iff the corresponding identity type is inhabited.

The basic idea of the proof of completeness is to build the syntactic category, and then to construct a model out of it using categorical embedding theorems.

**Definition 31.** A functor  $F : \mathcal{C} \rightarrow \mathcal{D}$  is called LCC if  $\mathcal{C}$  is LCC and if  $F$  preserves that structure, i.e.,  $F$  maps terminal object, products and exponentials in all slices  $\mathcal{C}/A$  to corresponding structures in  $\mathcal{D}/F(A)$ . An LCC functor is called an LCC embedding if it is injective on objects, full, and faithful.

We make use of a theorem from topos theory due to Butz and Moerdijk ([BM99]) to establish the following central lemma.

**Lemma 32.** *For every LCC category  $\mathcal{C}$ , there is a poset  $P$  and an LCC embedding  $E : \mathcal{C} \rightarrow \mathcal{SET}^P$ .*

*Proof.* Clearly, the composition of LCC embeddings is an LCC embedding. We obtain  $E : \mathcal{C} \rightarrow \mathcal{SET}^P$  as a composite  $E_3 \circ E_2 \circ E_1$ . Here  $E_1 : \mathcal{C} \rightarrow \mathcal{SET}^{\mathcal{C}^{op}}$  is the Yoneda embedding, which maps  $A \in |\mathcal{C}|$  to  $\text{Hom}(-, A)$ . This is well-known to be an LCC embedding.  $E_2$  maps a presheaf on  $\mathcal{C}$  to a sheaf on a topological space  $S$ .  $E_2$  is the inverse image part of the spatial cover of the topos  $\mathcal{SET}^{\mathcal{C}^{op}}$  of presheaves on  $\mathcal{C}$ . This construction rests on a general topos-theoretical result established in [BM99], and we refer to [Awo00] for the details of the construction of  $S$ , the definition of  $E_2$ , and the proof that  $E_2$  is an LCC embedding. Finally  $E_3 : sh(S) \rightarrow \mathcal{SET}^{O(S)^{op}}$  includes a sheaf on  $S$  into the category of presheaves on the poset  $O(S)$  of open sets of  $S$ . That  $E_3$  is an LCC embedding, can be verified directly. Finally, we put  $P := O(S)^{op}$  so that  $E$  becomes an LCC embedding into  $\mathcal{SET}^P$ .  $\square$

**Definition 33** (Term-Generated). A  $\Sigma$ -model  $I$  is called term-generated if for all closed  $\Sigma$ -types  $S$  and every indexed element  $e \in \text{Elem}(\llbracket \cdot | S \rrbracket^I)$ , there is a  $\Sigma$ -term  $s$  of type  $S$  such that  $\llbracket \cdot | s \rrbracket^I = e$ .

**Theorem 34** (Model Existence). *For every signature  $\Sigma$ , there is a term-generated model  $I$  such that for all types  $\Gamma \vdash_{\Sigma} S : \text{type}$*

$$\text{Elem}(\llbracket \Gamma | S \rrbracket^I) \neq \emptyset \quad \text{iff} \quad \Gamma \vdash_{\Sigma} s : S \text{ for some } s, \quad (1)$$

and for all such terms  $\Gamma \vdash_{\Sigma} s : S$  and  $\Gamma \vdash_{\Sigma} s' : S$

$$\llbracket \Gamma | s \rrbracket^I = \llbracket \Gamma | s' \rrbracket^I \quad \text{iff} \quad \Gamma \vdash_{\Sigma} s \equiv s'. \quad (2)$$

*Proof.* It is well known how to construct the syntactic category  $\mathcal{C}$  from  $\Sigma$  and  $\Gamma$  ([See84]). The objects of  $\mathcal{C}$  are given by the set of all types  $S$  such that  $\vdash_{\Sigma} S : \text{type}$  modulo the equivalence relation  $\vdash_{\Sigma} S \equiv S'$ . We will write  $[S]$  for the equivalence class of  $S$ .

The  $\mathcal{C}$ -morphisms from  $[S]$  to  $[S']$  are given by the terms  $f$  such that  $\vdash_{\Sigma} f : S \rightarrow S'$  modulo the equivalence relation  $\vdash_{\Sigma} f \equiv f'$ . We will write  $[f]$  for the equivalence class of  $f$ .

It is straightforward to check that  $\mathcal{C}$  is LCC (see, e.g., [See84]). For example, the exponential  $f_2^{f_1}$  of two objects  $\vdash_{\Sigma} f_1 : S_1 \rightarrow S$  and  $\vdash_{\Sigma} f_2 : S_2 \rightarrow S$  in a slice  $\mathcal{C}/[S]$  is given by

$$\lambda_{u:U} \pi_1(u) \quad \text{where} \quad U := \Sigma_{x:S} (\Sigma_{y_1:S_1} \text{Id}(x, f_1 y_1) \rightarrow \Sigma_{y_2:S_2} \text{Id}(x, f_2 y_2)).$$

By Lem. 32, there are a poset  $P$  and an LCC embedding  $E : \mathcal{C} \rightarrow \mathcal{SET}^P$ . From those, we construct the needed model  $I$  over  $P$ . Essentially,  $I$  arises by interpreting every term or type as its image under  $E$ .

Firstly, assume a declaration  $c : S$  in  $\Sigma$ . Since  $\mathcal{C}$  only uses types and function terms,  $E$  cannot in general be applied to  $c$ . But using the type 1, every term  $c$  of type  $S$  can be seen as the function term  $\lambda_{x:1} c$  of type  $1 \rightarrow S$ . Therefore, we define  $E'(c) := E([\lambda_{x:1} c])$ , which is an indexed element of  $E([1 \rightarrow S])$ . Since  $\text{Elem}(E([1 \rightarrow S]))$  and  $\text{Elem}(E([S]))$  are in bijection,  $E'(c)$  induces an indexed element of  $E([S])$ , which we use to define  $\llbracket c \rrbracket^I$ .

Secondly, assume a declaration  $a:(\Gamma_0)\text{type}$  in  $\Sigma$  for  $\Gamma_0 = x_1 : S_1, \dots, x_n : S_n$ .  $\llbracket a \rrbracket^I$  must be an indexed set over  $\llbracket \Gamma_0 \rrbracket^I$ . For the same reason as above,  $E$  cannot be applied directly to  $a$ . Instead, we use the type  $U := \Sigma_{x_1:S_1} \dots \Sigma_{x_n:S_n} (a \text{id}_{\Gamma_0})$ . The fibration  $F(E([U])) : \int_P E(U) \rightarrow P$  factors canonically through  $\llbracket \Gamma_0 \rrbracket^I$ , from which we obtain the needed indexed set  $\llbracket a \rrbracket^I$ .

That  $I$  is term-generated now follows directly from the fullness of  $E$ . Finally, the required property (1) clearly follows from  $I$  being term-generated, and (2) from the fact that  $E$  is faithful.  $\square$

The fact that the model  $I$  just constructed is term-generated can be interpreted as functional completeness of the semantics: If a natural transformation of a certain type exists in every model, then it is syntactically definable. In more detail, let  $I$  be the model constructed in Thm. 34, and assume a natural transformation  $\eta : \llbracket \cdot | S \rrbracket^I \rightarrow \llbracket \cdot | S' \rrbracket^I$  for some  $\Sigma$ -types  $S$  and  $S'$ . Then there exists a  $\Sigma$ -term  $f$  of type  $S \rightarrow S'$  such that  $\eta$  arises from  $\llbracket \cdot | f \rrbracket^I$  as follows. Put  $\eta' := \text{am}(\llbracket \cdot | f \rrbracket^I) \in \text{Elem}(\llbracket x : S | S' \rrbracket^I)$ . Then  $\eta'$  maps pairs  $(p, a)$  to elements of  $\llbracket x : S | S' \rrbracket^I(p, a) = \llbracket \cdot | S' \rrbracket^I(p)$  for  $a \in \llbracket \cdot | S \rrbracket^I(p)$ . Then we obtain  $\eta$  as  $\eta_p : a \mapsto \eta'(p, a)$ .

**Theorem 35** (Completeness). *For every signature  $\Sigma$  and any type  $\Gamma \vdash_{\Sigma} S : \text{type}$ , the following hold:*

1. *If in every  $\Sigma$ -model  $I$  we have*

$$\text{Elem}(\llbracket \Gamma | S \rrbracket^I) \neq \emptyset,$$

*then there is a term  $s$  with*

$$\Gamma \vdash_{\Sigma} s : S.$$

2. *For all terms  $\Gamma \vdash_{\Sigma} s : S$  and  $\Gamma \vdash_{\Sigma} s' : S$ , if  $\llbracket \Gamma | s \rrbracket^I = \llbracket \Gamma | s' \rrbracket^I$  holds for all  $\Sigma$ -models  $I$ , then  $\Gamma \vdash_{\Sigma} s \equiv s'$ .*

*Proof.* This follows immediately from Thm. 34, considering the term-generated model constructed there.  $\square$

Finally, observe that in the presence of extensional identity types, statement (1) of Thm. 35 already implies statement (2): For all well-formed terms  $s, s'$  of type  $S$ , if  $\llbracket \Gamma | s \rrbracket = \llbracket \Gamma | s' \rrbracket$  in all  $\Sigma$ -models, then  $\llbracket \Gamma | \text{Id}(s, s') \rrbracket$  always has an element, and so there must be a term  $\Gamma \vdash_{\Sigma} t : \text{Id}(s, s')$ , whence  $\Gamma \vdash_{\Sigma} s \equiv s'$ . An analogous result for types is more complicated and remains future work.

## 9 Conclusion and Future Work

We have presented a concrete and intuitive semantics for MLTT in terms of indexed sets on posets. And we have shown soundness and completeness. Our semantics is essentially that proposed by Lawvere in [Law69] in the hyperdoctrine

of posets, fibrations, and indexed sets on posets, but we have made particular choices for which the models are coherent. Our models use standard function spaces, and substitution has a very simple interpretation as composition. The same holds in the simply-typed case, which makes our models an interesting alternative to (non-standard) Henkin models. In both cases, we strengthen the existing completeness results by restricting the class of models.

We assume that the completeness result can still be strengthened somewhat further, e.g., to permit equality axioms between types. In addition, it is an open problem to find an elementary completeness proof, i.e., one that does not rely on topos-theoretical results.

## References

- [All87] S. Allen. A Non-Type-Theoretic Definition of Martin-Löf’s Types. In D. Gries, editor, *Proceedings of the Second Annual IEEE Symp. on Logic in Computer Science, LICS 1987*, pages 215–221. IEEE Computer Society Press, 1987.
- [AR09] S. Awodey and F. Rabe. Kripke Semantics for Martin-Löf’s Extensional Type Theory. In P. Curien, editor, *Typed Lambda Calculi and Applications (TLCA)*, volume 5608 of *Lecture Notes in Computer Science*, pages 249–263. Springer, 2009.
- [Awo00] S. Awodey. Topological representation of the lambda-calculus. *Mathematical Structures in Computer Science*, 10(1):81–96, 2000.
- [Bar92] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.
- [BM99] C. Butz and I. Moerdijk. Topological representation of sheaf cohomology of sites. *Compositio Mathematica*, 2(118):217–233, 1999.
- [Car86] J. Cartmell. Generalized algebraic theories and contextual category. *Annals of Pure and Applied Logic*, 32:209–243, 1986.
- [CF58] H. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.
- [Chu40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.
- [Cur89] P. Curien. Alpha-Conversion, Conditions on Variables and Categorical Logic. *Studia Logica*, 48(3):319–360, 1989.
- [Fri75] H. Friedman. Equality Between Functionals. In R. Parikh, editor, *Logic Colloquium*, pages 22–37. Springer, 1975.

- [Hen50] L. Henkin. Completeness in the Theory of Types. *Journal of Symbolic Logic*, 15(2):81–91, 1950.
- [HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
- [Hof94] M. Hofmann. On the Interpretation of Type Theory in Locally Cartesian Closed Categories. In *CSL*, pages 427–441. Springer, 1994.
- [Hof97] M. Hofmann. Syntax and Semantics of Dependent Types. In A. Pitts and P. Dybjer, editors, *Semantics and Logic of Computation*, pages 79–130. Cambridge University Press, 1997.
- [How80] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.
- [Jac90] B. Jacobs. *Categorical Type Theory*. PhD thesis, Catholic University of the Netherlands, 1990.
- [Jac99] B. Jacobs. *Categorical Logic and Type Theory*. Elsevier, 1999.
- [Joh02] P. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium*. Oxford Science Publications, 2002.
- [Kri65] S. Kripke. Semantical Analysis of Intuitionistic Logic I. In J. Crossley and M. Dummett, editors, *Formal Systems and Recursive Functions*, pages 92–130. North-Holland, 1965.
- [Law69] W. Lawvere. Adjointness in Foundations. *Dialectica*, 23(3–4):281–296, 1969.
- [Lip92] J. Lipton. Kripke Semantics for Dependent Type Theory and Realizability Interpretations. In J. Myers and M. O’Donnell, editors, *Constructivity in Computer Science, Summer Symposium*, pages 22–32. Springer, 1992.
- [Mac98] S. Mac Lane. *Categories for the working mathematician*. Springer, 1998.
- [ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.
- [MM91] J. Mitchell and E. Moggi. Kripke-style Models for Typed Lambda Calculus. *Annals of Pure and Applied Logic*, 51(1–2):99–124, 1991.
- [MM92] S. Mac Lane and I. Moerdijk. *Sheaves in geometry and logic*. Lecture Notes in Mathematics. Springer, 1992.

- [MS89] J. Mitchell and P. Scott. Typed lambda calculus and cartesian closed categories. In *Categories in Computer Science and Logic*, volume 92 of *Contemporary Mathematics*, pages 301–316. Amer. Math. Society, 1989.
- [Pit00] A. Pitts. Categorical Logic. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 5. Algebraic and Logical Structures*, chapter 2, pages 39–128. Oxford University Press, 2000.
- [Rab08] F. Rabe. *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen, 2008. see <http://kwarc.info/frabe/Research/phdthesis.pdf>.
- [See84] R. Seely. Locally cartesian closed categories and type theory. *Math. Proc. Cambridge Philos. Soc.*, 95:33–48, 1984.
- [Sim95] A. Simpson. Categorical completeness results for the simply-typed lambda-calculus. In M. Dezani-Ciancaglini and G. Plotkin, editor, *Typed Lambda Calculi and Applications*, pages 414–427, 1995.
- [Str91] T. Streicher. *Semantics of Type Theory*. Springer-Verlag, 1991.

# First-Order Logic with Dependent Types

Florian Rabe  
`florian@cs.cmu.edu` \*

Carnegie Mellon University and International University Bremen

**Abstract.** We present DFOL, an extension of classical first-order logic with dependent types, i.e., as in Martin-Löf type theory, signatures may contain type-valued function symbols. A model theory for the logic is given that stays close to the established first-order model theory. The logic is presented as an institution, and the logical framework LF is used to define signatures, terms and formulas. We show that free models over Horn theories exist, which facilitates its use as an algebraic specification language, and show that the classical first-order axiomatization is complete for DFOL, too, which implies that existing first-order theorem provers can be extended. In particular, the axiomatization can be encoded in LF.

## 1 Introduction and Related Work

Classical first-order logic (FOL) and its variations are folklore knowledge. Lots of variations classify the elements of a model into different sorts, e.g., many-sorted or order-sorted FOL. Type theory may also be viewed as an extension of FOL, introducing function sorts. Further extensions of type theory include type operators, type polymorphism and dependent types. All these extensions have varying advantages and disadvantages and various implementations exist. Surprisingly, not much work has been undertaken to extend FOL with just dependent types.

PVS ([ORS92]) is a classical verification system that extends simply-typed higher order logic with dependent function, record and product types and other concepts. Its type system is undecidable, and a set theoretic semantics for an idealized language exists ([OS97]). Coq ([BC04]), Nuprl ([CAB<sup>+</sup>86]) and LF ([Pfe01]) implement Martin-Löf's intuitionistic type theory with dependent types ([ML74]); while the first two add further concepts and are directed at general mathematics, the main purpose of LF is the specification of logics. A semantics for Martin-Löf type theory was introduced in [Car86], where also algebraic theories are treated. It was linked with locally cartesian closed categories as analogues of FOL structures in [See84] (see also [Hof94] and [Dyb95] for related approaches). However, these concepts are mathematically very complex and not tightly connected to research on theorem proving. Neither are they easy to specialize to FOL, even if intuitionistic FOL is used.

---

\* The author was supported by a fellowship for Ph.D. research of the German Academic Exchange Service.

We could only locate one attempt at combining FOL with just dependent types ([Mak], never published), which is mainly directed at studying equivalence of categories. It adds connectives and quantifiers to the treatment in [Car86] and gives an axiomatization, but does not allow general equality and function symbols (without which dependent types are significantly less interesting). Their type hierarchy is similar to ours, but the chosen notation is completely different from the usual one.

Our motivation in defining DFOL is to add as little as possible to FOL, keeping not only notation and intuition but also the results and applications. Thus, both researchers and implementations can use DFOL more easily. Therefore, we deliberately dispense with one feature of dependent types, namely circular dependencies, which greatly simplifies the model theory while hardly excluding interesting applications.

DFOL is presented as an institution. Institutions were introduced in [?] as a unifying concept for model theory. Examples for institutional definitions of logics are OBJ ([GWM<sup>+</sup>93]) and Maude ([CELM96]). The syntax of DFOL is presented directly in LF (see [HST94] for other logic specifications in LF) because even in our special case the inherent complexity of dependent types makes any independent introduction inconvenient. We introduce DFOL in section 2, sections 3 to 5 treat free models, axiomatization and examples.

## 2 Syntax and Semantics

### 2.1 Preliminary Definitions

*Institutions* An institution is a tuple  $(\text{Sig}, \text{Sen}, \text{Mod}, \models)$  where  $\text{Sig}$  is a category of signatures; signature morphisms are notation changes, usually mappings between the symbols of the signatures;  $\text{Sen} : \text{Sig} \rightarrow \text{Set}$  is a functor that assigns to each signature its set of formulas and with each signature morphism the induced formula translation;  $\text{Mod} : \text{Sig} \rightarrow \text{Cat}^{\text{op}}$  is a functor that assigns to every signature its category of models, and with every signature morphism the induced model reduction; and for every signature  $\Sigma$ , the satisfaction relation  $\models_{\Sigma} \subseteq |\text{Mod}(\Sigma)| \times \text{Sen}(\Sigma)$  between models and sentences determines truth. Institutions must satisfy the satisfaction condition which can be paraphrased as "Truth is invariant under change of notation.". We refer to [?] for a thorough introduction.

*LF* The logical framework LF and its implementation Twelf ([Pfe01], [PS99]) implement Martin-Löf type theory ([ML74]). LF will be used as a meta-language to define the syntax of DFOL. An LF signature<sup>1</sup> consists of a sequence  $c : T$  of declarations, where  $c$  is a new symbol, and  $T$  is its type, which may depend on the previously declared symbols.  $T$  is of the form  $\prod x_1:T_1. \dots \prod x_n:T_n. T_{n+1}$ , which means that  $c$  is a function symbol taking arguments of the types  $T_1, \dots, T_n$ , called  $x_1, \dots, x_n$ , and returning an argument of the type  $T_{n+1}$ ; dependent types

---

<sup>1</sup> Readers familiar with LF will notice that our introduction is highly simplified.

means that  $x_i$  may occur in  $T_{i+1}, \dots, T_{n+1}$ . If  $T_{n+1} = \text{type}$ ,  $c$  is not a function symbol but returns a new dependent type for every argument tuple. If  $x$  does not occur in  $B$ ,  $\Pi x:A. B$  is abbreviated by  $A \rightarrow B$ .

To illustrate this, look at the signature  $\Sigma_B$  which represents our metalinguage:

$$\begin{aligned} \mathbf{S} : \text{type}. \quad \text{Univ} : \mathbf{S} \rightarrow \text{type}. \quad o : \text{type}. \\ \text{true}, \text{false} : o. \quad \wedge, \vee, \Rightarrow : o \rightarrow o \rightarrow o. \quad \neg : o \rightarrow o. \\ \forall, \exists : \prod S:\mathbf{S}. (\text{Univ } S \rightarrow o) \rightarrow o. \quad \doteq : \prod S:\mathbf{S}. \text{Univ } S \rightarrow \text{Univ } S \rightarrow o. \end{aligned}$$

In this signature,  $\mathbf{S}$  is a type, the type of sorts declared in a DFOL signature.  $\text{Univ}$  is a dependent type family that returns a new type for each sort  $S$ , namely the type of terms of sort  $S$ ; models will interpret the type  $\text{Univ } S$  as the universe for the sort  $S$ .  $o$  is the type of formulas. The remainder of the signature encodes the usual grammar for FOL formulas. Higher-order abstract syntax is used, i.e.,  $\lambda$  is used to bind the free variables in a formula, and quantifiers are operators taking a  $\lambda$  expression as an argument.<sup>2</sup> Quantifiers and the equality symbol take the sort they operate on as their first argument; we will omit this argument if no ambiguities arise. When we refer to sorts, terms or formulas, we always mean objects with the respective type that do not contain any lambda abstractions except for those preceded by quantifiers.

A context for a signature  $\Sigma$  is a sequence of typed variables  $x : \text{Univ } S$ , where previously declared variables and symbols declared in  $\Sigma$  may occur in  $S$ . Sorts, terms and formulas in context  $C$  may contain the variables declared in  $C$ .

We introduce abbreviations to make the LF syntax more familiar: The usual infix notation and bracket conventions of FOL are used, and conjunction binds stronger than implication:  $\forall \lambda x : \text{Univ } S. F$  is abbreviated as  $\forall x : S. F$ , and we write  $\forall x, y : S, z : S'$  instead of  $\forall x : S. \forall y : S. \forall z : S'$ , and similarly for  $\exists$ . Note that application is written as  $f t_1 \dots t_n$  instead of the familiar  $f(t_1, \dots, t_n)$  and that substitution is written as  $\beta$ -reduction.

We allow a harmless<sup>3</sup> extension of LF: Contexts and signatures need not be finite but may contain infinitely many declarations of the form  $c : \text{Univ } S$ . The reason for this is purely technical: It allows to have an infinite reservoir of names  $c$  for elements that occur in the universe of  $S$ , which facilitates some proofs.

## 2.2 Signatures

We are now ready to introduce DFOL signatures as certain LF signatures. A DFOL signature will consist of  $\Sigma_B$  followed by declarations of the form

$$c : \prod x_1 : \text{Univ } S_1. \dots. \prod x_m : \text{Univ } S_m. T$$

where  $m \in \mathbb{N}$ ,  $T = \mathbf{S}$ ,  $T = \text{Univ } S$  or  $T = o$ , and  $S_1, \dots, S_m, S$  are sorts.  $c$  is called a sort symbol if  $T = \mathbf{S}$ , a function symbol with target  $S$  if  $T = \text{Univ } S$ ,

---

<sup>2</sup> The reflexivity of LF is used to encode the sort dependencies. Alternatively,  $\mathbf{S}$  could be replaced with `type` and `Univ` omitted everywhere. But then sorts could not be used as arguments since LF does not support polymorphism.

<sup>3</sup> It is not harmless in general, only in our setting as explained below.

and a predicate symbol if  $T = o$ . The sorts  $S_i$  are called arguments of  $c$ . If we only allow sort symbols without arguments, we obtain the usual many-sorted FOL. Sort symbols with arguments construct dependent sorts.

**Definition 1 (Signatures).** Let  $\Sigma_i$  be partial LF signatures such that  $\Sigma = \Sigma_B \Sigma_0 \dots \Sigma_d$  is an LF signature, and let  $\Sigma^n$  abbreviate  $\Sigma_B \Sigma_0 \dots \Sigma_n$  for  $n \leq d$ .  $\Sigma$  is called a DFOL signature if

1. only sort, function or predicate symbols are declared in  $\Sigma_0 \dots \Sigma_d$ ,
2. all sort symbol declarations in  $\Sigma_n$  have only arguments from  $\Sigma^{n-1}$ ,
3. the target of a function symbol declaration in  $\Sigma_n$  is not an LF term over  $\Sigma^{n-1}$  (i.e., only over  $\Sigma_n$ ).

Condition 1 prevents the use of more expressive LF declarations<sup>4</sup>. Condition 2 establishes an acyclic sort dependency: Every sort declared in  $\Sigma_n$  may only depend on sorts that have been declared in  $\Sigma^{n-1}$ .  $d$  is called the depth of  $\Sigma$ . A sort, term or formula has depth  $n$  if it is defined over  $\Sigma^n$  (for some context) but not over  $\Sigma^{n-1}$ . Condition 3 is the most important one: It requires that a function symbol that takes a sort of depth  $n$  as an argument may not return an element of a smaller depth.<sup>5</sup>

Condition 3 is rather restrictive to ensure the existence of free models. It excludes, e.g., projection functions  $\pi$  from a sort  $T n$  of  $n$ -tuples (at depth 1) over  $B$  to  $B$  (at depth 0). A weaker restriction that is still sufficient for free models could allow  $\pi$  if there are equality axioms that identify every term  $\pi x$  with a term of depth 0, i.e., if  $\pi$  does not generate new objects.

We have the following property.

**Lemma 1.** Let  $\Sigma$  be a DFOL signature of depth  $d$ . Then  $\Sigma^n$ , for  $0 \leq n \leq d$ , is a DFOL signature such that the sorts of  $\Sigma^n$  are precisely the sorts of  $\Sigma$  that have depth at most  $n$ ; and such that the terms of a sort  $S$  of  $\Sigma^n$  are precisely the terms of sort  $S$  over  $\Sigma$ .

*Proof.* Trivial but note how condition 3 is needed.

This ensures that infinitely many declarations of the form  $c : Univ S$  are indeed harmless:  $S$  may depend on the previously declared symbols but these must have strictly smaller depth than  $S$  and so on; therefore,  $S$  can only depend on finitely many symbols. From now on let the word signature only refer to DFOL signatures.

In general every sort symbol in  $\Sigma_n$  has a type of the form

$$\Pi x_1 : Univ S_1. \dots . \Pi x_m : Univ S_m. \mathbf{S} \quad (\text{F } 1)$$

---

<sup>4</sup> In particular, function types may not occur as arguments.

<sup>5</sup> In the terminology of [Vir96], this corresponds to dependence relations between sort symbols given by  $s \prec t$  iff  $s$  has at most the depth of  $t$ .

where  $S_1, \dots, S_m$  have depth smaller than  $n$ . Without loss of generality, we can assume that every function symbol in  $\Sigma_n$  has a type of the form

$$\Pi x_1 : \text{Univ } S_1. \dots. \Pi x_r : \text{Univ } S_r. \text{Univ } S_{r+1} \rightarrow \dots \rightarrow \text{Univ } S_m \rightarrow \text{Univ } S \quad (\text{F } 2)$$

where  $S_1, \dots, S_r$  have depth smaller than  $n$  and  $S_{r+1}, \dots, S_m$  have depth  $n$ . Similarly, we can assume that every predicate symbol in  $\Sigma_n$  has a type of the form

$$\Pi x_1 : \text{Univ } S_1. \dots. \Pi x_r : \text{Univ } S_r. \text{Univ } S_{r+1} \rightarrow \dots \rightarrow \text{Univ } S_m \rightarrow o. \quad (\text{F } 3)$$

A signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$  is a mapping of  $\Sigma$  symbols to  $\Sigma'$  symbols such that (i)  $\sigma$  is the identity for symbols declared in  $\Sigma_B$ ; (ii)  $\sigma$  respects types and depths of all mapped symbols. We omit the formal definition. For example, the identity mapping is a signature morphism from  $\Sigma_B \Sigma_0 \dots \Sigma_c$  to  $\Sigma = \Sigma_B \Sigma_0 \dots \Sigma_d$  for  $c \leq d$ . These morphisms are called extensions.

*Running Example* We will use a running example. Consider the signature  $\Sigma$

$$\begin{aligned} t : \mathbf{S}. \quad i : \text{Univ } t. \quad j : \text{Univ } t. \quad d : \text{Univ } t \rightarrow \mathbf{S}. \\ f : \Pi x : \text{Univ } t. \text{Univ } d x. \quad c : \text{Univ } d i. \quad p : \Pi x : \text{Univ } t. \text{Univ } d x \rightarrow o \end{aligned}$$

$\Sigma$  has depth 1 ( $\Sigma_0$  consists of the declarations for  $t$ ,  $i$  and  $j$ ) and declares three sorts,  $t$ ,  $d i$  and  $d j$ , and five terms,  $i$  and  $j$  with sort  $t$ ,  $f i$  and  $c$  with sort  $d i$ , and  $f j$  with sort  $d j$ , and one predicate.

### 2.3 Sentences and Models

**Definition 2 (Sentences).** *Sen( $\Sigma$ )*, the set of formulas over  $\Sigma$ , is the set of LF objects  $F = \lambda x_1 : \text{Univ } S_1. \dots. \lambda x_n : \text{Univ } S_n. G$  where  $G$  is of type  $o$  and all  $\lambda$ 's in  $G$  are preceded by  $\forall$  or  $\exists$ . For a signature morphism  $\sigma$ ,  $\text{Sen}(\sigma)$  is the mapping between formulas induced by  $\sigma$ .

Taking the above lambda closure over the free variables in  $G$  is not needed in FOL. In DFOL, however, it is helpful to keep track of the sorts of the free variables in  $G$  because  $x_i$  may occur in  $S_{i+1}, \dots, S_n$ . For simplicity, we identify  $G$  and  $F$  if it does not cause confusion. We do not distinguish between, e.g.,  $F$  and  $\lambda x : \text{Univ } S. F$  if  $x$  does not occur in  $F$ . Closed and atomic formulas are defined in the obvious way.

*Running Example* Examples for closed formulas are  $E_1 = \forall x : t. (i \doteq x \Rightarrow p x f x)$  and  $E_2 = \forall x : t. \exists y : d x. (p x y)$ .

**Definition 3 (Models, Assignments).** Let  $\Sigma$  have depth  $d$ . We define  $\Sigma$ -models and assignments by induction on  $d$ . If  $d = 0$ ,  $\text{Mod}(\Sigma)$  is the category of many-sorted FOL models of  $\Sigma$ , i.e., interpretation functions  $M$  given by

- a universe  $s^M$  for every sort symbol  $s : \mathbf{S} \in \Sigma$ ,

- a function  $f^M : s_1^M \times \dots \times s_m^M \rightarrow s^M$  for every function symbol  $f$  with  $m$  arguments,
- a mapping  $p^M : s_1^M \times \dots \times s_m^M \rightarrow \{0, 1\}$  for every predicate symbol  $p$  with  $m$  arguments.

If  $d > 0$  and  $\text{Mod}$  is defined for signatures of depth  $d - 1$ , the objects  $M \in \text{Mod}(\Sigma)$  are interpretation functions  $\cdot^M$  that interpret the sort, function and predicate symbols of  $\Sigma$  in the following way.

1.  $\cdot^M$  restricted to  $\Sigma^{d-1}$  (which has depth  $d - 1$ ) is a  $\Sigma^{d-1}$ -model. We denote this restriction by  $N$ .

2. Let  $C = (x_i : \text{Univ } S_i)_{i=1,\dots,n}$  be a context over  $\Sigma^{d-1}$ . We define assignments and model extensions through two entwined recursions:

- An assignment from  $C$  into  $N$  is a mapping  $u$  that assigns to every  $x_i : \text{Univ } S_i$  in  $C$  an element  $u(x_i) \in S_i^N(u)$ . Here,  $S_i^N(u)$  is the extension of  $\cdot^N$  to sorts induced by the assignment  $u$ .
- The extension of  $\cdot^N$  to sorts and terms for an assignment  $u$  from the respective free variables into  $N$ , is defined recursively by  $x_i^N(u) = u(x_i)$  and

$$(c t_1 \dots t_m)^N(u) = c^N(t_1^N(u), \dots, t_m^N(u))$$

for a sort or function symbol  $c$  and terms  $t_i$ .

3. For every sort symbol of the form (F 1) declared in  $\Sigma_d$  and every assignment  $u$  from  $(x_i : \text{Univ } S_i)_{i=1,\dots,m}$  into  $N$ ,

$$s^M(u(x_1), \dots, u(x_m)) \text{ is a set.}$$

4. Note that at this point,  $\cdot^M$  is defined for all sort symbols in  $\Sigma_d$  and all terms of depth at most  $d - 1$ . As in step 2, we define assignments into  $M$  and the extension  $\cdot^M$  to sorts of depth  $d$ .

5. For every function symbol  $f$  of the form (F 2) declared in  $\Sigma_d$  and every assignment  $u$  from  $(x_i : \text{Univ } S_i)_{i=1,\dots,m}$  into  $M$ ,

$$f^M(u(x_1), \dots, u(x_m)) \in S^M.$$

6. As in step 2, we define  $\cdot^M$  for all terms of depth  $n$ .

7. For every predicate symbol  $p$  of the form (F 3) declared in  $\Sigma_d$  and every assignment  $u$  from  $(x_i : \text{Univ } S_i)_{i=1,\dots,m}$  into  $M$ ,

$$p^M(u(x_1), \dots, u(x_m)) \in \{0, 1\}.$$

For a sort  $S$ ,  $S^M$  is called the universe of  $M$ . A model morphism  $\varphi : M \rightarrow N$  for  $\Sigma$ -models  $M$  and  $N$  maps from each universe of  $M$  to some universe of  $N$  as follows.<sup>6</sup> For every assignment  $u$  from  $(x_i : \text{Univ } S_i)_{i=1,\dots,m}$  into  $M$ , we put  $\mathbf{u} = (u(x_1), \dots, u(x_m))$ ; then  $\varphi(\mathbf{u}) = (\varphi(u(x_1)), \dots, \varphi(u(x_m)))$  is an assignment into  $N$ . We require that for every  $d$  and every appropriate assignment  $u$  into  $M$

---

<sup>6</sup> Since the universes of  $M$  are not required to be pairwise disjoint,  $\varphi$  should be indexed with the universes to distinguish these mappings. We omit these indexes and rely on the context.

1. for every sort symbol  $s$  declared in  $\Sigma_d$ ,  $\varphi$  is a mapping from  $s^M(\mathbf{u})$  to  $s^N(\varphi(\mathbf{u}))$ ,
2. for every function symbol  $f$  declared in  $\Sigma_d$ ,  $\varphi(f^M(\mathbf{u})) = f^N(\varphi(\mathbf{u}))$ ,
3. for every predicate symbol  $p$  declared in  $\Sigma_d$ ,  $p^M(\mathbf{u}) \leq p^N(\varphi(\mathbf{u}))$ .<sup>7</sup>

Note that for  $d = 0$ , this reduces to the usual first-order definition of homomorphisms.

For a signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$ ,  $Mod(\sigma)$  is the usual model reduction functor from  $Mod(\Sigma')$  to  $Mod(\Sigma)$ , i.e.,  $Mod(\sigma)$  maps a  $\Sigma'$ -model  $N$  to a  $\Sigma$ -model  $M$  defined by  $c^M = \sigma(c)^N$  for every symbol  $c$  of  $\Sigma$  (Thus every universe of  $M$  is also a universe of  $N$ .); and  $Mod(\sigma)$  maps a  $\Sigma'$ -model morphism from  $N$  to  $N'$  to its restriction to the universes of  $Mod(\sigma)(N)$ . We omit the formal proof that  $Mod$  is indeed a functor.

In particular for an extension  $\varphi : \Sigma^n \rightarrow \Sigma^d$  for  $n \leq d$ ,  $Mod(\varphi)$  maps every  $\Sigma^d$ -model  $M$  to its restriction  $M^n$  over  $\Sigma^n$ .

*Running Example* A model  $M$  for the example signature is given by  $t^M = \mathbb{N} \setminus \{0\}$ ,  $i^M = 2$ ,  $j^M = 3$ ,  $d^M(n) = \mathbb{N}^n$ ,  $f(n) = (1, \dots, n)$ , and  $p(n, (m_1, \dots, m_n)) = 1 \Leftrightarrow m_1 = n$ . Note how all interpretations must be defined for all elements regardless of whether they can be named by terms.

## 2.4 Satisfaction

Satisfaction  $M \models_{\Sigma} F$  is defined in the usual way:  $M \models_{\Sigma} F \Leftrightarrow F^M(u) = 1$  for all assignments  $u$  from the free variables in  $F$  into  $M$ , where  $\cdot^M$  is extended to all formulas by

- if  $F = p t_1 \dots t_m$  for a predicate symbol  $p$ , then  $F^M(u) = p^M(t_1^M(u), \dots, t_m^M(u))$ ,
- if  $F = t_1 \dot{=} t_2$ , then  $F^M(u) = 1 \Leftrightarrow t_1^M(u) = t_2^M(u)$ ,
- the usual definition for propositional connectives,
- if  $F = \forall x : S. G$ , then  $F^M(u) = \inf\{G^M(u\{x \mapsto v\}) \mid v \in S^M(u)\}$ ,
- if  $F = \exists x : S. G$ , then  $F^M(u) = \sup\{G^M(u\{x \mapsto v\}) \mid v \in S^M(u)\}$ ,

where  $u\{x \mapsto v\}$  is as  $u$  but with  $u(x) = v$ . We omit the formal proof of the satisfaction condition.

*Running Example* We have  $M \not\models_{\Sigma} E_1$  since for  $u(x) = 2$ , we have  $(i \dot{=} x)^M(u) = 1$  but  $(f x)^M(u) = (1, 2)$  and therefore,  $(p x f x)^M(u) = 0$ . And we have  $M \models_{\Sigma} E_2$  since for every  $n \in t^M$  there is an  $m \in d^M(n)$  such that  $p^M(n, m) = 1$ , for example  $m = (n, 1, \dots, 1)$ .

---

<sup>7</sup> Using  $\leq$  means that truth of atomic formulas is preserved along homomorphisms.

### 3 Free Models

A theory  $K = (\Sigma, T)$  consists of a signature  $\Sigma$  and a set  $T$  of closed  $\Sigma$ -formulas.  $Sen$  and  $Mod$  are extended to theories by putting  $Sen(\Sigma, T) = Sen(\Sigma)$  and  $Mod(\Sigma, T) = \{M \in Mod(\Sigma) \mid M \models F \text{ for all } F \in T\}$ .

For many-sorted first-order logic there is the standard result (see for example [?]) that for a Horn theory  $(\Sigma, T)$  and sets  $A_s$  of generators for all sort symbols  $s$ , there is a  $\Sigma$ -model  $Free_\Sigma(A)/T$  of  $T$  such that for every  $M \in Mod(\Sigma, T)$  and every family of mappings  $\varphi_s : A_s \rightarrow s^M$  there is a unique  $\Sigma$ -morphism  $\bar{\varphi} : Free_\Sigma(A)/T \rightarrow M$  that extends  $\varphi$ .

The purpose of this section is to establish the corresponding result for DFOL. Horn formulas over a DFOL signature  $\Sigma$  are defined in the usual way (i.e., a universally closed implication  $T \Rightarrow H$  in which the tail  $T$  is a conjunction of atomic formulas and the head  $H$  is an atomic formula except for *false*). A Horn formula is hierachic if the depth of its head is at least as big as the depth of its tail. We can allow only hierachic Horn formulas because, informally, otherwise axioms of a greater depth could influence the interpretation of symbols of a lower depth. As generators, we might use a family  $A_S$  where  $S$  runs over all sorts, but a stronger result is possible that also allows generators for sorts that depend on other generators. The most elegant way to describe such generators is by using signatures that contain arbitrarily many constant declarations of the form  $a : Univ S$ .<sup>8</sup> Then DFOL has free models over hierachic Horn theories in the following sense.

**Lemma 2.** *For a hierachic Horn theory  $K = (\Sigma, T)$ , there is a  $K$ -model  $Free_\Sigma/T$  such that for every  $K$ -model  $M$  there is a unique  $\Sigma$ -morphism from  $Free_\Sigma/T$  to  $M$ .*

*Proof.* Let  $\Sigma$  and  $T$  be as stated. Let  $T^n$  be the restriction of  $T$  to depth at most  $n$ . We define  $Free_\Sigma/T$  by induction on the depth of  $\Sigma$ , say  $d$ . If  $d = 0$ ,  $F^0 = Free_{\Sigma^0}/T^0$  is the classical result; note that the universes of  $F^0$  arise by taking equivalence classes of terms.

By the induction hypothesis, we assume

$$F^{d-1} = Free_{\Sigma^{d-1}}/T^{d-1} \in Mod(\Sigma^{d-1}, T^{d-1})$$

all universes of which consist of equivalence classes of terms, let  $[.]$  denote these equivalence classes. We define Herbrand models to be  $(\Sigma^d, T^d)$ -models  $M$  that satisfy the following conditions:

- $M$  agrees with  $F^{d-1}$  for all symbols in  $\Sigma^{d-1}$ ,
- for all sort symbols  $s$  of the form (F 1) in  $\Sigma_d$ :  $s^M([t_1], \dots, [t_m]) = U/\equiv$  where  $U$  contains those terms that have any of the sorts  $s b_1 \dots b_m$  for  $b_i \in [t_i]$ , and  $\equiv$  is some equivalence relation<sup>9</sup>; let  $\langle \cdot \rangle$  denote the equivalence classes of  $\equiv$ ; clearly, all universes are disjoint, so that we can use the symbols  $\equiv$  and  $\langle \cdot \rangle$  for all universes,

---

<sup>8</sup> This is why we allow infinite signatures.

– for a function symbol  $f$  of the form (F 2) in  $\Sigma_d$ :

$$f^M([t_1], \dots, [t_r], \langle t_{r+1} \rangle, \dots, \langle t_m \rangle) = \langle f \ t_1 \dots t_m \rangle,$$

– for a predicate symbol  $p$  of the form (F 3) in  $\Sigma_d$ :

$$p^M([t_1], \dots, [t_r], \langle t_{r+1} \rangle, \dots, \langle t_m \rangle) = 1 \Leftrightarrow M \models_{\Sigma^d} p \ t_1 \dots t_m.$$

We put  $F^d$  to be the Herbrand model that satisfies the least atomic formulas. By definition, it is a  $(\Sigma^d, T^d)$ -model in which all universes arise by taking equivalence classes of terms.

We have to prove that the above  $F^d$  exists. This is done in essentially the same way as for the FOL case. Informally, a Herbrand model is uniquely determined by the equivalence  $\equiv$  and the set  $P$  of atomic formulas of depth  $d$  that it satisfies. Let  $(\equiv_i, P_i)_{i \in I}$  be all Herbrand models. This family is not empty: It contains the model in which all atomic formulas of depth  $d$  are true. Then it is easy to show that this family also contains the model determined by  $\bigcap_{i \in I} \equiv_i$  and  $\bigcap_{i \in I} P_i$ , which we can put to be  $F^d$ . This completes the induction.

The unique morphism into  $M$  simply maps the equivalence class of a term  $t$  to  $t^M$ . This is well-defined because  $M$  satisfies  $T$  and by the definition of  $F^d$ .

*Running Example* The free model  $F$  for the example signature with the axioms  $\forall x : t. (p \ x \ f \ x)$  and  $c \doteq f \ i$  is given by  $t^F = \{\{i\}, \{j\}\}$ ,  $d^F(\{i\}) = \{\{c, f \ i\}\}$ ,  $d^F(\{j\}) = \{\{f \ j\}\}$  and  $(p \ x \ y)^F(u) = 1$  for both possible assignments  $u$ .

*Generators* Since we allow infinite signatures, Lem. 2 contains the result where there are arbitrarily many generators. We make this more precise: Let  $\Sigma(A)$  be the signature  $\Sigma$  enriched with the declarations from a context  $A$  over  $\Sigma$ . Then every  $\Sigma$ -model  $M$  and every assignment  $u$  from  $A$  into  $M$  induce a  $\Sigma(A)$ -model, which we call  $(M, u)$ .

**Theorem 1.** *For a signature  $\Sigma$ , a context  $A$  over  $\Sigma$ , and a set  $T$  of hierachic Horn formulas over  $\Sigma(A)$ , there is a  $\Sigma(A)$ -model  $Free_{\Sigma}(A)/T$  of  $T$  such that for every  $\Sigma$ -model  $M$  and every assignment  $u$  from  $A$  into  $M$  such that  $(M, u)$  models  $T$ , there is a unique  $\Sigma(A)$ -morphism  $\bar{u} : Free_{\Sigma}(A)/T \rightarrow (M, u)$  that extends  $u$ .*

*Proof.* Simply put  $Free_{\Sigma}(A)/T$  to be  $Free_{\Sigma(A)}/T$ .

Of particular interest is the case where  $T$  does not depend on  $A$ . Then  $Free_{\Sigma}(A)/T$  is a  $(\Sigma, T)$ -model and every assignment  $u$  from  $A$  into a  $(\Sigma, T)$ -model  $M$  has a unique extension to a  $\Sigma$ -morphism, namely the extension of the interpretation function  $M$  under the assignment  $u$ .

We abstain from a categorical interpretation in terms of adjoint functors and simply remark that  $Free_{\Sigma}(\emptyset)/T$  is initial in the category  $Mod(\Sigma, T)$ .

---

<sup>9</sup> Note that  $\equiv$  identifies more terms than the relation  $(t \doteq t')^M = 1$ : Term identification in  $F^{d-1}$  may lead to sort identification at depth  $d$ , thus causing terms of different sorts to become equal.

*Running Example* The free model  $F$  for the example signature with the generators  $A = a_1 : t$ ,  $a_2 : f a_1$  with the axioms  $E_1$  (from the running example above) and  $i \doteq a_1$  is given by:  $t^F = \{\{i, a_1\}, \{j\}\}$ ,  $d^F(\{i, a_1\}) = \{\{f i, f a_1\}, \{c\}, \{a_2\}\}$ ,  $d^F(\{j\}) = \{\{f j\}\}$  and  $(p x y)^F(u) = 1$  precisely for  $u(x) = \{i, a_1\}$ ,  $u(y) = \{f i, f a_1\}$ .

## 4 Axiomatization

$$\begin{array}{c}
\frac{}{\vdash A} \text{ for every } A \in T \quad \frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \quad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta} \\
\frac{}{\Gamma \vdash \text{true}, \Delta} \quad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \quad \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \\
\frac{\Gamma, A \vdash \Delta}{\Gamma, \exists x : S. A \vdash \Delta} \quad \frac{\Gamma \vdash (\lambda x : S. A) t, \Delta}{\Gamma \vdash \exists x : S. A, \Delta} \\
\frac{\vdash \forall x_1 : S_1, \dots, x_r : S_r, x_{r+1}, x'_{r+1} : S_{r+1}, \dots, x_m, x'_m : S_m.}{x_{r+1} \doteq x'_{r+1} \wedge \dots \wedge x_m \doteq x'_m \Rightarrow} \\
\frac{f x_1 \dots x_r x_{r+1} \dots x_m \doteq f x_1 \dots x_r x'_{r+1} \dots x'_m}{\text{for all } f \text{ of the form (F 2)}}
\end{array}$$

In the  $\exists$ left rule, we assume that  $x$  does not occur free in  $\Gamma$  or  $\Delta$ . In the  $\exists$ right rule,  $t$  is a term of sort  $S$  in which the free variables from  $\Gamma$ ,  $S$ ,  $A$  and  $\Delta$  may occur. We omit the structural rules (axioms, cut, weakening, contraction and exchange), the remaining equality rules (reflexivity, symmetry and transitivity) and the rules for definable connectives and quantifiers.

**Fig. 1.** Axiomatization for  $SC(\Sigma, T)$

*Completeness* To enhance readability, we omit the operator *Univ* completely from now on. We show that the classical axiomatizations are sufficient for DFOL. We use the common Gentzen style notation for sequents and rules. For simplicity, we only give the completeness result for the case that empty universes are forbidden. The general case is similar.

**Theorem 2.** *Let  $K = (\Sigma, T)$  be a finite theory (i.e.,  $\Sigma$  and  $T$  are finite), and let the rules of  $SC(K)$  be as in classical sequent style axiomatizations for FOL with equality (see [Gal86]) with slight modifications as given in Fig. 1. Then  $SC(K)$  is sound and complete for  $Mod(K)$ .*

The modifications mainly serve to account for free variables. Only the congruence axiom has an unfamiliar form and may even look incomplete, but note that the putatively underivable formulas are not well-typed in the first place.

*Proof.* We only sketch the completeness proof since it is almost the same as the classical Henkin style proof (see [Hen49]). There are only a few minor technical differences in the notation of free variables and quantifiers. Firstly, a set of formulas  $\Gamma$  such that  $\Gamma \cup T$  is consistent is extended to a maximal consistent set  $\overline{\Gamma}$  with witnesses. To do that, we use a context  $A$  containing infinitely many declarations for each sort over  $\Sigma$  and  $A$ .

Then  $M = \text{Free}_\Sigma(A)/T$ , where  $T$  is the set of atomic formulas in  $\overline{\Gamma}$ , is a  $(\Sigma(A), T)$ -model of  $\overline{\Gamma}$ . The proof proceeds by induction on the number of occurrences of logical symbols in  $F \in \overline{\Gamma}$ .

Then the  $\Sigma(A)$ -model  $M$  yields a  $\Sigma$ -model  $M'$  by forgetting the interpretations of the symbols from  $A$  (formally  $M' = \text{Mod}(\sigma)(M)$  where  $\sigma : \Sigma \rightarrow \Sigma(A)$  is the injection).  $M'$  is a model of  $\Gamma \cup T$  because no symbols from  $A$  occur in  $\Gamma \cup T$ . From this model existence result, the theorem follows as in the classical case.

As in the classical case, Thm. 4 yields the compactness theorem and the Löwenheim-Skolem result that any consistent set of sentences has a countable model.

*Running Example* Let an axiom for  $\Sigma$  be given by  $i \doteq j$ . This implies that the sorts  $d i$  and  $d j$  are equal, it also implies  $(f i)^M = (f j)^M$  in every model  $M$ . Note, however, that  $f i \doteq f j$  cannot be derived because it is not a well-formed formula. It cannot be well-formed because it only makes sense in the presence of the axiom  $i \doteq j$ . If equations between terms of the same depth but different sorts were allowed, and if equations in this broader sense were forbidden to occur in the axioms of a theory, the completeness result would still hold.

*Implementation* It follows that a Gentzen style theorem prover of FOL can be turned into one for DFOL, if its syntax is extended to support DFOL signatures. In the case of the existing implementation of FOL in LF, which is part of the Twelf distribution, only rules for equality need to be added.

Completeness results for resolution based proving as in Vampire ([RV02]) require additional work. The transformation into conjunctive normal form is as for FOL with the exception of skolemization, which transforms

$$\lambda x_1:S_1. \dots \lambda x_n:S_n. \exists x : S. G \text{ to } \lambda x_1:S_1. \dots \lambda x_n:S_n. (\lambda x:S. G)(fx_1 \dots x_n).$$

Here  $x_1, \dots, x_n$  also contain the free variables of  $S$ , and the substitution must also operate on  $S$ . If paramodulation (see for example [NR01]) is used to replace a subterm  $s$  of  $F$  with  $t$ , both the sorts of  $s$  and  $t$  as well as  $s$  and  $t$  themselves must be unified (see [PP03] for unification with dependent types in LF).

## 5 Examples

In the examples, we use infix notation for some symbols without giving the corresponding Twelf declarations. And we use the implicit arguments notation of Twelf, i.e., if a free variable  $X$  which, due to the context, must have type  $T$  occurs in a declaration, it is assumed that the type is prefixed with  $\Pi X:T$ . If such a free variable occurs in an axiom, we assume an implicit universal quantification.

*Categories* Let  $Cat$  be the following theory of depth 1 (where we leave a blank line between signature and axioms)

$$\begin{aligned} Ob &: \mathbf{S}. \\ Mor &: \Pi A, B : Ob. \mathbf{S}. \\ id &: \Pi A : Ob. Mor A A. \\ \circ &: Mor A B \rightarrow Mor B C \rightarrow Mor A C. \end{aligned}$$

$$\begin{aligned} \forall f : Mor A B. f \doteq f \circ id B \\ \forall f : Mor A B. f \doteq id A \circ f \\ \forall f : Mor A B, g : Mor B C, h : Mor C D. (f \circ g) \circ h \doteq f \circ (g \circ h) \end{aligned}$$

Then  $Mod(Cat)$  is the category of small categories. For example the formula  $\lambda X : Ob. \forall A : Ob. \exists f : Mor A X. \forall g : Mor A X. f \doteq g$  expresses the property that  $X : Ob$  is a terminal element.

The free category over a generating graph  $G$  can be obtained by applying Thm. 3 to  $Cat$  where  $A$  contains declarations  $N : Ob$  for every node  $N$  and  $E : Mor N N'$  for every edge  $E = (N, N')$  of  $G$ . Note that the theorem also allows to impose specific equalities, e.g., an axiom  $E = E_1 \circ E_2$  if the composition of  $E_1$  and  $E_2$  is already part of  $G$ .

A theory extension consists of additional declarations and additional axioms; it is simple, if it does not add sort declarations. Let  $2Cat$  be the following extension of  $Cat$

$$\begin{aligned} 2cell &: \Pi f, g : Mor A B. \mathbf{S}. \\ Id_2 &: \Pi f : Mor A B. 2cell f f. \\ \circ_{vert} &: \Pi f, g, h : Mor A B. 2cell f g \rightarrow 2cell g h \rightarrow 2cell f h. \\ \circ_{hor} &: \Pi f, g : Mor A B. \Pi f', g' : Mor B C. \\ &\quad 2cell f g \rightarrow 2cell f' g' \rightarrow 2cell f \circ f' g \circ g'. \end{aligned}$$

If we also add appropriate axioms,  $Mod(2Cat)$  becomes the category of small 2-categories. Bi-categories can be specified similarly, e.g., using the axiom  $\forall f : Mor A B, g : Mor B C, h : Mor C D, k, l : Mor A D.$

$$\begin{aligned} k \doteq (f \circ g) \circ h \wedge l \doteq f \circ (g \circ h) &\Rightarrow \exists \alpha : 2cell k l. \exists \beta : 2cell l k. \\ \alpha \circ_{vert} \beta \doteq Id_2 k \wedge \beta \circ_{vert} \alpha \doteq Id_2 l. \end{aligned}$$

Under the Curry-Howard-Tait correspondence, a 2-category corresponds to a logic with formulas, proofs and rewrites. If a simple extension of  $2Cat$  declares function symbols for connectives, proof rules and conditional rewrite rules of a logic, Thm. 3 yields its free 2-category of proofs.

Let  $OCat$  be the extension of  $Cat$  with

$$\rightsquigarrow : Mor A B \rightarrow Mor A B \rightarrow o.$$

$$\begin{aligned} \forall f : Mor A B. f \rightsquigarrow f \\ \forall f, g, h : Mor A B. (f \rightsquigarrow g \wedge g \rightsquigarrow h \Rightarrow f \rightsquigarrow h) \end{aligned}$$

where we interpret  $\rightsquigarrow$  as rewritability between morphisms. Let  $K$  be a simple extension of  $OCat$ , and let  $K'$  be as  $K$  but with the additional axioms

$$\forall X_1 : S_1 \dots X_m' : S_m.$$

$X_1 \rightsquigarrow X'_1 \wedge \dots \wedge X_m \rightsquigarrow X'_m \Rightarrow f X_1 \dots X_m \rightsquigarrow f X'_1 \dots X'_m$   
for every function symbol  $f$  of depth 1. We call  $Mod(K')$  the category of small order-enriched  $K$ -categories. The added axioms in  $K'$  are simply the congruence conditions for all function symbols with respect to rewriting. The arising axiomatization of rewriting is the same as in rewriting logic (see [BM03], which also allows frozen arguments).

In particular, this yields free order-enriched  $K$ -categories over Horn theories, and a complete axiomatization of this category. This allows a succinct view of logics under the Curry-Howard-Tait correspondence.<sup>10</sup>

*Linear Algebra* Let  $\Sigma$  be the following signature of depth 1

```

 $N : \mathbf{S}.$ 
 $one : N$ 
 $succ : N \rightarrow N.$ 
 $Mat : N \rightarrow N \rightarrow \mathbf{S}.$ 
 $0 : R.$ 
 $1 : R.$ 
 $RowAppend : Mat m one \rightarrow R \rightarrow Mat succ m one.$ 
 $ColAppend : Mat m n \rightarrow Mat m one \rightarrow Mat m succ n.$ 
 $E : Mat m m.$ 
 $+ : Mat m n \rightarrow Mat m n \rightarrow Mat m n.$ 
 $- : Mat m n \rightarrow Mat m n \rightarrow Mat m n.$ 
 $\cdot : Mat l m \rightarrow Mat m n \rightarrow Mat l n.$ 
 $det : Mat m m \rightarrow R.$ 
 $inv : Mat m m \rightarrow o.$ 
 $eigenvalue : Mat m m \rightarrow R \rightarrow o.$ 

```

(where we use  $R$  to abbreviate  $Mat\ one\ one$ ). Clearly,  $\Sigma$  can be used to axiomatize linear algebra over any ring that can be axiomatized in first-order logic. Examples for axioms are

$\forall M : Mat m m. (inv M \Leftrightarrow \neg det M \doteq 0)$  and  
 $\forall M : Mat m n, r : R. (eigenvalue M r \Leftrightarrow \exists v : Mat n one. M \cdot v \doteq v \cdot r)$   
(with the usual abbreviation  $\Leftrightarrow$ ).

## 6 Conclusion

We have introduced an extension of FOL with dependent types such that the generalization of definitions, results and implementations is very natural. The formulation of DFOL as an institution allows to apply the established institution-independent results (see the book [?]). The formulation of the syntax in LF

---

<sup>10</sup> Having a simple meta-language that supports dependent types while allowing axiomatization and free models was the original motivation to introduce DFOL. The idea for these specifications and the introduction of DFOL is due to Till Mossakowski and discussions with him and others.

immediately yields implementations of type checking, proof checking and simple theorem proving.

Of course, DFOL does not permit anything that has not been possible before. For example, category theory has been specified in Coq ([HS98]) or simply using FOL.<sup>11</sup> However, the simple model theory and the performance of automated theorem provers provide good arguments to stick to FOL if possible. The research presented here is targeted at those situations where specifications in (partial) FOL are desirable but awkward. In the examples, we demonstrated that only one or two dependent sort constructors can allow an elegant specification of a mathematical theory that would be awkward in FOL, but for which tools like Coq are far more powerful than necessary.

It can be argued that signatures of depth greater than 1 or 2 are not interesting. And in fact, the general case was not our original goal. But it turned out that the step from depth 0 to depth 1 is already almost as complex as the induction step for the general case so that no simplifications are to be expected from restricting the depth.

Although further work is needed (e.g., on resolution or Craig interpolation), it turned out that crucial classical results can be extended to DFOL. Free models make DFOL valuable as an algebraic specification language, and we plan to integrate it into CASL ([BM04]). And the axiomatization indicates that existing provers can be extended for DFOL. The FOL encoding in Twelf can be adapted easily so that both pure LF and the Twelf meta-theorem prover ([SP96]) can be applied.

## References

- [BC04] Y. Bertot and P. Castéran. *Coq’Art: The Calculus of Inductive Constructions*. Springer, 2004.
- [BM03] R. Bruni and J. Meseguer. Generalized rewrite theories. In *Proceedings of ICALP ’03*. Springer, 2003.
- [BM04] Michel Bidoit and Peter D. Mosses. *Casl User Manual*. LNCS 2900 (IFIP Series). Springer, 2004.
- [CAB<sup>+</sup>86] R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.
- [Car86] J. Cartmell. Generalized algebraic theories and contextual category. *Annals of Pure and Applied Logic*, 32:209–243, 1986.
- [CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4, pages 65–89, 1996.
- [Dyb95] P. Dybjer. Internal type theory. In *TYPES*, pages 120–134, 1995.

---

<sup>11</sup> DFOL cannot in general be encoded in partial many-sorted FOL since there may be more universes in a DFOL model than there are closed sort terms in the language. An encoding in FOL is straightforward and can provide an alternative completeness result but is very awkward.

- [Gal86] J. Gallier. *Foundations of Automatic Theorem Proving*. Wiley, 1986.
- [GWM<sup>+</sup>93] J. Goguen, Timothy Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.
- [Hen49] L. Henkin. The completeness of the first-order functional calculus. *Journal of Symbolic Logic*, 14:159–166, 1949.
- [Hof94] M. Hofmann. On the Interpretation of Type Theory in Locally Cartesian Closed Categories. In *CSL*, pages 427–441. Springer, 1994.
- [HS98] G. Huet and A. Saibi. Constructive category theory. In G. Plotkin, C. Sterling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.
- [HST94] R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.
- [Mak] M. Makkai. First order logic with dependent sorts (FOLDS). Unpublished.
- [ML74] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In *Proceedings of the '73 Logic Colloquium*. North-Holland, 1974.
- [NR01] R. Nieuwenhuis and A. Rubio. Paramodulation-Based theorem proving. In *Handbook of Automated Reasoning*, pages 371–443. Elsevier Science Publishers, 2001.
- [ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, 1992. Springer.
- [OS97] S. Owre and N. Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, SRI International, 1997.
- [Pfe01] F. Pfenning. Logical frameworks. In *Handbook of automated reasoning*, pages 1063–1147. Elsevier, 2001.
- [PP03] B. Pientka and F. Pfenning. Optimizing higher-order pattern unification. In *19th International Conference on Automated Deduction*, pages 473–487. Springer, 2003.
- [PS99] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.
- [RV02] A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Communications*, 15:91–110, 2002.
- [See84] R. Seely. Locally cartesian closed categories and type theory. *Math. Proc. Cambridge Philos. Soc.*, 95:33–48, 1984.
- [SP96] C. Schürmann and F. Pfenning. Automated theorem proving in a simple meta-logic for LF. In C. Kirchner and H. Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction*, pages 286–300. Springer, 1996.
- [Vir96] R. Virga. Higher-order superposition for dependent types. In H. Ganzinger, editor, *Proceedings of the 7th International Conference on Rewriting Techniques and Applications*, pages 123–137. Springer, 1996.

# Translating a Dependently-Typed Logic to First-Order Logic

Kristina Sojakova, Florian Rabe

Jacobs University Bremen

**Abstract.** DFOL is a logic that extends first-order logic with dependent types. We give a translation from DFOL to FOL formalized as an institution comorphism and show that it admits the model expansion property. This property together with the borrowing theorem implies the soundness of borrowing — a result that enables us to reason about entailment in DFOL by using automated tools for FOL. In addition, the translation permits us to deduce properties of DFOL such as completeness, compactness, and existence of free models from the corresponding properties of FOL, and to regard DFOL as a fragment of FOL. We give an example that shows how problems about DFOL can be solved by using the automated FOL prover Vampire. Future work will focus on the integration of the translation into the specification and translation tool HeTS.

## 1 Introduction and Related Work

Dependent type theory, DTT, ([ML75]) provides a very elegant language for many applications ([HHP93,NPS90]). However, its definition is much more involved than that of simple type theory because all well-formed terms, types, and their equalities must be defined in a single joint induction. Several quite complex model classes, mainly related to locally cartesian closed categories, have been studied to provide a model theory for DTT (see [Pit00] for an overview).

Many of the complications disappear if dependently-typed extensions of first-order logic are considered, i.e., systems that have dependent types, but no (simple or dependent) function types. Such systems were investigated in [Mak97], [Rab06], and [Bel08]. They provide very elegant axiomatizations of many important mathematical theories such as those of categories or linear algebra while retaining completeness with respect to straightforward set-theoretic models.

However, these systems are of relatively little practical use because no automated reasoning tools, let alone efficient ones, are available. Therefore, our motivation is to translate one of these systems into first-order logic, FOL. Such a translation would translate a proof obligation to FOL and discharge it by calling existing FOL provers. This is called borrowing ([CM93]).

In principle, there are two ways how to establish the soundness of borrowing: proof-theoretically by translating the obtained proof back to the original logic, or model-theoretically by exhibiting a model-translation between the two logics. Proof-theoretical translations of languages with dependent types have been

used in [JM93] to translate parts of DTT to simple type theory, in [Urb03] to translate Mizar ([TB85]) into FOL, and in Scunak [Bro06] to translate parts of DTT into FOL. The Scunak translation is only partial as for example the translation of lambda expressions is omitted. Similar partial translations, but in the simply-typed case, are used in Omega ([BCF<sup>+</sup>97]), Leo-II ([BPTF07]) and in the sledgehammer tactic of Isabelle ([Pau94]). If the FOL prover succeeds, the reconstruction of the FOL proof term is possible in practice but somewhat tricky: For example, sledgehammer uses the output of a strong prover to guide a second, weaker prover, from whose output the proof term is reconstructed. In the cases of Mizar and Scunak, it is not done at all. Furthermore, the more complex the translation of proof goals is, the more difficult it becomes to translate the FOL proof term back into the original logic.

Here we take the model-theoretic approach and formulate a translation from the system introduced in [Rab06] to FOL within the framework of institutions ([GB92]). Mathematically, our main results can be summarized as follows. We use the institution DFOL as given in [Rab06] and give an institution comorphism from DFOL into FOL. Every DFOL-signature is translated to a FOL-theory whose axioms are used to express the typing properties of the translated symbols. The signature translation uses an  $n+1$ -ary FOL-predicate  $P_s$  for every dependent type constructor  $s$  with  $n$  arguments. Then the formulas quantifying over  $x$  of type  $s(t_1, \dots, t_n)$  can be translated by relativizing (see [Obe62]) using the predicate  $P_s(t_1, \dots, t_n, x)$ . Finally, we show that this comorphism admits model expansion. Using the borrowing theorem ([CM93]), this yields the soundness of the translation.

Thus, we provide a simple way to write problems in the conveniently expressive DFOL syntax and solve them by calling FOL theorem provers. It is also possible to extend FOL theorem provers with dependently typed input languages, or to integrate DFOL seamlessly into existing implementations of institution-based algebraic specification languages such as OBJ ([GWM<sup>+</sup>93]) and CASL ([ABK<sup>+</sup>02]). Finally, our result provides easier proofs of the free model and completeness theorems given in [Rab06].

## 2 Definitions

We now present some definitions necessary for our work. We assume that the reader is familiar with the basic concepts of category theory and logic. For introduction to category theory, see [Lan98].

Using categories and functors we can define an *institution*, which is a formalization of a logical system abstracting from notions such as formulas, models, and satisfaction. Institutions structure the variety of different logics and allow us to formulate institution-independent theorems for the general theory of logic. For more on institutions, see [GB92].

**Definition 1 (Institution).** *An institution is a 4-tuple  $(\text{Sig}, \text{Sen}, \text{Mod}, \models)$  where*

- $Sig$  is a category,
- $Sen : Sig \rightarrow Set$  is a functor,
- $Mod : Sig \rightarrow Cat^{op}$  is a functor,
- $\models$  is a family of relations  $\models_{\Sigma}$  for  $\Sigma \in |Sig|$ ,  $\models_{\Sigma} \subseteq Sen(\Sigma) \times |Mod(\Sigma)|$

such that for each morphism  $\sigma : \Sigma \rightarrow \Sigma'$ , sentence  $F \in Sen(\Sigma)$ , and model  $M' \in |Mod(\Sigma')|$  we have

$$Mod(\sigma)(M') \models_{\Sigma} F \text{ iff } M' \models_{\Sigma'} Sen(\sigma)(F)$$

The category  $Sig$  is called the *category of signatures*. The morphisms in  $Sig$  are called *signature morphisms* and represent notation changes. The functor  $Sen$  assigns to each signature  $\Sigma$  a set of *sentences* over  $\Sigma$  and to each morphism  $\sigma : \Sigma \rightarrow \Sigma'$  the induced *sentence translation* along  $\sigma$ . Similarly, the functor  $Mod$  assigns to each signature  $\Sigma$  a category of *models* for  $\Sigma$  and to each morphism  $\sigma : \Sigma \rightarrow \Sigma'$  the induced *model reduction* along  $\sigma$ . For a signature  $\Sigma$ , the relation  $\models_{\Sigma}$  is called a *satisfaction relation*.

We now define what *entailment* and *theory* are in the context of institutions.

**Definition 2 (Entailment).** Let  $(Sig, Sen, Mod, \models)$  be an institution. For a fixed  $\Sigma$ , let  $T \subseteq Sen(\Sigma)$  and  $F \in Sen(\Sigma)$ . Then we say that  $T$  entails  $F$ , denoted  $T \models_{\Sigma} F$ , if for any model  $M \in |Mod(\Sigma)|$  we have that

$$\text{if } M \models_{\Sigma} G \text{ for all } G \in T \text{ then } M \models_{\Sigma} F$$

**Definition 3 (Category of theories).** Let  $I = (Sig, Sen, Mod, \models)$  be an institution. We define the category of theories of  $I$  to be the category  $Th^I$  where

- The objects are pairs  $(\Sigma, T)$ , with  $\Sigma \in |Sig|$ ,  $T \subseteq Sen(\Sigma)$
- $\sigma$  is a morphism from  $(\Sigma, T)$  to  $(\Sigma', T')$  iff  $\sigma$  is a signature morphism from  $\Sigma$  to  $\Sigma'$  in  $I$  and for each  $F \in T$  we have that  $T' \models_{\Sigma'} Sen(\sigma)(F)$

The objects in  $Th^I$  are called *theories* of  $I$ , and for each theory  $Th = (\Sigma, T)$ , the set  $T$  is called the set of *axioms* of  $Th$ . The morphisms in  $Th^I$  are called *theory morphisms*. For a theory  $(\Sigma, T)$  and a sentence  $F$  over  $\Sigma$ , we say  $(\Sigma, T) \models F$  in place of  $T \models_{\Sigma} F$ .

For a given institution  $I$ , we sometimes need to construct another institution  $I^{Th}$ , whose signatures are the theories of  $I$ . We have the following lemma.

**Lemma 1 (Institution of theories).** Let  $I = (Sig, Sen, Mod, \models)$  be an institution. Denote by  $I^{Th}$  the tuple  $(Th^I, Sen^{Th}, Mod^{Th}, \models^{Th})$  where

- $Sen^{Th}(\Sigma, T) = Sen(\Sigma)$  and  $Sen^{Th}(\sigma) = Sen(\sigma)$  for  $\sigma : (\Sigma, T) \rightarrow (\Sigma', T')$ .
- $Mod^{Th}(\Sigma, T)$  is the full subcategory of  $Mod(\Sigma)$  whose objects are those models  $M$  in  $|Mod(\Sigma)|$  for which we have  $M \models_{\Sigma} G$  whenever  $G \in T$ . For a theory morphism  $\sigma : (\Sigma, T) \rightarrow (\Sigma', T')$ ,  $Mod^{Th}(\sigma)$  is the restriction of  $Mod(\sigma)$  to  $Mod^{Th}(\Sigma', T')$ .
- $\models_{(\Sigma, T)}^{Th}$  is the restriction of  $\models_{\Sigma}$  to  $|Mod^{Th}(\Sigma, T)| \times Sen^{Th}(\Sigma, T)$ .

Then  $I^{Th}$  is an institution, called the institution of theories of  $I$ .

We are now ready to define a certain kind of translation between two institutions.

**Definition 4 (Institution comorphism).** Let  $I = (\text{Sig}^I, \text{Sen}^I, \text{Mod}^I, \models^I)$ ,  $J = (\text{Sig}^J, \text{Sen}^J, \text{Mod}^J, \models^J)$  be two institutions. An institution comorphism from  $I$  to  $J$  is a triple  $(\Phi, \alpha, \beta)$  where

- $\Phi : \text{Sig}^I \rightarrow \text{Sig}^J$  is a functor,
- $\alpha : \text{Sen}^I \rightarrow \Phi; \text{Sen}^J$  is a natural transformation,
- $\beta : \text{Mod}^I \leftarrow \Phi; \text{Mod}^J$  is a natural transformation

such that for each  $\Sigma \in |\text{Sig}^I|$ ,  $F \in \text{Sen}^I(\Sigma)$ , and  $M' \in |\text{Mod}^J(\Phi(\Sigma))|$  we have

$$\beta_\Sigma(M') \models_\Sigma^I F \quad \text{iff} \quad M' \models_{\Phi(\Sigma)}^J \alpha_\Sigma(F)$$

where  $\beta_\Sigma$  is regarded as a morphism from  $\text{Mod}^J(\Phi(\Sigma))$  to  $\text{Mod}^I(\Sigma)$  in the category  $\text{Cat}$ .

Institution comorphisms are particularly useful if they have the following property.

**Definition 5 (Model expansion property).** Let  $(\Phi, \alpha, \beta)$  be an institution comorphism from  $I$  to  $J$ . We say that the comorphism has the model expansion property if each functor  $\beta_\Sigma$  for  $\Sigma \in \text{Sig}^I$  is surjective on objects.

The following lemma is then applicable.

**Lemma 2 (Borrowing).** Let  $(\Phi, \alpha, \beta)$  be an institution comorphism from  $I^{Th}$  to  $J^{Th}$  having the model expansion property. Then for any theory  $(\Sigma, T)$  in  $I$  and a sentence  $F$  over  $\Sigma$ , we have that

$$(\Sigma, T) \models^I F \quad \text{iff} \quad \Phi(\Sigma, T) \models^J \alpha_\Sigma(F)$$

In other words, we can use the institution  $J$  to reason about theories in  $I$ . For more on borrowing, see [CM93].

### 3 DFOL and FOL as Institutions

The formal definition of a dependent type theory is typically very complex and long because both for the syntax and for the semantics a joint induction over signatures, contexts, terms, and types must be used. Therefore, in [Rab06], the syntax of DFOL is defined within the Edinburgh logical framework (LF, [HHP93]), thus saving one induction. In [Rab08], a model theory for LF is given so that both inductions can be done once and for all in the logical framework, thus permitting a very elegant and compact definition of DFOL.

Here, to be self-contained, we give the syntax directly, but omit the precise definition of well-formed expressions. Then the semantics is given by a partial interpretation function defined only for well-formed expressions. This has the advantage of making the main concepts intuitively clear while being short and precise.

### 3.1 Signatures

In DFOL, we have three base types, defined as follows:

$$\mathbf{S} : type \quad Univ : \mathbf{S} \rightarrow type \quad o : type$$

Here  $\mathbf{S}$  is the type of sorts (semantically: names of universes). The type  $Univ$  is an operator assigning to each sort the type of its terms (semantically: its universe of individuals). The type  $o$  is the type of formulas (semantically: the values *true* and *false*).

A DFOL signature consists of a finite sequence of declarations of the form

$$c : \Pi x_1 : Univ(S_1), \dots, \Pi x_n : Univ(S_n). T$$

meaning that  $c$  is a function taking  $n$  arguments of types  $S_1, \dots, S_n$  respectively, and returning an argument of type  $T$ , where  $T$  is one of the three base types. Here  $\Pi x_i : Univ(S_i)$  denotes the domain of a dependent function type, i.e.,  $x_i$  may occur in  $S_{i+1}, \dots, S_n, T$ .

When the return type of  $c$  is  $o$ , we say that  $c$  is a *predicate symbol*. Likewise, if the return type is  $\mathbf{S}$  or  $Univ(S)$ , we say that  $c$  is a *sort symbol* or a *function symbol* respectively. We abbreviate  $\Pi x : Univ(S)$  as  $\Pi x : S$  and  $\Pi x : A. B$  as  $A \rightarrow B$  if the variable  $x$  does not occur in  $B$ .

We define DFOL signatures  $\Sigma$  inductively on the number of declarations. Let  $\Sigma_k$  be a DFOL signature consisting of  $k$  declarations,  $k \geq 0$ . We define a *function* over  $\Sigma_k$  as follows:

- Any variable symbol is a function over  $\Sigma_k$
- If  $f$  in  $\Sigma_k$  is a function symbol of arity  $n$  and  $\mu_1, \dots, \mu_n$  are functions over  $\Sigma_k$ , then  $f(\mu_1, \dots, \mu_n)$  is a function over  $\Sigma_k$

If  $s$  in  $\Sigma_k$  is a sort symbol of arity  $n$  and  $\mu_1, \dots, \mu_n$  are functions over  $\Sigma_k$ , then  $s(\mu_1, \dots, \mu_n)$  is a *sort* over  $\Sigma_k$ . Similarly, if  $p$  in  $\Sigma_k$  is a predicate symbol of arity  $n$  and  $\mu_1, \dots, \mu_n$  are functions over  $\Sigma_k$ , then  $p(\mu_1, \dots, \mu_n)$  is a *predicate* over  $\Sigma_k$ . The word *term* refers to either a function, a sort, or a predicate.

Clearly, not all terms are well-formed in DFOL. A context  $\Gamma$  for a signature  $\Sigma$  in DFOL has the form  $\Gamma = x_1 : S_1, \dots, x_n : S_n$ , where  $S_1, \dots, S_n$  are sorts and  $S_i$  contains no variables except possibly  $x_1, \dots, x_{i-1}$ . Given a valid context  $\Gamma$ , a DFOL term is well-formed with respect to  $\Gamma$  only if it is well-typed in the LF type theory. For details we refer the reader to [Rab06].

Now the  $k + 1$ -th declaration has one of the following forms:

- $s : \Pi x_1 : S_1, \dots, \Pi x_n : S_n. \mathbf{S}$   
where  $S_1, \dots, S_n$  are sorts over  $\Sigma_k$  and  $S_i$  contains no variables except possibly  $x_1, \dots, x_{i-1}$ . We say that  $s$  is a sort symbol.
- $f : \Pi x_1 : S_1, \dots, \Pi x_n : S_n. S_{n+1}$   
where  $S_1, \dots, S_{n+1}$  are sorts over  $\Sigma_k$  and  $S_i$  contains no variables except possibly  $x_1, \dots, x_{i-1}$ .

- $p : \Pi x_1 : S_1, \dots, \Pi x_n : S_n. o$   
where  $S_1, \dots, S_n$  are sorts over  $\Sigma_k$  and  $S_i$  contains no variables except possibly  $x_1, \dots, x_{i-1}$ . We say that  $p$  is a predicate symbol.

As with terms, the declaration must be a well-typed according to the rules of LF.

*Running example* The theory of categories has the following DFOL signature  
 $Ob : \mathbf{S}$

$$\begin{aligned} Mor &: Ob \rightarrow Ob \rightarrow \mathbf{S} \\ id &: \Pi A : Ob. Mor(A, A) \\ \circ &: \Pi A, B, C : Ob. Mor(A, B) \rightarrow Mor(B, C) \rightarrow Mor(A, C) \\ term &: Ob \rightarrow o \\ isom &: Ob \rightarrow Ob \rightarrow o \end{aligned}$$

For simplicity, we declare the signature model morphisms in DFOL to just be the identity morphisms.

### 3.2 Sentences

The set of DFOL formulas over a signature  $\Sigma$  can be described as follows:

- If  $P$  is a predicate over  $\Sigma$ , then  $P$  is a  $\Sigma$ -formula
- If  $\mu_1, \mu_2$  are functions over  $\Sigma$ , then  $\mu_1 \doteq \mu_2$  is a  $\Sigma$ -formula
- If  $F$  is a  $\Sigma$ -formula, then  $\neg F$  is a  $\Sigma$ -formula
- If  $F, G$  are  $\Sigma$ -formulas, then  $F \wedge G$ ,  $F \vee G$ , and  $F \Rightarrow G$  are  $\Sigma$ -formulas
- If  $F$  is a  $\Sigma$ -formula and  $S$  is a sort term over  $\Sigma$ , then  $\forall x : S. F$  is a  $\Sigma$ -formula
- If  $F$  is a  $\Sigma$ -formula and  $S$  is a sort term over  $\Sigma$ , then  $\exists x : S. F$  is a  $\Sigma$ -formula

Closed and atomic formulas are defined in the obvious way analogous to first-order logic. As with terms, DFOL formulas are well-formed only if they are well-typed in the LF type theory. For a precise definition, see [Rab06].

*Running example* We have the following axioms for the theory of categories, with equivalence defined as usual

$$\begin{aligned} I1 &: \forall A, B : Ob. \forall f : Mor(A, B). id(A) \circ f \doteq f \\ I2 &: \forall A, B : Ob. \forall f : Mor(B, A). f \circ id(A) \doteq f \\ A1 &: \forall A, B, C, D : Ob. \forall f : Mor(A, B). \forall g : Mor(B, C). \forall h : Mor(C, D). \\ &\quad f \circ (g \circ h) \doteq (f \circ g) \circ h \\ D1 &: \forall A : Ob. (term(A) \iff \forall B : Ob. \exists f : Mor(B, A). \forall g : Mor(B, A). f \doteq g) \\ D2 &: \forall A, B : Ob. (isom(A, B) \iff \exists f : Mor(A, B). \exists g : Mor(B, A). \\ &\quad (f \circ g \doteq id(A) \wedge g \circ f \doteq id(B))) \end{aligned}$$

### 3.3 Models

A model of a DFOL signature  $\Sigma$  is an interpretation function  $I$ . Since the declaration of a symbol may depend on symbols declared before, we define  $I$  inductively on the number of declarations.

Suppose  $I$  is defined for the first  $k$  declarations,  $k \geq 0$ . An *assignment function*

$\varphi$  for  $I$  is a function mapping each variable to an element of any set defined by  $I$  as an interpretation of a sort symbol.

Let  $\mu$  be a term over  $\Sigma$ . We define the *interpretation* of  $\mu$  induced by  $\varphi$  to be  $I_\varphi(\mu)$ , where  $I_\varphi$  is given by:

- $I_\varphi(x) = \varphi(x)$  for any variable  $x$
- $I_\varphi(d(\mu_1, \dots, \mu_k)) = d^I(I_\varphi(\mu_1), \dots, I_\varphi(\mu_k))$  for a sort, predicate, or function symbol  $d$  if
  - each of the interpretations  $I_\varphi(\mu_1), \dots, I_\varphi(\mu_n)$  exists and
  - $d^I$  is defined for the tuple  $(I_\varphi(\mu_1), \dots, I_\varphi(\mu_n))$
Otherwise we say  $I_\varphi(d(\mu_1, \dots, \mu_k))$  does not exist.

Given a valid context  $\Gamma = x_1 : S_1, \dots, x_n : S_n$  and an assignment function  $\varphi$ , we say that  $\varphi$  is an *assignment function for  $\Gamma$*  if for each  $i$  we have that  $I_\varphi(S_i)$  exists and  $\varphi(x_i) \in I_\varphi(S_i)$ . From now on, we will only talk about assignment functions for a context; the general definition was introduced only to avoid some technical difficulties.

Now the  $k + 1$ -st declaration has one of the following forms:

- $s : \Pi x_1 : S_1, \dots, \Pi x_n : S_n. \mathbf{S}$

Let  $\varphi$  be any assignment function for the context  $\Gamma = x_1 : S_1, \dots, x_n : S_n$ . Then

$$s^I(\varphi(x_1), \dots, \varphi(x_n)) \text{ is a (possibly empty) set}$$

disjoint from any other set defined by  $I$  as an interpretation of a sort symbol.

- $f : \Pi x_1 : S_1, \dots, \Pi x_n : S_n. S$

Let  $\varphi$  be any assignment function for the context  $\Gamma = x_1 : S_1, \dots, x_n : S_n$ . Then

$$f^I(\varphi(x_1), \dots, \varphi(x_n)) \in I_\varphi(S)$$

- $p$  is a predicate symbol,  $p : \Pi x_1 : S_1, \dots, \Pi x_n : S_n. o$

Let  $\varphi$  be any assignment function for the context  $\Gamma = x_1 : S_1, \dots, x_n : S_n$ . Then

$$p^I(\varphi(x_1), \dots, \varphi(x_n)) \in \{\text{true}, \text{false}\}$$

*Running example* An example model  $I$  for the signature of categories is given by any small category  $C$ . Then we have  $Ob^I = |C|$ ,  $Mor^I(A, B) = C(A, B)$ , and the obvious interpretations for composition and identity. Furthermore, we can put  $term^I(A) = \text{true}$  iff  $A$  is a terminal element and  $isom^I(A, B) = \text{true}$  iff  $A$  and  $B$  are isomorphic.

### 3.4 Satisfaction relation

To define the satisfaction relation, we first define the interpretation of formulas. Let  $\Sigma$  be a DFOL signature,  $I$  be a DFOL model for  $\Sigma$ , and  $\Gamma$  be a valid context over  $\Sigma$ . Furthermore, let  $\varphi$  be an assignment function for  $\Gamma$  and  $F$  be a well-formed DFOL formula for  $\Gamma$ . Then we define  $I_\varphi(F)$  recursively on the structure of  $F$ :

- $F$  is a predicate. Then  $I_\varphi(F)$  is true if and only if  $p^I(I_\varphi(\mu_1), \dots, I_\varphi(\mu_n)) = \text{true}$ .
- $F$  is of the form  $\mu_1 \doteq \mu_2$ . Then  $I_\varphi(F)$  is true if and only if  $I_\varphi(\mu_1) = I_\varphi(\mu_2)$ .
- $F$  is of the form  $\neg G$ . Then  $I_\varphi(F)$  is true if and only if  $I_\varphi(G)$  is false.
- $F$  is of the form  $F_1 \wedge F_2$ . Then  $I_\varphi(F)$  is true if and only if both  $I_\varphi(F_1)$  and  $I_\varphi(F_2)$  are true.
- $F$  is of the form  $F_1 \vee F_2$ . Then  $I_\varphi(F)$  is true if and only if  $I_\varphi(F_1)$  is true or  $I_\varphi(F_2)$  is true.
- $F$  is of the form  $F_1 \implies F_2$ . Then  $I_\varphi(F)$  is true if and only if  $I_\varphi(F_1)$  is false or  $I_\varphi(F_2)$  is true.
- $F$  is of the form  $\exists x : S. G$ . Then  $I_\varphi(F)$  is true if and only if  $I_{\varphi[x/a]}(G)$  is true for some  $a \in I_\varphi(S)$ .
- $F$  is of the form  $\forall x : S. G$ . Then  $I_\varphi(F)$  is true if and only if  $I_{\varphi[x/a]}(G)$  is true for any  $a \in I_\varphi(S)$ .

Now if  $F$  is in fact a closed formula, its interpretation is independent of  $\varphi$ . Hence, we define that  $I$  satisfies  $F$  if and only if  $I_\varphi(F)$  is true for some  $\varphi$ .

*Running example* It is easy to see that the example model for the signature of categories satisfies the axioms given in section 3.2.

Putting our previous definitions together, we have the following lemma.

**Lemma 3.**  $DFOL = (\text{Sig}, \text{Sen}, \text{Mod}, \models)$  is an institution.

The FOL institution is then obtained from DFOL by restricting the signatures to contain a unique sort symbol, having arity 0. Any other symbols are either function or predicate symbols. (Technically, this does not yield FOL because DFOL permits empty universes. But our FOL signatures will always have a nullary function symbol so that this does not constitute a problem). A FOL model is then denoted as  $(U, I)$ , where  $I$  is the interpretation function and  $U$  is the universe corresponding to the unique sort symbol.

## 4 Translation of DFOL to FOL

The main idea of the translation is to associate with each  $n$ -ary sort symbol in DFOL an  $n+1$ -ary predicate in FOL and relativize the universal and existential quantifiers (the technique of relativization was first introduced by Oberschelp in [Obe62]).

Formally, the translation will be given as an institution comorphism from  $DFOL$  to  $FOL^{Th}$ . We specify a functor  $\Phi$ , mapping DFOL signatures to FOL theories and DFOL signature morphisms to FOL theory morphisms. For each DFOL signature  $\Sigma$ , we give a function  $\alpha_\Sigma$  mapping DFOL sentences over  $\Sigma$  to FOL sentences over the translated signature  $\Phi(\Sigma)$ , and show that the family of functions  $\alpha_\Sigma$  defines a natural transformation. Similarly, for each DFOL signature  $\Sigma$  we give a functor  $\beta_\Sigma$  mapping FOL models for the translated signature  $\Phi(\Sigma)$  to DFOL models for  $\Sigma$ , and show that the family of functors  $\beta_\Sigma$  defines a natural transformation. Finally, we prove the satisfaction condition for  $(\Phi, \alpha, \beta)$  and show that the comorphism has the model expansion property.

**Definition 6 (Signature translation).** Let  $\Sigma$  be a DFOL signature. We define  $\Phi(\Sigma)$  to be the FOL theory  $(\Sigma', T')$ , where  $\Sigma'$  and  $T'$  are specified as follows.

$\Sigma'$  contains:

- an  $n$ -ary function symbol  $f$  for each  $n$ -ary function symbol  $f$  in  $\Sigma$ ,
- an  $n$ -ary predicate symbol  $p$  for each  $n$ -ary predicate symbol  $p$  in  $\Sigma$ ,
- an  $n + 1$ -ary predicate symbol  $s$  for each  $n$ -ary sort symbol  $s$  in  $\Sigma$ ,
- a special constant symbol  $\perp$ , different from any of the above symbols,
- no other symbols besides the above

$T'$  contains:

S1. Axioms ensuring that no element can belong to the universe of more than one sort. For any two sort symbols  $s_1, s_2$  with  $s_1$  different from  $s_2$ , we have the axiom

$$\forall x_1, \dots, x_n, y_1, \dots, y_m, z. (s_1(x_1, \dots, x_n, z) \implies \neg s_2(y_1, \dots, y_m, z))$$

and for each sort symbol  $s_1$  we have the axiom

$$\begin{aligned} \forall x_1, \dots, x_n, y_1, \dots, y_n, z. & (s_1(x_1, \dots, x_n, z) \wedge s_1(y_1, \dots, y_n, z) \\ & \implies x_1 \doteq y_1 \wedge \dots \wedge x_n \doteq y_n) \end{aligned}$$

S2. An axiom ensuring that each element different from  $\perp$  belongs to the universe of at least one sort. If  $s_1, \dots, s_k$  are the sort symbols, then we have the axiom

$$\begin{aligned} \forall y. (\neg y \doteq \perp \implies \exists x_1, \dots, x_{n_1}. s_1(x_1, \dots, x_{n_1}, y) \vee \dots \vee \\ \exists x_1, \dots, x_{n_k}. s_k(x_1, \dots, x_{n_k}, y)) \end{aligned}$$

S3. Axioms ensuring that the special symbol  $\perp$  is not contained in the universe of any sort. For each sort symbol  $s$ , we have the axiom

$$\forall x_1, \dots, x_n. \neg s(x_1, \dots, x_n, \perp)$$

S4. Axioms ensuring that if the arguments to a sort constructor are not of the correct types, the resulting sort has an empty universe. For each sort symbol  $s : \prod x_1 : s_1(\mu_1^1, \dots, \mu_{k_1}^1), \dots, \prod x_n : s_n(\mu_1^n, \dots, \mu_{k_n}^n)$ .  $s$ , we have the axiom

$$\begin{aligned} \forall x_1, \dots, x_n. & (\neg s_1(\mu_1^1, \dots, \mu_{k_1}^1, x_1) \vee \dots \vee \neg s_n(\mu_1^n, \dots, \mu_{k_n}^n, x_n) \\ & \implies \forall y. \neg s(x_1, \dots, x_n, y)) \end{aligned}$$

F1. Axioms ensuring that if the arguments to a function are of the correct types, the function returns a value of the correct type. For each function symbol  $f : \prod x_1 : s_1(\mu_1^1, \dots, \mu_{k_1}^1), \dots, \prod x_n : s_n(\mu_1^n, \dots, \mu_{k_n}^n)$ .  $s(\mu_1, \dots, \mu_k)$ , we have the axiom

$$\begin{aligned} \forall x_1, \dots, x_n. & (s_1(\mu_1^1, \dots, \mu_{k_1}^1, x_1) \wedge \dots \wedge s_n(\mu_1^n, \dots, \mu_{k_n}^n, x_n) \implies \\ & s(\mu_1, \dots, \mu_k, f(x_1, \dots, x_n))) \end{aligned}$$

*F2. Axioms ensuring that if the arguments to a function are not of the correct types, the function returns the special symbol  $\perp$ . For each function symbol  $f : \prod x_1 : s_1(\mu_1^1, \dots, \mu_{k_1}^1), \dots, \prod x_n : s_n(\mu_1^n, \dots, \mu_{k_n}^n). s(\mu_1, \dots, \mu_k)$ , we have the axiom*

$$\begin{aligned} \forall x_1, \dots, x_n. (\neg s_1(\mu_1^1, \dots, \mu_{k_1}^1, x_1) \vee \dots \vee \neg s_n(\mu_1^n, \dots, \mu_{k_n}^n, x_n) \\ \implies f(x_1, \dots, x_n) \doteq \perp) \end{aligned}$$

*P1. Axioms ensuring that if the arguments to a predicate are not of the correct types, the predicate is false. For each predicate symbol  $p : \prod x_1 : s_1(\mu_1^1, \dots, \mu_{k_1}^1), \dots, \prod x_n : s_n(\mu_1^n, \dots, \mu_{k_n}^n). o$ , we have the axiom*

$$\begin{aligned} \forall x_1, \dots, x_n. (\neg s_1(\mu_1^1, \dots, \mu_{k_1}^1, x_1) \vee \dots \vee \neg s_n(\mu_1^n, \dots, \mu_{k_n}^n, x_n) \\ \implies \neg p(x_1, \dots, x_n)) \end{aligned}$$

*N. No other axioms besides the above*

Defining  $\Phi$  on signature morphisms is trivial since by our definition the only signature morphisms in DFOL are the identity morphisms. From this it follows immediately that  $\Phi$  is a functor.

*Running example* Denote the translated signature of categories by the theory  $(\Sigma', T')$ . Then  $\Sigma'$  contains the following symbols:

- Function symbols:
  - id of arity 1
  - $\circ$  of arity 5
  - $\perp$  of arity 0
- Predicate symbols:
  - ob of arity 1
  - mor of arity 3
  - term of arity 1
  - isom of arity 2

The theory  $T'$  consists of the axioms *S1.1* up to *P1.2* in Fig.1.

**Definition 7 (Sentence translation).** Let  $\Sigma$  be a DFOL signature. We define the function  $\alpha_\Sigma$  on the set of all DFOL formulas over  $\Sigma$ . We do this recursively on the structure of the formula  $F$ :

- If  $F$  is of the form  $p(\mu_1, \dots, \mu_n)$ , we set  $\alpha_\Sigma(F) = F$
- If  $F$  is of the form  $\mu_1 \doteq \mu_2$ , we set  $\alpha_\Sigma(F) = F$
- If  $F$  is of the form  $\neg G$ , we set  $\alpha_\Sigma(F) = \neg \alpha_\Sigma(G)$
- If  $F$  is of the form  $F_1 \wedge F_2$ , we set  $\alpha_\Sigma(F) = \alpha_\Sigma(F_1) \wedge \alpha_\Sigma(F_2)$
- If  $F$  is of the form  $F_1 \vee F_2$ , we set  $\alpha_\Sigma(F) = \alpha_\Sigma(F_1) \vee \alpha_\Sigma(F_2)$
- If  $F$  is of the form  $F_1 \implies F_2$ , we set  $\alpha_\Sigma(F)$  to be the formula

$$\alpha_\Sigma(F_1) \implies \alpha_\Sigma(F_2)$$

– If  $F$  is of the form  $\forall x : s(\mu_1, \dots, \mu_n). G$ , we set  $\alpha_\Sigma(F)$  to be the formula

$$\forall x. s(\mu_1, \dots, \mu_n, x) \implies \alpha_\Sigma(G)$$

– If  $F$  is of the form  $\exists x : s(\mu_1, \dots, \mu_n). G$ , we set  $\alpha_\Sigma(F)$  to be the formula

$$\exists x. s(\mu_1, \dots, \mu_n, x) \wedge \alpha_\Sigma(G)$$

It is easy to see that  $\alpha_\Sigma$  maps closed formulas to closed formulas. Hence, we can restrict  $\alpha_\Sigma$  to the set of DFOL sentences over  $\Sigma$  to obtain our desired translation map. The naturality of  $\alpha_\Sigma$  follows immediately since the only signature morphisms in DFOL are the identity morphisms.

*Running example* The translated axioms of the theory of categories are the axioms  $I1$  up to  $D2$  in Fig.1.

$$\begin{aligned}
S1.1 &: \forall A_1, B_1, f. (\text{mor}(A_1, B_1, f) \implies \forall A_2, B_2. (\text{mor}(A_2, B_2, f) \\
&\implies A_2 \doteq A_1 \wedge B_2 \doteq B_1)) \\
S1.2 &: \forall y, B, C. (\text{ob}(y) \implies \neg\text{mor}(B, C, y)) \\
S2 &: \forall y. (\neg y \doteq \perp \implies \text{ob}(y) \vee \exists A, B. \text{mor}(A, B, y)) \\
S3.1 &: \neg\text{ob}(\perp) \\
S3.2 &: \forall A, B. \neg\text{mor}(A, B, \perp) \\
S4 &: \forall A, B. ((\neg\text{ob}(A) \vee \neg\text{ob}(B)) \implies \forall f. \neg\text{mor}(A, B, f)) \\
F1.1 &: \forall A. (\text{ob}(A) \implies \text{mor}(A, A, \text{id}(A))) \\
F1.2 &: \forall A, B, C, f, g. (\text{ob}(A) \wedge \text{ob}(B) \wedge \text{ob}(C) \wedge \text{mor}(A, B, f) \wedge \text{mor}(B, C, g) \implies \\
&\text{mor}(A, C, f; g)) \\
F2.1 &: \forall A. (\neg\text{ob}(A) \implies \text{id}(A) \doteq \perp) \\
F2.2 &: \forall A, B, C, f, g. (\neg\text{ob}(A) \vee \neg\text{ob}(B) \vee \neg\text{ob}(C) \vee \neg\text{mor}(A, B, f) \vee \\
&\neg\text{mor}(B, C, g) \implies f; g \doteq \perp) \\
P1.1 &: \forall A. (\neg\text{ob}(A) \implies \neg\text{term}(A)) \\
P1.2 &: \forall A, B. (\neg\text{ob}(A) \vee \neg\text{ob}(B) \implies \neg\text{isom}(A, B)) \\
I1 &: \forall A, B, f. (\text{ob}(A) \wedge \text{ob}(B) \wedge \text{mor}(A, B, f) \implies \text{id}(A); f \doteq f) \\
I2 &: \forall A, B, f. (\text{ob}(A) \wedge \text{ob}(B) \wedge \text{mor}(B, A, f) \implies f; \text{id}(A) \doteq f) \\
A1 &: \forall A, B, C, D, f, g, h. (\text{ob}(A) \wedge \text{ob}(B) \wedge \text{ob}(C) \wedge \text{ob}(D) \wedge \text{mor}(A, B, f) \wedge \\
&\text{mor}(B, C, g) \wedge \text{mor}(C, D, h) \implies f; (g; h) \doteq (f; g); h) \\
D1 &: \forall A. (\text{ob}(A) \implies (\text{term}(A) \iff \forall B. (\text{ob}(B) \implies \exists f. (\text{mor}(B, A, f) \wedge \\
&\forall g. (\text{mor}(B, A, g) \implies f \doteq g)))) \\
D2 &: \forall A, B. (\text{ob}(A) \wedge \text{obj}(B) \implies (\text{isom}(A, B) \iff \exists f, g. (\text{mor}(A, B, f) \wedge \\
&\text{mor}(B, A, g) \wedge f; g \doteq \text{id}(A) \wedge g; f \doteq \text{id}(B)))) \\
\end{aligned}$$

$$\text{Conjecture} : \forall A, B. (\text{ob}(A) \wedge \text{obj}(B) \wedge \text{term}(A) \wedge \text{term}(B) \implies \text{isom}(A, B))$$

**Fig. 1.** Translation of the running example

**Definition 8 (Model reduction).** Let  $\Sigma$  be a DFOL signature and  $M = (U, I)$  be a FOL model for  $\Phi(\Sigma)$ . We define the translated DFOL model  $\beta_\Sigma(M)$  for  $\Sigma$

to be the interpretation function  $J$ , defined inductively on the number of declarations in  $\Sigma$ .

Suppose  $J$  is defined for the first  $k$  symbols in  $\Sigma$ ,  $k \geq 0$ . Then the  $(k+1)$ -st declaration has one of the following forms:

- $s : \Pi x_1 : S_1, \dots, \Pi x_n : S_n. S$

Let  $\varphi$  be any assignment function for the context  $\Gamma = x_1 : S_1, \dots, x_n : S_n$ . We set

$$s^J(\varphi(x_1), \dots, \varphi(x_n)) = \{u \in U \mid s^I(\varphi(x_1), \dots, \varphi(x_n), u)\}$$

- $f : \Pi x_1 : S_1, \dots, \Pi x_n : S_n. S$

Let  $\varphi$  be any assignment function for the context  $\Gamma = x_1 : S_1, \dots, x_n : S_n$ . We set

$$f^J(\varphi(x_1), \dots, \varphi(x_n)) = f^I(\varphi(x_1), \dots, \varphi(x_n))$$

- $p$  is a predicate symbol,  $c : \Pi x_1 : S_1, \dots, \Pi x_n : S_n. o$

Let  $\varphi$  be any assignment function for the context  $\Gamma = x_1 : S_1, \dots, x_n : S_n$ . We set

$$p^J(\varphi(x_1), \dots, \varphi(x_n)) \quad \text{iff} \quad p^I(\varphi(x_1), \dots, \varphi(x_n))$$

We note here how the axioms introduced earlier are needed to ensure that  $J$  is indeed a DFOL model for  $\Sigma$ . We now turn to the proof of the satisfaction condition.

**Theorem 1 (Satisfaction condition).**  $(\Phi, \alpha, \beta)$  is an institution comorphism.

*Proof.* We have already shown that  $\Phi$  is a functor and  $\alpha, \beta$  are natural transformations. It remains to show that the satisfaction condition holds.

Let  $\Sigma$  be a DFOL signature,  $\Gamma$  be a valid context for  $\Sigma$ ,  $\varphi$  be an assignment function for  $\Gamma$ , and  $F$  be a well-formed DFOL sentence for  $\Gamma$ . Furthermore, let  $M = (U, I)$  be a FOL model for the translated signature  $\Phi(\Sigma)$ , and  $J$  be the translated model  $\beta_\Sigma(M)$ . We first observe the following two facts:

- $\varphi$  is also an assignment function for  $M$
- if  $\mu$  is a well-formed function term for  $\Gamma$ , then  $J_\varphi(\mu) = I_\varphi(\mu)$

Both of these facts follow directly from the construction of  $J$ . We now show that we have

$$J_\varphi(F) \quad \text{iff} \quad I_\varphi(\alpha_\Sigma(F))$$

To prove the claim, we proceed recursively on the structure of  $F$ :

- $F$  is of the form  $p(\mu_1, \dots, \mu_n)$ . Then  $J_\varphi(F)$  is true if and only if  $p^J(J_\varphi(\mu_1), \dots, J_\varphi(\mu_n))$ . By the construction of  $J$ , we have

$$p^J(J_\varphi(\mu_1), \dots, J_\varphi(\mu_n)) \quad \text{iff} \quad p^I(J_\varphi(\mu_1), \dots, J_\varphi(\mu_n))$$

As noted above,  $J_\varphi(\mu_i) = I_\varphi(\mu_i)$  for each  $i$ , hence

$$p^J(J_\varphi(\mu_1), \dots, J_\varphi(\mu_n)) \text{ iff } p^I(I_\varphi(\mu_1), \dots, I_\varphi(\mu_n))$$

Thus we have  $J_\varphi(F)$  if and only if  $I_\varphi(F)$ . Since  $F = \alpha_\Sigma(F)$ , this proves the claim.

- $F$  is of the form  $\mu_1 \doteq \mu_2$ . Then  $J_\varphi(F)$  is true if and only if  $J_\varphi(\mu_1) = J_\varphi(\mu_2)$ . As noted above,  $J_\varphi(\mu_1) = I_\varphi(\mu_1)$  and  $J_\varphi(\mu_2) = I_\varphi(\mu_2)$ , hence

$$J_\varphi(\mu_1) = J_\varphi(\mu_2) \text{ iff } I_\varphi(\mu_1) = I_\varphi(\mu_2)$$

Thus we have  $J_\varphi(F)$  if and only if  $I_\varphi(F)$ . Since  $F = \alpha_\Sigma(F)$ , this proves the claim.

- $F$  is of the form  $\neg G$ . Then  $J_\varphi(F)$  is true if and only if  $J_\varphi(G)$  is false. By the induction hypothesis, we have  $J_\varphi(G)$  iff  $I_\varphi(\alpha_\Sigma(G))$ . Thus  $J_\varphi(F)$  is true if and only if  $I_\varphi(\alpha_\Sigma(G))$  is false, or equivalently

$$J_\varphi(F) \text{ iff } I_\varphi(\neg\alpha_\Sigma(G))$$

Since  $\neg\alpha_\Sigma(G) = \alpha_\Sigma(\neg G)$ , this proves the claim.

- $F$  is of the form  $F_1 \wedge F_2$ . Then  $J_\varphi(F)$  is true if and only if both  $J_\varphi(F_1)$  and  $J_\varphi(F_2)$  are true. By the induction hypothesis, we have  $J_\varphi(F_1)$  iff  $I_\varphi(\alpha_\Sigma(F_1))$  and  $J_\varphi(F_2)$  iff  $I_\varphi(\alpha_\Sigma(F_2))$ . Hence,  $J_\varphi(F)$  is true if and only if both  $I_\varphi(\alpha_\Sigma(F_1))$  and  $I_\varphi(\alpha_\Sigma(F_2))$  are true. Equivalently,

$$J_\varphi(F) \text{ iff } I_\varphi(\alpha_\Sigma(F_1) \wedge \alpha_\Sigma(F_2))$$

Since  $\alpha_\Sigma(F_1) \wedge \alpha_\Sigma(F_2) = \alpha_\Sigma(F)$ , this proves the claim.

- $F$  is of the form  $F_1 \vee F_2$ . Since  $F$  is equivalent to the formula  $\neg(\neg F_1 \wedge \neg F_2)$ , the claim follows from the previous steps.
- $F$  is of the form  $F_1 \implies F_2$ . Since  $F$  is equivalent to the formula  $\neg F_1 \vee F_2$ , the claim follows from the previous steps.
- $F$  is of the form  $\exists x : s(\mu_1, \dots, \mu_n). G$ . By definition,  $J_\varphi(F)$  is true if and only if there exists an  $a \in J_\varphi(s(\mu_1, \dots, \mu_n))$  such that  $J_{\varphi[x/a]}(G)$  is true.

Again by definition,

$$J_\varphi(s(\mu_1, \dots, \mu_n)) = s^J(J_\varphi(\mu_1), \dots, J_\varphi(\mu_n))$$

Since  $J_\varphi(\mu_i) = I_\varphi(\mu_i)$  for each  $i$ , we have

$$s^J(J_\varphi(\mu_1), \dots, J_\varphi(\mu_n)) = s^J(I_\varphi(\mu_1), \dots, I_\varphi(\mu_n))$$

By the construction of  $J$ , we have that  $a$  belongs to  $s^J(I_\varphi(\mu_1), \dots, I_\varphi(\mu_n))$  if and only if  $a$  belongs to  $U$  and  $s^I(I_\varphi(\mu_1), \dots, I_\varphi(\mu_n), a) = \text{true}$ . Now since  $\mu_i$  does not contain  $x$  for any  $i$ , we have that

$$s^I(I_\varphi(\mu_1), \dots, I_\varphi(\mu_n), a) = I_{\varphi[x/a]}(s(\mu_1, \dots, \mu_n, x))$$

Also, by the induction hypothesis we have that

$$J_{\varphi[x/a]}(G) \text{ iff } I_{\varphi[x/a]}(\alpha_\Sigma(G))$$

Combining this, we get precisely that

$$J_\varphi(F) \text{ iff } I_\varphi(\exists x. s(\mu_1, \dots, \mu_n, x) \wedge \alpha_\Sigma(G))$$

Since  $\exists x. s(\mu_1, \dots, \mu_n, x) \wedge \alpha_\Sigma(G) = \alpha_\Sigma(F)$ , this proves the claim.

- $F$  is of the form  $\forall x : s(\mu_1, \dots, \mu_n). G$ . Since  $F$  is equivalent to the formula  $\neg \exists x : s(\mu_1, \dots, \mu_n). \neg G$ , the claim follows from the previous steps.

At last, we prove the model expansion property.

**Theorem 2 (Model expansion property).** *The institution comorphism  $(\Phi, \alpha, \beta)$  has the model expansion property.*

*Proof.* Let  $\Sigma$  be a DFOL signature and  $J$  be a DFOL model for  $\Sigma$ . We construct a FOL model  $M = (U, I)$  for the translated signature  $\Phi(\Sigma)$  such that  $J = \beta_\Sigma(M)$ .

To define  $U$ , let  $s_1, \dots, s_k$  be the sort symbols of  $\Sigma$ . For  $s_i$  of arity  $n_i$ , set

$$U_i = \bigcup_{(x_1, \dots, x_{n_i})} s_i(x_1, \dots, x_{n_i})$$

where  $(x_1, \dots, x_{n_i})$  ranges through all  $n_i$ -tuples for which  $s_i$  is defined. Set

$$U = \{\perp\} \cup U_1 \cup \dots \cup U_n$$

We now define  $I$  as follows.

- Let  $p$  be a predicate symbol in  $\Sigma$ ,  $p : \Pi x_1 : S_1, \dots, \Pi x_n : S_n. o$ . Let  $\varphi$  be an assignment function for  $M$ . If  $\varphi$  is also an assignment function for the context  $\Gamma = x_1 : S_1, \dots, x_n : S_n$ , we set

$$p^I(\varphi(x_1), \dots, \varphi(x_n)) \text{ iff } p^J(\varphi(x_1), \dots, \varphi(x_n))$$

otherwise we set  $p^I(\varphi(x_1), \dots, \varphi(x_n))$  to be false.

- Let  $f$  be a function symbol in  $\Sigma$ ,  $f : \Pi x_1 : S_1, \dots, \Pi x_n : S_n. S$ . Let  $\varphi$  be an assignment function for  $M$ . If  $\varphi$  is also an assignment function for the context  $\Gamma = x_1 : S_1, \dots, x_n : S_n$ , we set

$$f^I(\varphi(x_1), \dots, \varphi(x_n)) = f^J(\varphi(x_1), \dots, \varphi(x_n))$$

otherwise we set  $f^I(\varphi(x_1), \dots, \varphi(x_n)) = \perp$ .

- Let  $s$  be a sort symbol in  $\Sigma$ ,  $s : \Pi x_1 : S_1, \dots, \Pi x_n : S_n. \mathbf{S}$ . Let  $\varphi$  be an assignment function for  $M$ . If  $\varphi$  is also an assignment function for the context  $\Gamma = x_1 : S_1, \dots, x_n : S_n$ , we set

$$s^I(\varphi(x_1), \dots, \varphi(x_n), \varphi(y)) \text{ iff } \varphi(y) \in s^J(\varphi(x_1), \dots, \varphi(x_n))$$

otherwise we set  $s^I(\varphi(x_1), \dots, \varphi(x_n), \varphi(y)) = \text{false}$ .

It is easy to see that  $M = (U, I)$  satisfies all the axioms in the translated signature  $\Phi(\Sigma)$  and that we have  $J = \beta_\Sigma(M)$ .

Hence, the institution comorphism  $(\Phi, \alpha, \beta)$  permits borrowing and we have that a DFOL theory entails a sentence if and only if the translated FOL theory entails the translated sentence.

## 5 Conclusion and Future Work

We have given an institution comorphism from a dependently-typed logic to FOL and have shown that it admits model expansion. Together with the borrowing theorem [CM93] this implies the soundness of borrowing.

This result is important for several reasons. The need for dependent types arises in several areas of mathematics such as linear algebra and category theory. DFOL provides a more natural way of formulating mathematical problems while staying close to FOL formally and intuitively. On the other hand, for FOL we have machine support in the form of automated theorem-provers and model-finders. The translation enables us to formulate a DFOL problem, translate it to FOL, and then use the known automated methods for FOL (e.g., theorem-provers such as Vampire [RV02] or SPASS [WAB<sup>+</sup>99], and model finders such as Paradox [CS03]) to find a solution.

First experiments with the translation have proved successful: For example, Vampire was able to prove instantaneously that the translation of our running example is a FOL theorem. It remains to be seen how much the encoding of type information in predicates and the addition of axioms in the translation affects the performance of FOL provers on larger theories. In the future we will integrate our translation into HeTS ([MML07]), a CASL-based application that provides a framework for the implementation of institutions and institution translations. That will provide the infrastructure to create and translate big, structured DFOL theories, and thus to apply our translation on a larger scale.

Since DFOL is defined within LF, we will also treat it as a running example for an implementation of the framework introduced in [Rab08]. That will permit to define arbitrary institutions and institution translations in LF and then incorporate these definitions into HeTS.

On the theoretical side, the translation shows that DFOL can be regarded as a fragment of FOL, which generalizes the well-known results for many-sorted first-order logic. In particular, we are able to derive properties of DFOL such as completeness, compactness, and the existence of free models immediately from the corresponding properties of FOL.

## References

- [ABK<sup>+</sup>02] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. Mosses, D. Sannella, and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 2002.
- [BCF<sup>+</sup>97] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, and V. Sorge.  $\Omega$ MEGA: Towards a mathematical assistant. In W. McCune, editor, *Proceedings of the 14th Conference on Automated Deduction*, pages 252–255. Springer, 1997.
- [Bel08] J. Belo. Dependently Sorted Logic. In M. Miculan, I. Scagnetto, and F. Honsell, editors, *TYPES 2008*, pages 33–50. Springer, 2008.
- [BPTF07] C. Benzmüller, L. Paulson, F. Theiss, and A. Fietzke. The LEO-II Project. In *Automated Reasoning Workshop*, 2007.

- [Bro06] C. Brown. Combining Type Theory and Untyped Set Theory. In N. Shankar and U. Furbach, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, pages 205–219. Springer, 2006.
- [CM93] M. Cerioli and J. Meseguer. May I Borrow Your Logic? In A. Borzyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science*, pages 342–351. Springer, 1993.
- [CS03] K. Claessen and N. Sorensson. New techniques that improve MACE-style finite model finding. In *19th International Conference on Automated Deduction (CADE-19) Workshop on Model Computation - Principles, Algorithms, Applications*, 2003.
- [GB92] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.
- [GWM<sup>+</sup>93] J. Goguen, Timothy Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.
- [HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
- [JM93] B. Jacobs and T. Melham. Translating dependent type theory into higher order logic. In M. Bezem and J. Groote, editors, *Typed Lambda Calculi and Applications*, pages 209–29, 1993.
- [Lan98] S. Mac Lane. *Categories for the Working Mathematician*. Springer, 1998.
- [Mak97] M. Makkai. First order logic with dependent sorts (FOLDS), 1997. Unpublished.
- [ML75] P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the Logic Colloquium 1973*, pages 73–118, 1975.
- [MML07] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522, 2007.
- [NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf’s Type Theory: An Introduction*. Oxford University Press, 1990.
- [Obe62] A. Oberschelp. Untersuchungen zur mehrsortigen Quantorenlogik. *Mathematische Annalen*, 145:297–333, 1962.
- [Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
- [Pit00] A. Pitts. Categorical Logic. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 5. Algebraic and Logical Structures*, chapter 2, pages 39–128. Oxford University Press, 2000.
- [Rab06] F. Rabe. First-Order Logic with Dependent Types. In N. Shankar and U. Furbach, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2006.
- [Rab08] F. Rabe. *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen, 2008.
- [RV02] A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Communications*, 15:91–110, 2002.
- [TB85] A. Trybulec and H. Blair. Computer assisted reasoning with Mizar. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, Los Angeles, CA, 1985.
- [Urb03] J. Urban. Translating Mizar for first-order theorem provers. In *MKM*, pages 203–215, 2003.

- [WAB<sup>+</sup>99] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topić. System description: SPASS version 1.0.0. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 314–318, Trento, Italy, 1999. Springer.

# Solving the \$100 Modal Logic Challenge

Florian Rabe<sup>a,1</sup> Petr Pudlák<sup>b,2</sup> Geoff Sutcliffe<sup>c</sup> Wein Shen<sup>c</sup>

<sup>a</sup>*Department of Computer Science, Carnegie Mellon University, USA*

<sup>b</sup>*Department of Theoretical Computer Science and Mathematical Logic  
Charles University, Czech Republic*

<sup>c</sup>*Department of Computer Science, University of Miami, USA*

---

## Abstract

We present the theoretical foundation, design, and implementation, of a system that automatically determines the subset relation between two given axiomatizations of propositional modal logics. This is an open problem for automated theorem proving. Our system solves all but six out of 121 instances formed from 11 common axiomatizations of seven modal logics. Thus, although the problem is undecidable in general, our approach is empirically successful in practically relevant situations.

*Key words:* Modal Logic, \$100 challenge, subset relationship

---

## 1 Introduction and Related Work

Modal logics are extensions of classical logic that handle the concept of modalities. Modern modal logic was founded by Clarence Irving Lewis in his 1910 Harvard thesis, and further developed in a series of scholarly articles beginning in 1912. In his book *Symbolic Logic* (with C. H. Langford), he introduced the five well-known modal logics **S1** through **S5** (Lewis and Langford, 1932). The contemporary era in modal logic began in 1959 when Saul Kripke introduced semantics for modal logics (Kripke, 1963). The mathematical structures of modal logics are modal algebras – Boolean algebras augmented with unary operations. Their study began to emerge with McKinsey’s proof that **S2** and **S4** are decidable (McKinsey, 1941). Today plain propositional modal logic

---

<sup>1</sup> The author was supported by a fellowship for Ph.D. research of the German Academic Exchange Service.

<sup>2</sup> The author was supported by the Grant Agency of Charles University (grant no. 377/2005/A INF).

is standard knowledge, see, e.g., Hughes and Cresswell (1996), and first order modal logic has been thoroughly studied, e.g., Fitting and Mendelsohn (1998). Henceforth in this paper attention is limited to propositional modal logic with the standard modalities *possibility* and *necessity*.

Many modal logics have multiple axiomatizations that are *equivalent*, in the sense that they generate the same *theory* - the same set of theorems. Similarly, one modal logic may be *stronger* than another in the sense that the stronger logic's theory is a strict superset of that of the weaker logic. Finally, two modal logic may be *incomparable* with one another, because each has theorems that the other does not. Such relationships between different axiomatizations of individual modal logics and between different modal logics, are well known (Hughes and Cresswell, 1996).

The Modal Logic \$100 Challenge (Sutcliffe, 2006) calls for a program that can determine the relationships between common Hilbert-style axiomatizations of the modal logics K, T, S1, S1°, S3, S4 and S5 (see Halleck (2006) for an overview and a list of references to various modal logics and axiomatizations). The program cannot simply encode known relationships. Rather, it must use logical reasoning from input axiomatizations to establish the relationships as if they were unknown. The syntactic representation of the axiomatizations can be anything reasonable, but the use of the TPTP syntax is encouraged. The challenge was sponsored by John Halleck, who has a practical need for such a program as an aid to maintaining his overview (Halleck, 2006).

Determining the relationship between two axiomatizations is undecidable in general (Blackburn et al., 2001). Decidability can be established separately for some modal logics. Historically, finding a complete Kripke semantics (Kripke, 1963) and establishing the finite model property by filtration (Lemmon, 1966) were used to obtain decidability of theoremhood. However, checking the subset relation between modal logics also involves checking the admissibility of rules. Here, decidability has been shown for a few cases, including K4 and S4 (Rybakov, 1997). Recently a framework has been developed in which admissibility is reduced to terminating analytic proofs for a variety of modal logics (Iemhoff and Metcalfe, 2007). In (Kracht, 1990) splittings are used to decide the admissibility of a rule for some logics, which correspond to transitive Kripke frames, but no algorithm is known for obtaining a splitting for an arbitrary logic. None of these methods is applicable to the full range of differently axiomatized modal logics. So far sophisticated implementations have focused on deriving theoremhood. (Goré et al., 1997) describe the *Logics Work Bench* program that is capable of reasoning about the modal systems K, KT, KT4, KT45 and KW. (Giunchiglia et al., 2002) use a SAT solver to decide a few classical systems. (Schmidt and Hustadt, 2006) give an overview over various methods based on translations of modal logic into first-order logic (e.g. Ohlbach and Schmidt, 1997). (Hustadt and Schmidt, 2000) extends the first-

order theorem prover SPASS with the ability to apply such translations to its input.

This paper describes an implemented and tested system within which relationships between modal logics can be determined. The system has been applied successfully to the \\$100 challenge. A partial preliminary version of this work has been presented as (Rabe, 2006b). The core idea is to use a simple translation to first-order logic (described in Section 3 of (Schmidt and Hustadt, 2006)), which encodes modal logic formulae as first-order terms, modal logic axioms and theorems as first-order atoms, and modal logic rules as first-order implications. This translation comes at the price of efficiency (McCune and Wos, 1992). We use it because it is applicable to any modal logic, in particular non-normal logics. Since we do not presuppose any semantics, the applicability of any other translation would itself have to be established automatically (which we do in the Kripke-based strategy described in Section 3.4). With this encoding, reasoning is performed using several modularly implemented (possibly incomplete) strategies, using first-order automated reasoning tools to prove or disprove the subset relationship: direct strategies, strategies based on Kripke semantics, and algebraic strategies that represent modal logics as modal algebras.

Section 2 provides the necessary background in modal logic and the encoding in first-order logic. Section 3 describes the implementation of our system, and the theoretical basis and implementation of the strategies. Section 4 documents and analyzes the results achieved by the system in attacking the \\$100 challenge. Section 5 concludes and provides directions for future research.

## 2 Modal Logics

### 2.1 Formulas and Rules

Modal formulae  $F, G, \dots$  are defined as the elements of the languages generated from the atomic formulae and connectives given in Figure 1 (note that prefix notation is used for the binary connectives). Rules are of the form

$$\frac{H_1 \quad \dots \quad H_n}{C}$$

where the  $H_i$  and  $C$  are modal formulae. The semantics of a rule is that if for any substitution  $\sigma$  of formulae for the propositional variables, all  $\sigma(H_i)$  are derivable, then so is  $\sigma(C)$ . The case  $n = 0$  means that  $C$  is an axiom. The rules with  $n \neq 0$  relevant for the \\$100 challenge are given in Figure 2.

---

|                      |   |   |
|----------------------|---|---|
| $p, q, \dots$        | a countably infinite set of propositional variables |   |
| $\neg F$             | negation  | (primitive)                                     |
| $\wedge FG$          | conjunction   | (primitive)                                     |
| $\diamond F$         | possibility   | (primitive)                                     |
| $\vee FG$            | disjunction   | $=_{defn} \neg \wedge \neg F \neg G$            |
| $\rightarrow FG$     | implication   | $=_{defn} \neg \wedge F \neg G$                 |
| $\leftrightarrow FG$ | equivalence   | $=_{defn} \wedge \rightarrow FG \rightarrow GF$ |
| $\Rightarrow FG$     | strict implication                                  | $=_{defn} \square \rightarrow FG$               |
| $\Leftrightarrow FG$ | strict equivalence                                  | $=_{defn} \wedge \Rightarrow FG \Rightarrow GF$ |
| $\Box F$             | necessity   | $=_{defn} \neg \diamond \neg F$                 |

---

Fig. 1. Atomic formulae and connectives

An *axiomatization* of a modal logic is a set of axioms and rules, from which the theory is generated by finitely many (including no) rule applications. A modal logic is characterized by its theory. For an axiomatization  $\mathcal{L}$  and a formula  $F$ ,  $\mathcal{L} \vdash F$  denotes that  $F$  is a theorem of  $\mathcal{L}$ . A rule  $R$  is an admissible rule of  $\mathcal{L}$  if  $\mathcal{L}$  and  $\mathcal{L} \cup \{R\}$  are equivalent; this is denoted by  $\mathcal{L} \vdash R$ .

## 2.2 The Modal Logics of the \$100 Challenge

The relationships between the modal logics to be compared in the \$100 challenge are shown in Figure 3. A solid arrow shows that an axiomatization of the logic at the head can be constructed by adding axioms or rules to an axiomatization of the logic at the tail. A dashed arrow shows that the logic at the head is stronger than the logic at the tail, but the axiomatizations have different heritages, e.g., the axiomatization of  $T$  is built by adding to  $K$  rather than by adding to  $S1$ . Regardless of the type of arrow, any path from one logic to another shows that the logic at the head of the path is stronger than the logic at the tail.

The axiomatizations used for the logics are given in Figure 4. There are two starting points for their construction - the propositional calculus  $PC$  and the strict system  $S1^\circ$ . Since different axiomatizations can generate the same theory, some axiomatizations are equivalent, e.g., all four axiomatizations of  $S5$  are equivalent. Different axiomatizations of the same logic are differentiated by Greek subscripts.

$PC$  is defined by the Hilbert and Bernays (Hilbert and Bernays, 1934), Lukas-

|            |                                    |  |
|------------|------------------------------------|--|
| <i>US</i>  | uniform substitution               | $\frac{F}{\sigma(F)}$  |
| <i>MP</i>  | modus ponens                       | $\frac{F}{\frac{G}{FG}}$   |
| <i>SMP</i> | strict modus ponens                | $\frac{F}{\frac{\Rightarrow FG}{G}}$                             |
| <i>AD</i>  | adjunction                         | $\frac{F}{\frac{G}{\wedge FG}}$                                  |
| <i>EQ</i>  | substitution of equivalents        | $\frac{\leftrightarrow FF'}{\frac{G}{G[F \rightsquigarrow F']}}$ |
| <i>EQS</i> | substitution of strict equivalents | $\frac{\leftrightarrow FF'}{\frac{G}{G[F \rightsquigarrow F']}}$ |

$\sigma$ : a substitution of propositional with formulae  
 $G[F \rightsquigarrow F']$ : formed from  $G$  by replacing some occurrences of  $F$  with  $F'$

Fig. 2. Rules

siewicz (Łukasiewicz, 1963), Rosser (Rosser, 1953), and Principia (Russell and Whitehead, 1910) axiomatizations as follows.

**Definition 1** (PC). The axiomatizations of PC are defined by the rules *US*, *MP*, and the following axioms:

For PCH (Hilbert-style):

- MT* :  $\rightarrow \rightarrow \neg p \neg q \rightarrow qp$
- A1* :  $\rightarrow \wedge pqp$
- A2* :  $\rightarrow \wedge pqq$
- A3* :  $\rightarrow p \rightarrow q \wedge pq$
- I1* :  $\rightarrow p \rightarrow qp$
- I2* :  $\rightarrow \rightarrow p \rightarrow pq \rightarrow pq$
- I3* :  $\rightarrow \rightarrow pq \rightarrow \rightarrow qr \rightarrow pr$
- O1* :  $\rightarrow p \vee pq$
- O2* :  $\rightarrow q \vee pq$
- O3* :  $\rightarrow pr \rightarrow \rightarrow qr \rightarrow \vee pqr$
- E1* :  $\rightarrow \leftrightarrow pq \rightarrow pq$
- E2* :  $\rightarrow \leftrightarrow pq \rightarrow qp$
- E3* :  $\rightarrow \rightarrow pq \rightarrow \rightarrow qp \leftrightarrow pq$

For PCL (Łukasiewicz-style):

- CN1* :  $\rightarrow \rightarrow pq \rightarrow \rightarrow qr \rightarrow pr$
- CN2* :  $\rightarrow p \rightarrow \neg pq$
- CN3* :  $\rightarrow \rightarrow \neg ppp$

For PCR (Rosser-style):

- KN1* :  $\rightarrow p \wedge pp$
- KN2* :  $\rightarrow \wedge pqp$
- KN3* :  $\rightarrow \rightarrow pq \rightarrow \neg \wedge qr \neg \wedge rp$

For PCP (Principia-style):<sup>3</sup>

- R1* :  $\rightarrow \vee ppp$
- R2* :  $\rightarrow q \vee pq$
- R3* :  $\rightarrow \vee pq \vee qp$
- R4* :  $\rightarrow \vee p \vee qr \vee q \vee pr$
- R5* :  $\rightarrow \rightarrow qr \rightarrow \vee pq \vee pr$

**S1°** is axiomatized in the Lewis-style, as taken from Zeman (1973). The Lemmon-style axiomatization of **S1°**, which is an extension of PC, was not used because it requires the weakened necessitation rule “if  $A$  is a PC theorem then  $\Box A$  is a theorem”, which is unreasonable to encode using the single-sorted

<sup>3</sup> Note that the axioms include the redundant *R4*, which can be proved from the others (Bernays, 1926).

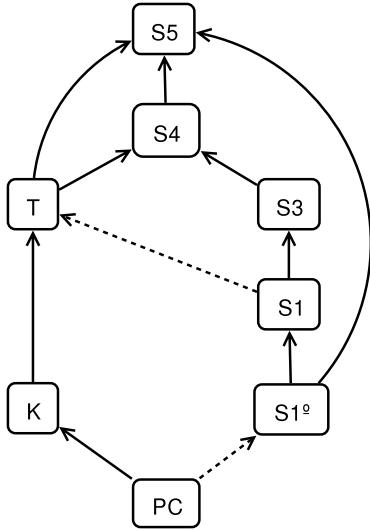


Fig. 3. The Hierarchy of the Modal Logics

first-order approach taken in this work.

**Definition 2** ( $S1^\circ$  – Lewis-style). The axiomatization  $S1^\circ$  is defined by the rules  $US$ ,  $SMP$ ,  $AD$ , and  $EQS$ , and the axioms

$$\begin{aligned}
 M1 &: \Rightarrow \wedge pq \wedge qp \\
 M2 &: \Rightarrow \wedge pqp \\
 M3 &: \Rightarrow \wedge \wedge pqr \wedge p \wedge qr \\
 M4 &: \Rightarrow p \wedge pp \\
 M5 &: \Rightarrow \wedge \Rightarrow pq \Rightarrow qr \Rightarrow pr
 \end{aligned}$$

Note that all the axiomatizations include the structural rule  $US$ , which is crucial for the soundness of the first-order encoding described in Section 2.3.

### 2.3 First-order Encoding

Modal formulae are encoded as first-order formulae with equality, the first-order connectives are written as  $\neg$ ,  $\wedge$ , and  $\rightarrow$ , the universal quantifier as  $\forall X, Y, \dots$ , and equality as  $=$ .  $T \vdash^{FOL} F$  denotes that  $F$  is a first-order theorem of the theory  $T$ .

The first-order signature used for encoding modal formulae consists of the following symbols:

- unary function symbols: `not`, `poss`, `necess`,
- binary function symbols: `and`, `or`, `impl`, `equiv`, `s_impl`, `s_equiv`,
- unary predicate symbol: `thm`.

|             |     |             |     |         |  |
|-------------|-----|-------------|-----|---------|--|
| $K$         | $=$ | $PC$        | $+$ | $Nec :$ | $\frac{F}{\Box F}$   |
|             |     |             |     | $+ K :$ | $\rightarrow \Box \rightarrow pq \rightarrow \Box p \Box q$              |
| $T$         | $=$ | $K$         | $+$ | $M :$   | $\rightarrow \Box pp$  |
| $S1$        | $=$ | $S1^\circ$  | $+$ | $M6 :$  | $\Rightarrow p \Diamond p$   |
| $S3$        | $=$ | $S1$        | $+$ | $S3 :$  | $\Rightarrow \Rightarrow pq \Rightarrow \neg \Diamond q \neg \Diamond p$ |
| $S4_\alpha$ | $=$ | $S3$        | $+$ | $M9 :$  | $\Rightarrow \Diamond \Diamond p \Diamond p$                             |
| $S4_\beta$  | $=$ | $T$         | $+$ | $4 :$   | $\rightarrow \Box p \Box \Box p$   |
| $S5_\alpha$ | $=$ | $S4_\alpha$ | $+$ | $B :$   | $\rightarrow p \Box \Diamond p$  |
| $S5_\beta$  | $=$ | $S4_\beta$  | $+$ | $B :$   | $\rightarrow p \Box \Diamond p$  |
| $S5_\gamma$ | $=$ | $S1^\circ$  | $+$ | $M10 :$ | $\Rightarrow \Diamond p \Box \Diamond p$                                 |
| $S5_\delta$ | $=$ | $T$         | $+$ | $5 :$   | $\rightarrow \Diamond p \Box \Diamond p$                                 |

Fig. 4. Axiomatizations to be Compared

Then the encoding  $\mathcal{E}(\cdot)$  is defined as follows:

- (1) for an axiomatization  $\mathcal{L} = \{R_1, \dots, R_n\}$ :

$$\mathcal{E}(\mathcal{L}) = Def \cup \{\mathcal{E}(R_1), \dots, \mathcal{E}(R_n)\}$$

is a first-order theory over the above signature where  $Def$  consists of the following axioms:

- $\forall X, Y \text{ or}(X, Y) = \text{not}(\text{and}(\text{not}(X), \text{not}(Y))),$
- $\forall X, Y \text{ impl}(X, Y) = \text{not}(\text{and}(X, \text{not}(Y))),$
- $\forall X, Y \text{ equiv}(X, Y) = \text{and}(\text{impl}(X, Y), \text{impl}(Y, X)),$
- $\forall X, Y \text{ s\_impl}(X, Y) = \text{necess}(\text{impl}(X, Y)),$
- $\forall X, Y \text{ s\_equiv}(X, Y) = \text{and}(\text{s\_impl}(X, Y), \text{s\_impl}(Y, X)),$
- $\forall X \text{ necess}(X) = \text{not}(\text{poss}(\text{not}(X))).$

- (2) for a rule  $R = \frac{H_1 \quad \dots \quad H_n}{C}$  with propositional variables  $p_1, \dots, p_m$ :

$$\mathcal{E}(R) = \forall X_1, \dots, X_m ((\mathcal{E}(H_1) \wedge \dots \wedge \mathcal{E}(H_n)) \rightarrow \mathcal{E}(C)),$$

- (3) for a modal formula  $F$ :  $\mathcal{E}(F) = \text{thm}(\varepsilon(F))$ , where  $\varepsilon(F)$  encodes every formula  $F$  as a first-order term by
- $\varepsilon(\wedge FG) = \text{and}(\varepsilon(F), \varepsilon(G))$  and similarly for  $\vee, \rightarrow, \leftrightarrow, \Rightarrow,$  and  $\Leftarrow,$
  - $\varepsilon(\Box F) = \text{necess}(\varepsilon(F))$  and similarly for  $\neg$  and  $\Diamond,$
  - $\varepsilon(p_i) = X_i$  for a propositional variable  $p_i.$

For example, for the rule  $MP$ , we have

$$\mathcal{E}(MP) = \forall X_1, X_2 (\text{thm}(X_1) \wedge \text{thm}(\text{impl}(X_1, X_2))) \rightarrow \text{thm}(X_2)$$

Note that the rules  $US$ ,  $EQ$  and  $EQS$  cannot be encoded in this way.  $US$  is inherent in the encoding as Theorem 3 shows. Section 3.2.2 shows how  $EQ$  and  $EQS$  are replaced by congruence rules, and Section 3.2.3 shows how the congruence rules can be replaced by formulae allowing use of efficient first-order equality reasoning.

The following soundness result guarantees that reasoning about the first-order encoding is equivalent to reasoning about the encoded axiomatization.

**Theorem 3.** *Let  $\mathcal{L}$  be an axiomatization. Then for modal rules  $R$*

$$\mathcal{E}(\mathcal{L}) \vdash^{FOL} \mathcal{E}(R) \text{ if and only if } \mathcal{L} \vdash R.$$

*Proof.* Since  $\mathcal{E}(\mathcal{L})$  contains only Horn formulae, there is a free first-order model  $M$  of  $\mathcal{E}(\mathcal{L})$  such that  $M$  is term-generated and  $\mathcal{E}(R)$  holds in  $M$  iff  $\mathcal{E}(\mathcal{L}) \vdash^{FOL} \mathcal{E}(R)$ . The universe of  $M$  can be constructed by taking the set of equivalence classes generated by equality axiomatized by reflexivity, symmetry, transitivity, congruence and the equality axioms. Let  $[t]$  denote the equivalence class of  $t$ . Clearly, two terms are equal in  $M$  iff the modal formulae they represent can be transformed into each other by eliminating and introducing abbreviations of modal formulae.

Function symbols are interpreted in  $M$  as induced by the equivalence relation. And  $\text{thm}^M$  is the smallest fixed point of the following operation:  $[t] \in \text{thm}^M$  iff there is a rule in  $\mathcal{L}$  encoded as

$$\forall X_1, \dots, X_m ((\text{thm}(h_1) \wedge \dots \wedge \text{thm}(h_n)) \rightarrow \text{thm}(c))$$

and a substitution  $\alpha$  for the variables  $X_1, \dots, X_m$  such that  $[\alpha(h_i)] \in \text{thm}^M$  for  $i = 1, \dots, m$  and  $\alpha(c) = t$ .

Then because  $US$  is admissible in  $\mathcal{L}$ , we have for every modal formula  $F$  and every substitution  $\alpha$ :

$$[\alpha(\varepsilon(F))] \in \text{thm}^M \text{ if and only if } \mathcal{L} \vdash F^\alpha$$

where  $F^\alpha$  denotes the uniform substitution instance of  $F$  under  $\alpha$ . Therefore, the definition of  $\mathcal{L} \vdash R$  is equivalent to saying that  $\mathcal{E}(R)$  holds in  $M$ , which completes the proof.  $\square$

### 3 Solution

In this section our solution to the \$100 challenge is presented. This section is organized as follows. In Section 3.1, we give an overview over our system, and in the remaining sections, we present its theoretical basis and the implementation. In particular, Section 3.2 describes the preprocessing phase, and Sections 3.3 to 3.5 describe the comparison strategies used.

#### 3.1 System Architecture and Process

Our system is implemented in Standard ML of New Jersey (SML, 2007). The source code can be obtained from (Rabe, 2006a). After loading the sources into the SML top-level, the user can call a function `compare : string * string -> unit`. This function takes the filenames of the logics to be compared as arguments, and prints the results of the comparison.

The input files must contain two axiomatizations,  $\mathcal{L}$  and  $\mathcal{M}$ , in the TPTP format (Sutcliffe and Suttner, 1998). In addition to the encoded axioms and rules, the input files can contain *special rules* of the form:

```
fof(name, special_rule, ignored).
```

where *name* identifies the special rule and the rest is ignored. Special rules are used to import PC axiomatizations into PC based modal logics, and to represent aspects that require special processing, e.g., rules for substitution of equivalents. After reading in the files, two phases can be distinguished.

The *preprocessing* phase, described in Section 3.2, includes the expansion of special rules into sets of normal rules, and optimizations related to congruence relations. We also try to establish certain properties of the logics, like normality, so that these properties can be reused later. The preprocessing returns two different but equivalent axiomatizations for every logic. For  $\mathcal{L}$  and  $\mathcal{M}$ , we obtain  $\mathcal{L}^b$  &  $\mathcal{L}^s$  and  $\mathcal{M}^b$  &  $\mathcal{M}^s$ . The *b* axiomatizations are “big”, containing redundant axioms, useful lemmas, etc., and are used when proving *from* the logic. The *s* axiomatizations are “small” and used when proving *to* the logic.

The *comparison* phase, described in Sections 3.3 to 3.5, attempts to determine the relationship between the two input axiomatizations. First the system checks whether  $\mathcal{L}^b$  is stronger than  $\mathcal{M}^s$ , and then it checks whether  $\mathcal{M}^b$  is stronger than  $\mathcal{L}^s$ . In both directions the following happens: The system tries to prove every axiom and rule of the *s* logic from the *b* logic. Several proving strategies are available for proving each axiom and rule. The strategies are tried in turn until one succeeds or all have failed. Axioms and rules that fail

to be proved are passed to disproving strategies. The disproving strategies try to find a counterexample for each axiom and rule, establishing that the axiom or rule cannot be proved.

Three kinds of strategies are used in the comparison phase: direct strategies are described in Section 3.3, strategies based on Kripke semantics in Section 3.4, and strategies based on algebraic encodings in Section 3.5. All strategies are parametric in the specific first-order prover or model finder that is used.

If both directions succeed, whether by proving or by disproving, the relationship between the logics is decided, and  $\mathcal{L} \subset \mathcal{M}$ ,  $\mathcal{M} \subset \mathcal{L}$ ,  $\mathcal{L} = \mathcal{M}$  or  $\mathcal{L}$  incomparable to  $\mathcal{M}$  is printed. If only one direction succeeds, a partial result is printed.

### 3.2 Preprocessing

#### 3.2.1 Special Rule pc

The special rule `pc` is expanded into an axiomatization of `PC`. The four axiomatizations of `PC` defined in Section 2.2 are equivalent (see also McCune et al. (2002)). This can be demonstrated automatically by proving the axioms of each from the axiomatizations of each other (as all axiomatizations use the same rules, the rules do not need to be proved), which was done using the ATP system VAMPIRE 8.1 with a 180s CPU time limit, on a 2.8GHz PC with 1GB memory and running Linux 2.6. The results are summarized in Fig. 5, which gives the CPU times in seconds for the proofs of the axioms from the named axiomatizations, or TO for proof attempts that timed out at 180s. The results show that the Hilbert axiomatization can prove the Łukasiewicz and Principia axioms, the Łukasiewicz axiomatization can prove the Rosser axioms, and the Principia axiomatization can prove the Łukasiewicz axioms. While the results are not all positive, the results are useful: (i) if the Hilbert axiomatization can be proved, that is sufficient for claiming that all four axiomatizations have been proved, and (ii) if the Hilbert axiomatization is used as a basis for constructing modal logics, then it is possible to add the other three axiomatizations' axioms as lemmas.

Due to these results, the `pc` special rule is expanded into the Hilbert axioms when computing an *s* axiomatization, and into the union of all four `PC` axiomatizations when computing a *b* axiomatization. For simplicity, the proofs justifying this treatment are not executed explicitly every time.

| Prove→ | PCH |     |     |     |     |     |     |    |    |    |    |    |    |
|--------|-----|-----|-----|-----|-----|-----|-----|----|----|----|----|----|----|
| From↓  | MT  | A1  | A2  | A3  | O1  | O2  | O3  | I1 | I2 | I3 | E1 | E2 | E3 |
| PCL    | 124 | 3   | 68  | 0   | TO  | TO  | TO  | 0  | 3  | TO | 69 | 72 | TO |
| PCR    | 110 | 5   | 11  | TO  | 0   | 5   | 132 | 2  | 5  | TO | 0  | 4  | TO |
| PCP    | 0   | 0   | 55  | 16  | 2   | 0   | 4   | 2  | 0  | TO | 2  | 0  | TO |
| Prove→ | PCL |     |     | PCR |     |     | PCP |    |    |    |    |    |    |
| From↓  | CN1 | CN2 | CN3 | KN1 | KN2 | KN3 | R1  | R2 | R3 | R4 | R5 |    |    |
| PCH    | 0   | 0   | 1   | 0   | 0   | TO  | 1   | 0  | 2  | 4  | 4  |    |    |
| PCL    | -   | -   | -   | 58  | 58  | 59  | 112 | TO | TO | TO | TO |    |    |
| PCR    | TO  | 4   | 3   | -   | -   | -   | 1   | 5  | 1  | TO | TO |    |    |
| PCP    | 16  | 1   | 4   | 0   | 2   | TO  | -   | -  | -  | -  | -  |    |    |

Fig. 5. Relationships between PC Axiomatizations

### 3.2.2 Special Rules $\text{eq}$ and $\text{eqs}$

The special rules  $\text{eq}$  represents the  $EQ$  rules, which cannot be expressed directly using the first-order encoding. The first step around this is to use the following rules that define  $\leftrightarrow$  to be a congruence relation:

$$\begin{array}{c}
 \frac{\leftrightarrow FG}{\leftrightarrow \neg F \neg G} (EQ1) \quad \frac{\leftrightarrow FF' \quad \leftrightarrow GG'}{\leftrightarrow \wedge FG \wedge F'G'} (EQ2) \quad \frac{\leftrightarrow FG}{\leftrightarrow \diamond F \diamond G} (EQ3) \\
 \frac{\leftrightarrow FG \quad F}{G} (EQ4) \quad \frac{}{\leftrightarrow FF} (EQ5)
 \end{array}$$

The following lemma then relates  $EQ$  to  $\leftrightarrow$  being a congruence relation in the context of the modal logic under consideration.

**Lemma 4.** *If  $\mathcal{L} \vdash EQ5$ , then  $\mathcal{L} \cup \{EQ\}$  and  $\mathcal{L} \cup \{EQ1, EQ2, EQ3, EQ4\}$  are equivalent.*

*Proof.* If  $\mathcal{L} \cup \{EQ1, EQ2, EQ3, EQ4\}$  is given, we need to derive  $EQ$ . Let  $F$ ,  $F'$  and  $G$  be as in the definition of  $EQ$ , where we can assume without loss of generality that no defined connective occurs in them. We need to derive  $G[F \rightsquigarrow F']$ . We construct a backwards proof, firstly applying  $EQ4$ , to reduce to  $\leftrightarrow G(G[F \rightsquigarrow F'])$ . This can be derived by repeated application of  $EQ1-EQ3$  along the structure of  $G$  until all open proof goals are  $\leftrightarrow FF'$  or are instances of  $EQ5$  under  $US$ . Conversely, let  $\mathcal{L} \cup \{EQ\}$  be given. We need to derive the rules  $EQ1-EQ4$ .  $EQ1-EQ3$  are special cases of  $EQ$  with, e.g.,  $G = \leftrightarrow \neg p \neg p$ , and  $EQ4$  is the special case of  $EQ$  where  $F = G$ .  $\square$

Given Lemma 4, when the special rule `eq` is found, an attempt is made to prove  $EQ5$ . If this succeeds the special rule is expanded to  $EQ1-EQ4$ , and the proved  $EQ5$  is added to the axiomatization. The analogue of Lemma 4 for strict equivalence can be proved, and the rule  $EQS$  is handled correspondingly by a special rule `eqs`.

### 3.2.3 Congruences

None of the axiomatizations of the challenge is defined to include the rule  $EQ$ . However, this rule is extremely powerful, and is necessary for success when proving relationships between modal logics. Section 3.2.2 explains that  $EQ$  is represented in input files as a special rule, and is expanded to  $EQ1-EQ4$  if  $EQ5$  can be proved. The congruence rules are inefficient in implementing substitution. A much more efficient approach is to exploit the equational reasoning of a first-order theorem prover. If the relation  $\mathcal{L} \leftrightarrow FG$  is a congruence relation on the set of modal formulas, the rule

$$\forall X, Y (\text{thm}(\text{equiv}(X, Y)) \rightarrow X = Y) \quad (*)$$

is added to  $\mathcal{L}^b$ . The soundness of this addition is given by the following lemma:

**Lemma 5.** *If  $\mathcal{L} \vdash EQ_i$  for all  $i = 1, \dots, 5$ , then adding  $(*)$  to the first-order encoding of  $\mathcal{L}$  does not destroy the soundness of the encoding.*

*Proof.* Let  $M$  be the free model constructed in the proof of Theorem 3. Because  $\mathcal{L}$  has the rules mentioned above and due to Lemma 4, if  $\leftrightarrow FG$  is derivable in  $\mathcal{L}$ , either both  $F$  and  $G$  are derivable in  $\mathcal{L}$  or none. Then, by induction on the construction of  $M$ , it follows that adding the above rule will never identify two terms in the term model of which only one corresponds to a derivable modal formula. Therefore, the terms that are in the equivalence classes in the interpretation of `thm` stay the same, and soundness is preserved.  $\square$

For a PC based axiomatization  $\mathcal{L}$ , proving  $\mathcal{L} \vdash EQ_i$  for all  $i = 1, \dots, 5$  can be done in parts. The proofs of  $\text{PC} \vdash \{EQ1, EQ2, EQ4, EQ5\}$ , which do not mention the modal operators, can be done offline in advance. This is described below. Then given a PC based axiomatization  $\mathcal{L}$  it is necessary to prove only  $\mathcal{L} \vdash EQ3$ .

The proofs of  $\text{PC} \vdash \{EQ1, EQ2, EQ4, EQ5\}$  were done using the combined axiomatization  $\text{PC} = \text{PCH} \cup \text{PCL} \cup \text{PCR} \cup \text{PCP}$  (whose combination is justified above), and the same hardware and software environment as above.  $EQ1$  was proved in 50s,  $EQ4$  in 0s, and  $EQ5$  in 1s. However,  $EQ2$  could not be proved, and two lemmas were used as stepping stones:

$$\frac{\leftrightarrow pp'}{\leftrightarrow \wedge pq \wedge p'q} (EQ2a), \quad \frac{\leftrightarrow pp'}{\leftrightarrow \wedge qp \wedge qp'} (EQ2b)$$

$EQ2a$  was proved in 79s and  $EQ2b$  in 95s. Attempts to prove  $EQ2$  from the combined axiomatization augmented with the two lemmas were not successful. However using only PCH augmented with the two lemmas produced a proof of  $EQ2$  in 6s (the redundancy in the combined axiomatization clearly affected VAMPIRE’s search in this case).

The rule  $EQS$  is used in  $S1^\circ$  (based) axiomatizations. The analogue of Lemma 5 for strict equivalence can be proved, and the rule  $EQS$  is handled correspondingly. The proofs of the analogues of  $\mathcal{L} \vdash EQ_i$  are all trivial, because  $S1^\circ$  based axiomatizations include the `eqs` special rule, which would have been expanded to those rules beforehand.

### 3.2.4 Testing the Applicability of Advanced Strategies

We use advanced strategies the applicability of which has to be proved itself. Those parts of these proofs that depend only on the modal logic we are proving from (and not on the axiom or rule to be proved or disproved) are executed in the preprocessing phase, and the results of the computations are stored along with  $\mathcal{L}^b$  and  $\mathcal{L}^s$  to represent  $\mathcal{L}$ . The details of these preprocessing steps are given in Sections 3.4 and 3.5 when describing these advanced strategies, namely the strategies `kripke_pos` and `s10_pos`.

The strategy `kripke_pos` uses a relational translation into first-order logic, which depends on the normality of the logic  $\mathcal{L}$ . Therefore, we try to prove that  $\mathcal{L}$  is normal, i.e., closed under the rules of K. If so, those rules are added to  $\mathcal{L}^b$ . This translation can be further improved by finding a property of Kripke frames that characterizes  $\mathcal{L}$ . Therefore, we identify the Sahlqvist axioms of  $\mathcal{L}$  and find their corresponding frame properties.

The strategy `s10_pos`, which uses an algebraic encoding of  $S1^\circ$ , requires an axiomatization of  $\mathcal{L}$  that consists of the axioms and rules of  $S1^\circ$  and additional axioms. Therefore, we try to find such an axiomatization. We also try to bring the additional axioms into a certain form to enhance the algebraic encoding.

## 3.3 Direct Strategies

In this section, the direct strategies are presented. The two proving strategies are purely syntactic, and the disproving strategy uses a first-order model

finder. All the direct strategies are always applicable and do not require additional knowledge about the logics.

### 3.3.1 Proving

Let  $\mathcal{L}$  be an axiomatization produced by the preprocessing and let  $\mathcal{M}'$  be as  $\mathcal{M}$  but with an additional rule  $R$ . Obviously, we have:

**Lemma 6.** *If  $R$  is an axiom,*

$$\mathcal{M}' \subseteq \mathcal{L} \text{ if and only if } \mathcal{M} \subseteq \mathcal{L} \text{ and } \mathcal{L} \vdash R,$$

*and if  $R$  is not an axiom,*

$$\mathcal{M}' \subseteq \mathcal{L} \text{ if } \mathcal{M} \subseteq \mathcal{L} \text{ and } \mathcal{L} \vdash R.$$

Lemma 6 is used to implement the strategy `direct_pos`. It takes a logic  $\mathcal{L}$  and a rule  $R$  as input and calls a first-order theorem prover to prove  $\mathcal{L} \vdash R$ .

In Lemma 6, the “only if” direction does not hold for rules. This is because deriving  $R$  from  $\mathcal{L}$  requires showing that whenever  $\mathcal{L}$  contains instances of the hypotheses of  $R$ , it also contains the appropriate instance of the conclusion. For the “only if” direction to hold, we would need the weaker condition that whenever  $\mathcal{M}'$  (which is a subset of  $\mathcal{L}$ ) contains instances of the hypotheses of  $R$ , then  $\mathcal{L}$  contains the appropriate instance of the conclusion. For a trivial example, let  $\mathcal{M}$  be the empty axiomatization,  $R$  be the rule

$$\frac{p}{\neg p}$$

and  $\mathcal{L}$  be any consistent non-empty axiomatization. Clearly,  $R$  is not admissible in  $\mathcal{L}$  because  $\mathcal{L}$  is consistent. But  $\mathcal{M}'$  is still empty because an axiomatization without axioms has no theorems even if it contains an inconsistent rule, and therefore,  $\mathcal{M}$  is a subset of  $\mathcal{L}$ .

Furthermore, a theorem prover will often not even find a proof of  $\mathcal{L} \vdash R$ , in particular if  $R$  is a rule that is admissible in  $\mathcal{L}$  but not derivable. The simplest such case arises when  $R$  is the necessitation rule and  $\mathcal{L}$  is an  $S1^\circ$ -based axiomatization of  $S4$  or  $S5$ . The following lemma gives an inductive admissibility criterion.

**Lemma 7.** *Let  $R$  be of the form*

$$\frac{p}{F(p)}$$

for some formula  $F$  in one propositional variable  $p$ . We write  $F(G)$  for substituting  $p$  in  $F$  with  $G$ . Then  $\mathcal{M}' \subseteq \mathcal{L}$  if

- $\mathcal{M} \subseteq \mathcal{L}$  and
- for every rule of  $\mathcal{L}$  with hypotheses  $H_1, \dots, H_n$  and conclusion  $C$ , the rule

$$\frac{H_1 \quad \dots \quad H_n \quad F(H_1) \quad \dots \quad F(H_n)}{F(C)}$$

is derivable from  $\mathcal{L}$ .

*Proof.* We need to show that  $R$  is admissible in  $\mathcal{L}$ , i.e., whenever a formula  $G$  is derivable, then so is  $F(G)$ . This is proved by a straightforward induction over the theorems of  $\mathcal{L}$ . The base case means that  $\mathcal{L} \vdash F(A)$  for every axiom  $A$ . This holds due to the above condition (here  $n = 0$ ). The induction step is a rule application leading from  $H_1, \dots, H_n$  to  $C$ : Under the induction hypothesis that  $F(H_i)$  is a theorem for  $i = 1, \dots, n$ ,  $F(C)$  must be a theorem. This is exactly what the above condition states.  $\square$

The necessitation rule arises in the special case where  $F(p) = \Box p$ . Lemma 7 is used to implement the strategy `direct_ind_pos`, which takes  $\mathcal{L}$  and  $R$  as input and calls a first-order theorem prover to prove every induction step. Note that it would also be sufficient if the second condition quantified over the rules of  $\mathcal{M}'$  instead of those of  $\mathcal{L}$ . But since these rules include  $R$ , it is less successful in practice.

### 3.3.2 Disproving

The direct strategy to disprove the subset relation  $\mathcal{M} \subseteq \mathcal{L}$  is to show a certain satisfiability.

**Lemma 8.** *If  $R$  is an axiom or rule of  $\mathcal{M}$ , and if there is a first-order model  $M$  of  $\mathcal{E}(\mathcal{L}) \cup \{\neg \mathcal{E}(R)\}$ , then  $\mathcal{M} \not\subseteq \mathcal{L}$ .*

This approach is implemented in the strategy `direct_neg`, which calls a first-order model finder to search for a model of  $\mathcal{E}(\mathcal{L}) \cup \{\neg \mathcal{E}(R)\}$  if  $R$  could not be proved by any positive strategy. This criterion is not complete since we only check finite models; see Section 4 for a discussion.

## 3.4 Strategies using Kripke semantics

This subsection presents a proving and a disproving strategy using relational translations, which we call Kripke-based strategies.

### 3.4.1 Proving

By standard first-order translation, we mean the translation based on the relational semantics of modal logics by making worlds explicit, e.g.,  $\Box p$  is translated to  $\forall w \forall x (Acc(w, x) \rightarrow p(x))$  for an accessibility relation  $Acc$  (see Section 4.1 in Schmidt and Hustadt (2006)). Then we have:

**Lemma 9.** *Let*

- (1)  $\mathcal{L}$  be normal,
- (2)  $\mathcal{F}$  be a set of theorems of  $\mathcal{L}$  that are Sahlqvist formulas,
- (3)  $P$  be the first-order property of Kripke frames completely characterized by  $\mathcal{F}$ ,
- (4)  $R'$  be the standard first-order translation of the modal formula  $R$ ,
- (5)  $R'$  be first-order provable from  $P$ .

*Then  $\mathcal{L} \vdash R$ .*

This result follows from Sahlqvist's theorem (Sahlqvist, 1975). Lots of practically relevant axioms are Sahlqvist formulas, e.g., any formula of the form  $F \rightarrow G$  where  $G$  is a positive formula and  $F$  is constructed by applying conjunction, disjunction and possibility to boxed atoms and negative formulas. To compute  $P$  from  $\mathcal{F}$ , we use the SCAN algorithm (Gabbay and Ohlbach, 1992; Goranko et al., 2004) for second-order quantifier elimination, for which an implementation is available.<sup>4</sup>

In our implementation, the first three steps, i.e., proving normality of  $\mathcal{L}$  and computing  $P$ , are done in the preprocessing phase. The direct strategies are used for the normality proof. Then the strategy `kripke_pos` computes  $R'$  from  $R$ , where  $R$  is the rule or axiom that is to be proved from  $\mathcal{L}$ , and calls a first-order theorem prover to prove  $R'$  from  $P$ .

Note that we cannot use relational semantics in general, because Kripke semantics may not be sound (e.g., for  $S1$ ) or not be complete (see, e.g., Thomason (1974)) for a given modal logic. It is necessary to find a set of Kripke frames that corresponds to the modal logic and show that this set of frames is complete for it. Lemma 9 gives the most important class of modal logics for which this has been proved.

---

<sup>4</sup> Technically, a SCAN implementation is only available for SunOS. Our Linux system outputs SCAN command lines, and the user has to run them on a SunOS machine and submit the result to a database.

### 3.4.2 Disproving

We cannot easily use the proving approach as a disproving strategy because, in general, it only gives us a sublogic of  $\mathcal{L}$  that is characterized by the property of Kripke frames. But this is not necessary anyway because the following simpler and more general strategy is successful. We search for a Kripke model  $m' = (U, Acc', \alpha)$  such that the formulas satisfied by  $m'$  include the theorems of  $\mathcal{L}$  but not  $F$ , in order to prove  $\mathcal{M} \not\subseteq \mathcal{L}$  for  $\mathcal{M} \vdash F$ ; here  $U$  is the set of worlds,  $Acc'$  the accessibility relation, and  $\alpha$  an assignment of truth values to the propositional variables of  $F$ . This means that, firstly,  $m'$  must satisfy all rules of  $\mathcal{L}$ , i.e., an instance of the conclusion of a rule must hold in all worlds whenever the appropriate instances of all hypotheses of the rule hold in all worlds. Secondly,  $m'$  must satisfy  $\neg F$  in one world.

This is non-trivial to implement. If Kripke semantics is used to translate modal logic to first-order logic, the first-order language is not a meta-language anymore, i.e., modal formulas are translated to first-order formulas, not to terms. Therefore, the possibility of quantifying over all modal formulas is lost, which is necessary to express that a model satisfies a rule. To circumvent this problem, we fix the number of worlds in  $U$ , say  $n$ , and proceed as follows: We assume that all propositional variables are of the form  $p_j$  for some natural number  $j$ . We search for a first-order model  $m$ , from which we can construct the Kripke model  $m'$ . Let the first-order signature  $\Sigma$  contain the following symbols: constants  $1, \dots, n$  (intended semantics: one constant for every world of  $U$ ), the constants  $t$  and  $f$  (intended semantics: truth values of truth and falsity), the binary predicate  $Acc$  (intended semantics: the accessibility relation  $Acc'$ ), and one constant  $a_{ji}$  for every variable  $p_j$  occurring in  $F$  and for every  $i = 1, \dots, n$  (intended semantics:  $a_{ji}$  gives the value of the assignment  $\alpha$  to  $p_j$  in world  $i$ ). Now let  $\tilde{\mathcal{E}}(\cdot)$  be the translation from modal logic rules and formulas to first-order logic over  $\Sigma$  defined as follows.

- (1) A rule with hypotheses  $H_1, \dots, H_r$  and conclusion  $C$  containing the propositional variables  $p_1, \dots, p_s$  is translated to

$$\forall X_{11}, \dots, X_{1n}, X_{21}, \dots, X_{sn} : ((\tilde{\mathcal{E}}(H_1) \wedge \dots \wedge \tilde{\mathcal{E}}(H_r)) \rightarrow \tilde{\mathcal{E}}(C))$$

- (2)  $\tilde{\mathcal{E}}(\Rightarrow FG)$  and  $\tilde{\mathcal{E}}(\Leftrightarrow FG)$  are reduced to the other cases by replacing them with their definitions,
- (3) A formula  $F$  is translated to  $\tilde{\mathcal{E}}(F) = \bigwedge_{i=1}^n \tilde{\mathcal{E}}^i(F)$  where  $\tilde{\mathcal{E}}^i(F)$  is given by
  - $\tilde{\mathcal{E}}^i(\wedge FG) = \tilde{\mathcal{E}}^i(F) \wedge \tilde{\mathcal{E}}^i(G)$  and accordingly for the other binary propositional connectives,
  - $\tilde{\mathcal{E}}^i(\neg F) = \neg \tilde{\mathcal{E}}^i(F)$ ,
  - $\tilde{\mathcal{E}}^i(\Box F) = \bigwedge_{j=1}^n (Acc(i, j) \rightarrow \tilde{\mathcal{E}}^j(F))$ ,

- $\tilde{\mathcal{E}}^i(\Diamond(F)) = \bigvee_{j=1}^n (Acc(i, j) \wedge \tilde{\mathcal{E}}^j(F)),$
- $\tilde{\mathcal{E}}^i(p_j) = (X_{ji} = t)$  for a propositional variable  $p_j$  (where the first equality sign is a meta-operator and the second one the logical symbol).

Here, the intended semantics of  $\tilde{\mathcal{E}}(F)$  for a formula  $F$  is that  $F$  holds in all worlds of  $m'$  and that of  $\tilde{\mathcal{E}}^i(F)$  is that  $F$  holds in the world  $i$ . With these definitions, we have the following lemma.

**Lemma 10.** *If  $F$  is a theorem of  $\mathcal{M}$  and there is an  $n$  such that a first-order  $\Sigma$ -model  $m$  exists satisfying the following axioms*

- $\neg c = d$  for all constants  $c$  and  $d$  of  $\Sigma$ ,
- $a_{ji} = t \vee a_{ji} = f$  for all constants  $a_{ji}$  of  $\Sigma$ ,
- $\tilde{\mathcal{E}}(R)$  for every rule  $R$  of  $\mathcal{L}$ ,
- $\neg F'$  where  $F'$  is as  $\tilde{\mathcal{E}}^1(F)$  but with all variables  $X_{ji}$  replaced with  $a_{ji}$ ,

then  $\mathcal{M} \not\subseteq \mathcal{L}$ .

*Proof.* From  $m$ ,  $m'$  is constructed by

- $U$ : the universe of  $m$  minus the interpretations of  $t$  and  $f$ ,
- $Acc'$ : the restriction of the interpretation of  $Acc$  to  $U$ ,
- for a variable  $p_j$  of  $F$  and a world  $i$  of  $U$ ,  $\alpha(p_j)(i)$  is true if  $(a_{ji} = t)$  holds in the model, and false if  $(a_{ji} = f)$  holds.

Let  $T$  be the set of modal formulas that hold in all worlds of  $m'$ . Then, we observe that the above translation indeed has the intended semantics, i.e., if for a rule  $R$ ,  $\tilde{\mathcal{E}}(R)$  holds in  $m$ , then if  $T$  contains the hypotheses of  $R$ ,  $T$  also contains the conclusion of  $R$ . Therefore,  $T \subseteq \mathcal{L}$ . And also by the translation, since  $\neg F'$  holds in  $m$ ,  $F$  does not hold in world 1 of  $m'$ , and therefore  $F \notin \mathcal{L}$ . Because  $F$  is a theorem of  $\mathcal{M}$ , we have  $\mathcal{M} \not\subseteq \mathcal{L}$ .  $\square$

This criterion can be applied regardless of whether  $\mathcal{L}$  has a complete Kripke semantics,  $\mathcal{L}$  does not even have to be normal. Whereas for the proving case, the lack of a complete Kripke semantics threatens soundness, for a disproving strategy, it only threatens completeness, which is harmless.

Lemma 10 is used to implement the strategy `kripke_neg`, which executes the above translation and calls a first-order model finder to search for the model  $m$ . Experiments showed that very low values of  $n$ , e.g.,  $n = 3$ , already lead to very satisfactory results. For example, when trying to show  $S1 \not\subseteq S1^\circ$  with  $F = M6$ , our test runs returned  $m'$  as  $U = \{0, 1, 2\}$  and  $Acc' := U^2 \setminus \{(1, 1), (1, 0)\}$  with the only constant  $p_1$  being true in the worlds 0 and 1 and false in world 2.

Indeed,  $m'$  satisfies all rules of  $\text{S}1^\circ$  (This is always the case if all worlds of  $m'$  are a successor of some world.), and  $M6$  does not hold in world 0.

### 3.5 Algebraic Strategies

In this section, we describe an algebraic strategy for exploring extensions of  $\text{S}1^\circ$ . For a modal logic  $\mathcal{L}$  we construct a Boolean algebra  $\Pi^{\mathcal{L}}$  such that we can convert reasoning about formulae in  $\mathcal{L}$  to algebraic reasoning about  $\Pi^{\mathcal{L}}$ . In general, this procedure could be applied to any modal logic, but we focus on extensions of  $\text{S}1^\circ$ , for which the other strategies are not very successful.

Originally, the idea of using algebraic means to analyze the structure of modal systems appeared in (McKinsey, 1941), and it was further developed by Tarski and Jónsson (Jónsson and Tarski, 1951, 1952).

Since the focus of the paper is on the empirical results, we will only present the main theorems that are required to describe the strategy and only sketches of the proofs. The complete derivation of the theoretical background can be found in (Pudlák, 2006).

#### 3.5.1 Theoretical Basis

First, let us give definitions of a few concepts that we shall use often throughout this section.

**Definition 11** (Strict formulae). We shall call a formula *strict* if its topmost connective is  $\square$  or  $\Rightarrow$ .

One of the defining rules of  $\text{S}1^\circ$  is the substitution of strict equivalents  $EQS$  (recall Definition 2). Therefore, we can factor the set of formulae by strict equivalence and explore the constructed factor. The following lemma summarizes the main properties. This can be proved easily from basic properties of  $\text{S}1^\circ$ , and therefore, we omit the proof.

**Lemma 12.** *Let  $\mathcal{L}$  be an extension of  $\text{S}1^\circ$ . If we construct the (Lindenbaum-Tarski) algebra of  $\mathcal{L}$  by factoring the modal formulae by strict equivalence, then the algebra is a Boolean algebra defined by  $F \cap G = \wedge F G$  and  $\overline{F} = \neg F$ . Its top element  $\top$  is the class of propositional tautologies, and we write  $\top$  to abbreviate any such tautology whose variables are used nowhere else.*

*Furthermore, if we view the algebra as a lattice, the relation  $\mathcal{L} \vdash \Rightarrow FG$  is the ordering of the lattice. In particular,  $\mathcal{L} \vdash \Rightarrow FG$  if and only if  $\mathcal{L} \vdash \Leftrightarrow F \wedge FG$  (which is the same as  $\mathcal{L} \vdash \Leftrightarrow \rightarrow F G \top$ ).*

Looking at extensions  $\mathcal{L}$  of  $S1^\circ$ , our aim is to express  $\mathcal{L} \vdash F$  using strict equivalence. Then we are able to express it as an equality in the algebra. It is not difficult to express the trueness of strict formulae, which can again be proved easily using basic properties of  $S1^\circ$ :

**Lemma 13.**  $S1^\circ \vdash \square F$  if and only if  $S1^\circ \vdash \Leftrightarrow F \top$ .

However, we would like to be able to express trueness of all formulae. Let us first examine the special case that the extension is formed by just strict axioms.

**Lemma 14.** If  $\mathcal{L}$  is an extension of  $S1^\circ$  that can be constructed from  $S1^\circ$  by adding only strict axioms, then the rule  $\frac{F}{\Rightarrow \square \top F}$  (or equivalently  $\frac{F}{\Leftrightarrow \rightarrow \square \top F \top}$ ) is an admissible rule of  $\mathcal{L}$ . In other words,  $\square \top$  is the weakest true formula of the extension with respect to strict implication.

*Proof sketch.* This is shown by induction on the proof of  $F$ . Since all the axioms are strict, the base case follows from Lemma 13, and we omit the induction step.  $\square$

The next lemma shows that if the extension is formed by adding arbitrary axioms, we can add a new logical constant  $\pi$  (a connective of arity 0) that will represent the weakest true formula:

**Lemma 15.** Let  $\mathcal{L}$  be the logic  $S1^\circ$  extended by the axioms  $H_1, \dots, H_n$ . Let us construct an extension  $\mathcal{L}_\pi$  of this logic by adding a new symbol  $\pi$  to the language of  $\mathcal{L}$  and by adding the axiom and the rule

$$A_\pi : \pi \\ R_\pi : \frac{F}{\Rightarrow \pi F} \quad \left( \text{or equivalently} \quad \frac{F}{\Leftrightarrow \rightarrow \pi F \top} \right)$$

Then,  $\mathcal{L}_\pi$  is a conservative extension of  $\mathcal{L}$ , that is if  $F$  does not contain  $\pi$ , then  $\mathcal{L}_\pi \vdash F$  if and only if  $\mathcal{L} \vdash F$ . Moreover,  $\mathcal{L}_\pi \vdash F$  if and only if  $\mathcal{L}_\pi \vdash \Rightarrow \pi F$ , which is the same as  $\mathcal{L}_\pi \vdash \Leftrightarrow \rightarrow \pi F \top$ .

*Proof.* In the proof we shall often replace  $\pi$  in a formula  $F$  by another formula  $G$ .  $F[\pi \rightsquigarrow G]$  is the formula obtained by replacing all occurrences of the symbol  $\pi$  in  $F$  by  $G$ .

We shall first prove two auxiliary propositions and then use them to prove the main statement.

- (1) If  $\mathcal{L}' \vdash F$  in an extension  $\mathcal{L}'$  of  $S1^\circ$  ( $F$  may or may not contain  $\pi$ ) then there is a proof<sup>5</sup> of  $F$  such that the first part of the proof consists only of applications of the rule of substitution for propositional variables to axioms, and the rest of the proof uses only the remaining three rules (substitution of strict equivalents, strict detachment and adjunction).

*Proof.* If we examine the three remaining rules, we see that the rules are closed under substitution for propositional variables. Instead of deriving  $F$  by one of the three rules and then substituting for variables, we can first substitute for variables and then apply the particular rule. Hence, we can propagate all uses of the rule of substitution for variables backwards, until the substitution is performed on only the axioms.

- (2) If we can prove a formula  $F$  in  $\mathcal{L}_\pi$  without using the rule  $R_\pi$ , then there is a formula  $E$  (not containing  $\pi$ ) such that  $\mathcal{L}_\pi \vdash E$  and such that we can prove  $\mathcal{L}_\pi \vdash \Rightarrow \wedge E\pi F[\pi \rightsquigarrow \wedge E\pi]$  without using  $R_\pi$ .

*Proof.* By (1) we can construct a proof of  $F$  of the form

$$\underbrace{G_1, \dots, G_k}_{\text{instances of the axioms}}, \quad , \quad \underbrace{H_1, \dots, H_n}_{\text{only the three remaining rules being used}}$$

where  $H_n = F$ . Let  $E$  be the formula  $\wedge \dots \wedge G_1 \dots G_k$ . This formula is surely true. We shall prove by induction on the length of the proof of  $H_i$  that  $\mathcal{L}_\pi \vdash \Rightarrow \wedge E\pi H_i$  for  $1 \leq i \leq n$ . And moreover, each of the proofs will not use  $R_\pi$ .

It is clear that  $\mathcal{L}_\pi \vdash \Rightarrow \wedge E\pi G_j$  for all  $G_j$ s. By examining the all possible rules we show that  $\mathcal{L}_\pi \vdash \Rightarrow \wedge E\pi H_i$  assuming that it is true for all  $H_j$ s ( $1 \leq j < i$ ). Now, since  $\mathcal{L}_\pi \vdash \wedge E\pi$  and since we have never used the rule  $R_\pi$ , we can replace  $\pi$  by  $\wedge E\pi$  and get a proof of the formula

$$\Rightarrow \wedge E \wedge E\pi F[\pi \rightsquigarrow \wedge E\pi] \equiv \Rightarrow \wedge \wedge E E\pi F[\pi \rightsquigarrow \wedge E\pi] \equiv \Rightarrow \wedge E\pi F[\pi \rightsquigarrow \wedge E\pi].$$

We now prove the main statement of the lemma. Let  $F$  be a formula proved inside  $\mathcal{L}_\pi$  such that  $F$  does not contain the symbol  $\pi$ . We shall show that  $F$  can also be proved just inside  $\mathcal{L}$ .

First, we use induction on the number of applications of  $R_\pi$  to prove that if  $G$  is any formula provable inside  $\mathcal{L}_\pi$  then there is a formula  $D$  such that we can construct a proof of  $G[\pi \rightsquigarrow D]$  without using  $R_\pi$ . Let  $H_1, \dots, H_n$  be the proof of  $G$  and let  $H_i$  be the first application of the rule  $R_\pi$ . Thus,  $H_i$  is  $\Rightarrow \pi H_j$

---

<sup>5</sup> By a proof of  $F$ , we mean a sequence  $G_1, \dots, G_n$  such that  $G_n = F$  and each  $G_i$  is either the axiom of  $\mathcal{L}$  or  $G_i$  is derived from some of  $G_1, \dots, G_{i-1}$  using one of the rules of  $\mathcal{L}$ .

where  $1 \leq j < i \leq n$ . By (2) we can find a formula  $E$  and construct a proof of  $\Rightarrow \wedge E\pi H_j[\pi \rightsquigarrow \wedge E\pi]$  without using  $R_\pi$ . Then the sequence

$$\left( \text{proof of } \Rightarrow \wedge E\pi H_j[\pi \rightsquigarrow \wedge E\pi] \right), H_1[\pi \rightsquigarrow \wedge E\pi], \dots, H_n[\pi \rightsquigarrow \wedge E\pi]$$

is a proof of  $G[\pi \rightsquigarrow \wedge E\pi]$ . If some  $H_k$  ( $1 \leq k \leq n$ ) is the axiom  $A_\pi : \pi$ , then  $H_k[\pi \rightsquigarrow \wedge E\pi] \equiv \wedge E\pi$  is a provable formula, and if some  $H_k \Rightarrow \pi H_m$  is the result of the application of the rule  $R_\pi$  to some formula  $H_m$ , then  $H_k[\pi \rightsquigarrow \wedge E\pi] \Rightarrow \wedge E\pi H_m[\pi \rightsquigarrow \wedge E\pi]$ . To prove it, we apply  $R_\pi$  to  $H_m[\pi \rightsquigarrow \wedge E\pi]$ , get  $\mathcal{L}_\pi \vdash \Rightarrow \pi H_m[\pi \rightsquigarrow \wedge E\pi]$ , and by combining it with  $\mathcal{L}_\pi \vdash \Rightarrow \wedge E\pi \pi$  we get  $\mathcal{L}_\pi \vdash \Rightarrow \wedge E\pi H_m[\pi \rightsquigarrow \wedge E\pi]$ .

Recall that  $H_i$  is the first result of the application of  $R_\pi$ . Since  $H_i[\pi \rightsquigarrow \wedge E\pi]$  is just  $\Rightarrow \wedge E\pi H_j[\pi \rightsquigarrow \wedge E\pi]$ , we have proved  $G[\pi \rightsquigarrow \wedge E\pi]$  using one less application of  $R_\pi$ . By induction hypothesis, we can then prove  $G[\pi \rightsquigarrow D]$  for some formula  $D$  without using  $R_\pi$  at all.

Thus, since the formula  $F$ , whose proof we are looking for, does not contain  $\pi$ , we can construct a proof  $F_1, \dots, F_n, F$  of  $F$  without using the rule  $R_\pi$ . Now we replace  $\pi$  by an arbitrary axiom (we choose  $M4$ ) and the sequence

$$F_1[\pi \rightsquigarrow \Rightarrow p \wedge pp], \dots, F_n[\pi \rightsquigarrow \Rightarrow p \wedge pp], F$$

is a proof of  $F$  without  $\pi$  at all, hence a proof within  $\mathcal{L}$ .  $\square$

The extensions with the added symbol  $\pi$  have one significant disadvantage – rules cannot be disproved. If we prove that a rule is an admissible rule of  $\mathcal{L}_\pi$ , then it is surely an admissible rule of  $\mathcal{L}$ . But the case where a rule is not an admissible rule of  $\mathcal{L}_\pi$  is problematic. For example, the necessitation rule  $\frac{F}{\Box F}$  is an admissible rule of  $S4$ , but not a rule of  $S4_\pi$  since we know nothing about  $\Box \pi$ . Clearly, finding out that  $\frac{F}{\Box F}$  is not a rule of  $S4_\pi$  gives no information about admissibility of the rule in  $S4$ .

Therefore, we add  $\pi$  whenever possible and finally obtain the following theorem as the basis of our strategy.

**Theorem 16.** *Let  $\mathcal{L}$  be the logic  $S1^\circ$  extended with the axioms  $H_1, \dots, H_n$ . Let  $\Pi^{\mathcal{L}}$  be the free algebra defined by the theory  $Def$  given in Section 2.3 extended with constants  $true$  and  $\pi$ , and the following axioms<sup>6</sup> (where, for brevity, we*

---

<sup>6</sup> The relation (1) is an implication of equations. Thus, these equational relations do not form a variety but a quasi-variety.

omit the universal quantifiers):

$$\begin{aligned}
& \text{and}(X, Y) = \text{and}(Y, X) \\
& \text{and}(X, \text{and}(Y, Z)) = \text{and}(\text{and}(X, Y), Z) \\
& \text{and}(X, \text{or}(X, Y)) = X \\
& \text{and}(X, \text{or}(Y, Z)) = \text{or}(\text{and}(X, Y), \text{and}(X, Z)) \\
& \text{true} = \text{not}(\text{and}(X, \text{not}(X))) \\
& \text{impl}(\text{and}(\text{s\_impl}(X, Y), \text{s\_impl}(Y, Z)), \text{s\_impl}(X, Z)) = \text{true} \\
& \text{impl}(\pi, \text{necess}(\text{true})) = \text{true} \\
& \text{impl}(\pi, \text{necess}(X)) = \text{true} \rightarrow X = \text{true} \\
& \text{impl}(\pi, \varepsilon(H_1)) = \text{true} \\
& \vdots \\
& \text{impl}(\pi, \varepsilon(H_n)) = \text{true}
\end{aligned} \tag{1}$$

(1) If all the axioms  $H_1, \dots, H_n$  are strict, we also add the equation

$$\pi = \text{necess}(\text{true})$$

Then  $\mathcal{L} \vdash A$  for a formula  $A$  if and only if

$$\Pi^{\mathcal{L}} \vdash^{FOL} \text{impl}(\text{necess}(\text{true}), \varepsilon(A)) = \text{true}$$

(2) If some of the axioms are not strict, then  $\mathcal{L} \vdash A$  if and only if

$$\Pi^{\mathcal{L}} \vdash^{FOL} \text{impl}(\pi, \varepsilon(A)) = \text{true}$$

*Proof sketch.* The axiom  $A_{\pi} : \pi$  and the rule  $R_{\pi} : \frac{F}{\Rightarrow \pi F}$  in the extension  $\mathcal{L}_{\pi}$  from Lemma 15 together with the rule of strict detachment guarantee that deriving  $\mathcal{L}_{\pi} \vdash \Rightarrow \pi G$  is equivalent to deriving  $\mathcal{L}_{\pi} \vdash G$  and hence equivalent to  $\mathcal{L} \vdash G$ , if  $G$  does not contain  $\pi$ . If in addition all the axioms  $H_1, \dots, H_n$  are strict then the conditions of Lemma 14 are satisfied and we can explicitly set  $\pi = \text{necess}(\text{true})$ .

It can be easily proved that for all these equations the corresponding equivalences are true in the corresponding system  $\mathcal{L}_{\pi}$ . Rule (1) is just Lemma 13. The rule of substitution of strict equivalents justifies combining equivalences in extensions of  $S1^\circ$  just in the same way as equations, therefore anything we derive from the equations can be derived as an equivalence within  $\mathcal{L}_{\pi}$  as well.

Now, let us prove the opposite, that if  $\mathcal{L} \vdash G$  then we can derive

$$\text{impl}(\pi, \varepsilon(G)) = \text{true}$$

using the equations. We shall prove that by induction on the number of steps of the proof of a formula  $G$ . This is trivial for the additional axioms  $H_1, \dots, H_n$

of  $\mathcal{L}$  and it can be also easily shown for the axioms  $M1\text{--}M5$  of  $S1^\circ$ . We then complete the proof by examining the last rule from the proof of  $G$  and showing that if we can derive the equality for all preceding formulae in the proof then we can derive the equality for  $G$ .  $\square$

### 3.5.2 Implementation

The algebraic strategy for an axiomatization  $\mathcal{L}$  with axioms  $\mathcal{A}$  and other rules  $\mathcal{R}$  is prepared by the following steps which are executed in the preprocessing phase:

- (1) Try to prove all the axioms and all the rules of  $S1^\circ$  from  $\mathcal{L}$ . If successful, then  $\mathcal{L} = S1^\circ + \mathcal{A} + \mathcal{R}$ .
- (2) For every rule  $R \in \mathcal{R}$  try to prove  $R$  from  $S1^\circ + \mathcal{A}$ . If successful, then  $\mathcal{L} = S1^\circ + \mathcal{A}$ .
- (3) Try to prove that the rule  $\frac{\Box F}{F}$  is admissible in  $S1^\circ + \mathcal{A}$ .
- (4) Construct  $\mathcal{A}'$  from  $\mathcal{A}$  as follows: For every every axiom  $F \in \mathcal{A}$  that is not strict, prove  $S1^\circ + \mathcal{A} \vdash \Box F$  and replace  $F$  in  $\mathcal{A}$  with  $\Box F$ . If successful, then  $\mathcal{L} = S1^\circ + \mathcal{A}'$ .

The mentioned proofs are attempted using the direct proving strategy. Then Theorem 16 yields the soundness of the following strategy, which is called to prove or disprove  $R$  from  $\mathcal{L}$ :

- If steps 1 to 4 have been successful, construct the algebra  $\Pi^{S1^\circ + \mathcal{A}'}$  with the additional equation  $\pi = \text{necess}(\text{true})$ . Call a first-order theorem prover or model finder to prove or disprove  $R$ , respectively.
- If only steps 1 and 2 have been successful, construct the algebra  $\Pi^{S1^\circ + \mathcal{A}}$  (without the additional equation). If  $R$  is an axiom, call a first-order theorem prover or model finder to prove or disprove  $R$ , respectively. If  $R$  is not an axiom, call a theorem prover to prove  $R$  (i.e., the strategy is not applicable for disproving rules).

## 4 Results

We ran our implementation on all 121 pairs of axiomatizations of the modal logic challenge on a machine with an 3.0GHz PC with 1GB memory, running Linux 2.6. For the proving strategies, we used the prover VAMPIRE 7.45 (Riazanov and Voronkov, 2002) with a time limit of five minutes, and for the disproving strategies we used the model finder PARADOX 1.3 (Claessen and Sorensson, 2003) with 8 elements per model for the direct and algebraic strategies and 3 worlds per Kripke-model for the Kripke-based strategy. All tools

were used with default settings. To compare the strategies against each other, we repeated the experiment three more times switching off the Kripke-based or the algebraic strategies or both, respectively.

The results are given in Fig. 6. For the run with all strategies switched on, we took the run time. First all axiomatizations went through the preprocessing which was timed independently. Then for every pair  $(\mathcal{L}, \mathcal{M})$  of axiomatizations, both directions of the comparison were run and timed separately. The results of (dis-)proving  $\mathcal{M}$  from  $\mathcal{L}$  are given in row  $\mathcal{L}$ , column  $\mathcal{M}$ . Remember that when (dis-)proving  $\mathcal{M}$  from  $\mathcal{L}$ , the system tries to prove every rule or axiom of  $\mathcal{M}$  from  $\mathcal{L}$  trying every applicable proving strategy. The strategies were applied in the order Kripke-based, algebraic, direct. If that fails, it tries to disprove the relationship.

It can be seen that the system can solve all but six instances of the challenge. In all five attempted derivations of K-based axiomatizations from  $S5_\alpha$ , which failed even when all strategies were used, the direct strategies failed only because the induction step for the axiom  $B$  in the derivation of  $Nec$  in the strategy `direct_ind_pos` failed. Thus normality could not be established either, and the Kripke-based strategies could not be applied. The algebraic strategy was not successful either because the  $S1^\circ$ -based axiomatization involves a non-strict axiom, which makes it less efficient. Note that because  $S4_\alpha$  does not have the axiom  $B$ , we could prove more inclusions from  $S4_\alpha$  than from the stronger system  $S5_\alpha$ . The sixth failing case is to disprove  $S1^\circ$  from K. Here the problem is that only those axioms and rules that could not be proved are used as potential counter-examples. A stronger strategy could apply the unproved rules to generate more formulae that may be counter-examples.

The preprocessing times are very high because the preprocessing already involves proving tasks, and every failed proving attempt takes five minutes. In particular, the ultimately failing attempts to establish normality lead to very high preprocessing times. On the other hand, this significantly reduces the run time spent in the comparison phases.

The execution time for the comparisons where the inclusion must be disproved is extremely high. This was to be expected because disproving is tried only after all proving strategies have failed. A reimplementation should switch between trying to prove and disprove the inclusion. Due to the preprocessing, when the inclusion can be proved, the execution time is either very small or medium. This mainly depends on how often a strategy is invoked that fails. For example, proving an  $S1^\circ$ -based axiomatization from itself can take surprisingly long because the algebraic strategy may time out for one rule, which is then proved instantaneously by the direct strategy. All proved inclusions take less than 900 seconds, i.e., there are at most two failing proving attempts.

| P               | K  | S1°                               | S1                               | T                                   | S3                                  | S4 <sub>α</sub>                     | S4 <sub>β</sub>                     | S5 <sub>α</sub>                     | S5 <sub>β</sub>                     | S5 <sub>γ</sub>                     | S5 <sub>δ</sub>                     |                                       |
|-----------------|----|-----------------------------------|----------------------------------|-------------------------------------|-------------------------------------|-------------------------------------|-------------------------------------|-------------------------------------|-------------------------------------|-------------------------------------|-------------------------------------|---------------------------------------|
| K               | 16 | $\subseteq_0 \bullet\bullet$      | $\not\subseteq \circ\circ$       | $\not\subseteq_{2110} \bullet\circ$ | $\not\subseteq_{301} \circ\bullet$  | $\not\subseteq_{2410} \bullet\circ$ | $\not\subseteq_{2710} \bullet\circ$ | $\not\subseteq_{602} \bullet\circ$  | $\not\subseteq_{3010} \bullet\circ$ | $\not\subseteq_{902} \bullet\circ$  | $\not\subseteq_{2115} \bullet\circ$ | $\not\subseteq_{602} \bullet\circ$    |
| S1°             | 61 | $\not\subseteq_{3339} \circ\circ$ | $\subseteq_{319} \bullet\bullet$ | $\subseteq_{932} \bullet\bullet$    | $\not\subseteq_{3954} \bullet\circ$ | $\not\subseteq_{1537} \bullet\circ$ | $\not\subseteq_{2145} \bullet\circ$ | $\not\subseteq_{4553} \bullet\circ$ | $\not\subseteq_{2736} \bullet\circ$ | $\not\subseteq_{5166} \bullet\circ$ | $\not\subseteq_{934} \bullet\circ$  | $\not\subseteq_{4551} \bullet\bullet$ |
| S1              | 55 | $\not\subseteq_{2915} \circ\circ$ | $\subseteq_{349} \bullet\bullet$ | $\subseteq_{348} \bullet\bullet$    | $\not\subseteq_{2927} \bullet\circ$ | $\not\subseteq_{963} \bullet\circ$  | $\not\subseteq_{1564} \bullet\circ$ | $\not\subseteq_{3535} \bullet\circ$ | $\not\subseteq_{2164} \bullet\circ$ | $\not\subseteq_{4136} \bullet\circ$ | $\not\subseteq_{963} \bullet\circ$  | $\not\subseteq_{3536} \bullet\bullet$ |
| T               | 14 | $\subseteq_0 \bullet\bullet$      | $\subseteq_0 \bullet\circ$       | $\subseteq_1 \circ\circ$            | $\subseteq_0 \bullet\bullet$        | $\not\subseteq_{301} \bullet\circ$  | $\not\subseteq_{602} \bullet\circ$  | $\not\subseteq_{301} \bullet\circ$  | $\not\subseteq_{902} \bullet\circ$  | $\not\subseteq_{602} \bullet\circ$  | $\not\subseteq_{301} \bullet\circ$  | $\not\subseteq_{301} \bullet\circ$    |
| S3              | 52 | $\not\subseteq_{3118} \circ\circ$ | $\subseteq_{349} \bullet\bullet$ | $\subseteq_{349} \bullet\bullet$    | $\not\subseteq_{2909} \bullet\circ$ | $\subseteq_{350} \bullet\bullet$    | $\not\subseteq_{968} \bullet\circ$  | $\not\subseteq_{3524} \bullet\circ$ | $\not\subseteq_{1569} \bullet\circ$ | $\not\subseteq_{4124} \bullet\circ$ | $\not\subseteq_{964} \bullet\circ$  | $\not\subseteq_{3521} \bullet\bullet$ |
| S4 <sub>α</sub> | 24 | $\subseteq_{312} \bullet\bullet$  | $\subseteq_{358} \bullet\bullet$ | $\subseteq_{358} \bullet\bullet$    | $\subseteq_{322} \bullet\circ$      | $\subseteq_{359} \bullet\bullet$    | $\subseteq_{359} \bullet\bullet$    | $\not\subseteq_{334} \bullet\circ$  | $\not\subseteq_{974} \bullet\circ$  | $\not\subseteq_{949} \bullet\circ$  | $\not\subseteq_{973} \bullet\circ$  | $\not\subseteq_{938} \bullet\bullet$  |
| S4 <sub>β</sub> | 11 | $\subseteq_0 \bullet\bullet$      | $\subseteq_0 \bullet\circ$       | $\subseteq_1 \circ\circ$            | $\subseteq_0 \bullet\bullet$        | $\subseteq_1 \bullet\circ$          | $\subseteq_1 \bullet\circ$          | $\subseteq_1 \bullet\circ$          | $\not\subseteq_{417} \bullet\circ$  | $\not\subseteq_{419} \bullet\circ$  | $\not\subseteq_{431} \bullet\circ$  | $\not\subseteq_{429} \bullet\circ$    |
| S5 <sub>α</sub> | 54 | $\subseteq \circ\circ$            | $\subseteq_{345} \bullet\bullet$ | $\subseteq_{344} \bullet\bullet$    | $\subseteq \circ\circ$              | $\subseteq_{358} \bullet\bullet$    | $\subseteq_{343} \bullet\bullet$    | $\subseteq \circ\circ$              | $\subseteq_{343} \bullet\bullet$    | $\subseteq \circ\circ$              | $\subseteq_{447} \bullet\bullet$    | $\subseteq \circ\circ$                |
| S5 <sub>β</sub> | 18 | $\subseteq_0 \bullet\bullet$      | $\subseteq_1 \bullet\circ$       | $\subseteq_1 \circ\circ$            | $\subseteq_1 \bullet\bullet$        | $\subseteq_1 \bullet\circ$          | $\subseteq_1 \bullet\bullet$          |
| S5 <sub>γ</sub> | 27 | $\subseteq_{376} \bullet\circ$    | $\subseteq_{340} \bullet\bullet$ | $\subseteq_{439} \bullet\bullet$    | $\subseteq_{475} \bullet\circ$      | $\subseteq_{544} \bullet\bullet$    | $\subseteq_{646} \bullet\bullet$    | $\subseteq_{576} \bullet\circ$      | $\subseteq_{747} \bullet\circ$      | $\subseteq_{678} \bullet\circ$      | $\subseteq_{338} \bullet\bullet$    | $\subseteq_{475} \bullet\circ$        |
| S5 <sub>δ</sub> | 15 | $\subseteq_0 \bullet\bullet$      | $\subseteq_0 \bullet\circ$       | $\subseteq_1 \circ\circ$            | $\subseteq_1 \bullet\bullet$        | $\subseteq_1 \bullet\circ$          | $\subseteq_1 \bullet\bullet$          |

P: Preprocessing time in minutes

$\subseteq$  or  $\not\subseteq$  in row  $\mathcal{L}$ , column  $\mathcal{M}$ :  $\mathcal{M} \subseteq \mathcal{L}$  or  $\mathcal{M} \not\subseteq \mathcal{L}$ , respectively

number: (dis-)proving time in seconds

• or  $\circ$ : system returned correct answer or failed, respectively

bottom symbol: only direct strategies used

middle left symbol: direct and Kripke-based strategies used

middle right symbols: direct and algebraic strategies used

top symbol: all strategies used

Fig. 6. Experimental Results

When comparing the strategies, we find that the Kripke-based and the algebraic strategy complement each other nicely. This is not surprising since the former is strong for normal logics and the latter for S1°-based axiomatizations. It cannot be seen from the table that the direct proving strategy was not superfluous: Apart from being needed to establish applicability of the other two more sophisticated strategies, it occasionally succeeded when the other ones failed, e.g., in the example above. Furthermore, the inductive direct strategy `direct_ind_pos` was often needed to prove the necessitation rule.

The disproving results show that the direct disproving strategy was never

successful. The reason for the failure is that only finite models are considered, while the first-order universe of the model needs to contain an interpretation of every formula. Using Herbrand models promises to be a more successful strategy, which is possible using the DARWIN model finder (Baumgartner et al., 2005). However, since the other two strategies were so successful, we have not pursued this. In general, we were surprised to find the disproving cases to be the much simpler than the proving cases.

## 5 Conclusion and Future Work

We have presented a system that approaches the open challenge problem of automatically determining the subset relationship between modal logics. The correctness of the system is based on theoretical development that in turn depends on successful proofs, in order to admit the various preprocessing steps (e.g., the proofs that show equivalence of the four axiomatizations of PC) and comparison strategies (e.g., proofs of the congruence rules to admit efficient equational reasoning). The full system has been tested on 121 pairs of 11 axiomatizations of 7 common modal logics. Only six cases could not be solved because of, in total, two failing subcases, thus obtaining a high degree of empirical success.

Future work will focus mainly on improving the efficiency and usability of the system. It may prove useful to develop heuristics that govern the order of strategy application. The system should switch between trying to prove and trying to disprove an inclusion. It is also promising to conduct experiments in order to further optimize the time limits for proving and the model sizes for disproving attempts or to change these values dynamically.

Only minor improvements of the underlying theoretical results are necessary. In particular, the strategy `direct_neg` should be improved to check infinite models. If there are rules that cannot be proved, they should be applied a few times to generate theorems which can serve as potential counter-examples for the disproving strategies. It may also be worthwhile to investigate whether an algebraic treatment of normal logics is more powerful than using Kripke semantics. Of course, it is generally interesting to consider integrating more strategies, e.g., the decidability results of Rybakov (Rybakov, 1997), if they can be formulated to apply to big classes of logics with decidable applicability conditions.

## References

- Baumgartner, P., Fuchs, A., Tinelli, C., 2005. Implementing the Model Evolution Calculus. In: Schulz, S., Sutcliffe, G., Tammet, T. (Eds.), Special Issue of the International Journal of Artificial Intelligence Tools (IJAIT). Vol. 15 of International Journal of Artificial Intelligence Tools.
- Bernays, P., 1926. Axiomatische Untersuchungen des Aussagenkalküls der Principia Mathematica. *Mathematische Zeitschrift* 25, 305–320.
- Blackburn, P., de Rijke, M., Venema, Y., 2001. Modal logic. Cambridge University Press, New York, NY, USA.
- Claessen, K., Sorensson, N., 2003. New techniques that improve MACE-style finite model finding. In: CADE-19 Workshop on Model Computation - Principles, Algorithms, Applications.
- Fitting, M., Mendelsohn, R., 1998. First-Order Modal Logic. Kluwer.
- Gabbay, D., Ohlbach, H., 1992. Quantifier elimination in second-order predicate logic. In: Nebel, B., Rich, C., Swartout, W. (Eds.), Principles of Knowledge Representation and Reasoning (KR92). Morgan Kaufmann, pp. 425–435.
- Giunchiglia, E., Giunchiglia, F., Tacchella, A., 2002. SAT-Based Decision Procedures for Classical Modal Logics. *Journal of Automated Reasoning* 28 (2), 143–171.
- Goranko, V., Hustadt, U., Schmidt, R., Vakarelov, D., 2004. SCAN is complete for all Sahlqvist formulae. In: Berghammer, R., Moller, B., Struth, G. (Eds.), Relational and Kleene-Algebraic Methods in Computer Science. pp. 149–162.
- Goré, R., Heinle, W., Heuerding, A., 1997. Relations between propositional normal modal logics: An overview. *Journal of Logic and Computation* 7 (5), 649–658.
- Halleck, J., 2006. Logic Systems. WWW pages, see <http://www.cc.utah.edu/~nahaj/logic/structures/systems/index.html>.
- Hilbert, D., Bernays, P., 1934. Grundlagen der Mathematik. Julius Springer Verlag.
- Hughes, G., Cresswell, M., 1996. A New Introduction to Modal Logic. Routledge.
- Hustadt, U., Schmidt, R. A., 2000. MSPASS: Modal Reasoning by Translation and First-Order Resolution. In: Dyckhoff, R. (Ed.), Automated Reasoning with Analytic Tableaux and Related Methods, International Conference (TABLEAUX 2000). pp. 67–71.
- Iemhoff, R., Metcalfe, G., 2007. Proof theory for admissible rules. Preprint submitted to <http://www.phil.uu.nl/preprints/lgps/>.
- Jónsson, B., Tarski, A., 1951. Boolean algebras with operators, I. *Amer. J. Math.* 73, 891–939.
- Jónsson, B., Tarski, A., 1952. Boolean algebras with operators, II. *Amer. J. Math.* 74, 127–162.
- Kracht, M., 1990. An almost general splitting theorem for modal logic. *Studia*

- Logica 49 (4), 455–470.
- Kripke, S., 1963. Semantical analysis of modal logic I. Normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik* 9, 67–96.
- Lemmon, E., 1966. Algebraic Semantics for Modal Logics II. *The Journal of Symbolic Logic* 31, 191–218.
- Lewis, C., Langford, C., 1932. *Symbolic Logic*. The Century Co, New York and London.
- Lukasiewicz, J., 1963. *Elements of Mathematical Logic*. Pergamon Press.
- McCune, W., Veroff, R., Fitelson, B., Harris, K., Feist, A., Wos, L., 2002. Short single axioms for boolean algebra. *Journal of Automated Reasoning archive* 29 (1), 1–16.
- McCune, W., Wos, L., 1992. Experiments in automated deduction with condensed detachment. In: CADE-11: Proceedings of the 11th International Conference on Automated Deduction. Springer, pp. 209–223.
- McKinsey, J. C., December 1941. A solution of the decision problem for the Lewis systems S2 and S4 with an application to topology. *The Journal of Symbolic Logic* 6 (4), 117–134.
- Ohlbach, H., Schmidt, R., 1997. Functional translation and second-order frame properties of modal logics. *Journal of Logic and Computation* 7 (5), 581–603.
- Pudlák, P., 2006. Verification of mathematical proofs. Ph.D. thesis, Charles University in Prague, Faculty of Mathematics and Physics, <http://lipa.ms.mff.cuni.cz/~pudlak/pp-thesis.ps.gz>.
- Rabe, F., 2006a. Determining the Subset Relation between Propositional Modal Logics. See <http://kwarc.eecs.iu-bremen.de/frabe/Research/moloss/index.html>.
- Rabe, F., 2006b. Towards Determining the Subset Relation between Propositional Modal Logics. In: Sutcliffe, G., Schmidt, R., Schulz, S. (Eds.), *Proceedings of the FLoC 06 Workshop on Empirically Successful Computerized Reasoning*, 3rd International Joint Conference on Automated Reasoning. Vol. 192 of CEUR Workshop Proceedings. pp. 126–140.
- Riazanov, A., Voronkov, A., 2002. The design and implementation of Vampire. *AI Communications* 15, 91–110.
- Rosser, J., 1953. *Logic for Mathematicians*. McGraw-Hill.
- Russell, B., Whitehead, A., 1910. *Principia Mathematica*. Cambridge University Press.
- Rybakov, V., 1997. *Admissibility of Logical Inference Rules*. North-Holland.
- Sahlqvist, H., 1975. Completeness and Correspondence in the First and Second Order Semantics for Modal Logic. In: Kanger, S. (Ed.), *Proceedings of the Third Scandinavian Logic Symposium*. North-Holland, pp. 110–143.
- Schmidt, R., Hustadt, U., 2006. First-Order Resolution Methods for Modal Logics. To appear in Volume in memoriam of Harald Ganzinger.
- SML, 2007. Standard ML of New Jersey. See <http://www.smlnj.org>.
- Sutcliffe, G., 2006. The Modal Logic \$100 Challenge. See <http://www.cs>.

- `miami.edu/~tptp/HHDC/.`
- Sutcliffe, G., Suttner, C., 1998. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning* 21 (2), 177–203.
- Thomason, S., 1974. An incompleteness theorem in modal logic. *Theoria* 40, 30–34.
- Zeman, J. J., 1973. Modal Logic, the Lewis-Modal systems. Oxford University Press, <http://www.clas.ufl.edu/users/jzeman/modallogic/>.

# Semantics of OpenMath and MathML3

Michael Kohlhase and Florian Rabe

**Abstract.** Even though OPENMATH has been around for more than 10 years, there is still confusion about the “semantics of OPENMATH”. As the recent MATHML3 recommendation semantically bases Content MATHML on OPENMATH Objects, this question becomes more pressing.

One source of confusions about OPENMATH semantics is that it is given on two levels: a very weak algebraic semantics for expression trees, which is extended by considering mathematical properties in content dictionaries that interpret the meaning of (constant) symbols. While this two-leveled way to interpret objects is well-understood in logic, it has not been spelt out rigorously for OPENMATH.

We present two denotational semantics for OPENMATH: a construction-oriented semantics that achieves full coverage of all legal OPENMATH expressions at the cost of great conceptual complexity, and a symbol-oriented one for a subset of OPENMATH expressions. This subset is given by a variant of the OPENMATH 2 role system, which – we claim – does not exclude any representations of meaningful mathematical objects.

## 1. Introduction

MATHML [ABC<sup>+03</sup>] and OPENMATH [BCC<sup>+04</sup>] are standards for the representation and communication of mathematical objects. Even though they have been around for more than 10 years, there is still confusion about the “semantics of OPENMATH”. As the recent MATHML3 recommendation [ABC<sup>+10a</sup>] semantically bases Content MATHML on OPENMATH Objects, this question becomes more pressing.

### 1.1. OpenMath and MathML

MATHML comes in two parts: *presentation* MATHML, which provides XML-based layout primitives for the traditional two-dimensional notation of mathematical formulae and *content* MATHML, which focuses on encoding the meaning of objects

rather than visual representations to allow the free exchange of mathematical objects between software systems and human beings. OPENMATH has the same goals as content MATHML, but was developed by a different community with slightly different intuitions. Both representation formats represent mathematical objects as expression trees. Content MATHML tries to cover all of school and engineering mathematics (the “K-14” fragment) in a representation format intuitive to mathematicians, and OPENMATH concentrates on an extensible framework built on a minimal structural core language with a well-defined extension mechanism. Where MATHML supplies more than a dozen elements for special constructions, OPENMATH only supplies concepts for function application (`OMA`), binding constructions (`OMBIND`), and attributions (`OMATTR`). Where MATHML provides close to 100 elements for the K-14 fragment, OPENMATH gets by with only an `OMS` element that identifies symbols by pointing to declarations in an open-ended set of Content Dictionaries.

An OPENMATH Content Dictionary (CD) is a document that declares names (OPENMATH “symbols”) for basic mathematical concepts and objects. CDs act as the unique points of reference for OPENMATH symbols (via `OMS` elements) and thus supply a notion of context that situates and disambiguates OPENMATH expression trees. To maximize modularity and reuse, a CD typically contains a relatively small collection of definitions for closely related concepts. The OPENMATH Society maintains a large set of public CDs [OMC], including CDs for all pre-defined symbols in MATHML. There is a process for contributing privately developed CDs to the OPENMATH Society repository to facilitate discovery and reuse. OPENMATH does not require CDs be publicly available, though in most situations the goals of semantic markup will be best served by referencing public CDs available to all user agents.

To avoid fragmentation and to smooth out interoperability obstacles, an effort has been made to align OPENMATH and MATHML semantically. To remedy the lack of regularity and specified meaning in MATHML, content MATHML was extended by concepts like binding structures and full semantic annotations from OPENMATH and a structurally regular subset of the extended content MATHML was identified that is isomorphic to OPENMATH objects. This subset is called **strict content MathML** to contrast it to full content MATHML that is seen to strike a more pragmatic balance between regularity and human readability. Full content MATHML borrows the semantics from strict MATHML by a mapping specified in the MATHML3 specification [ABC<sup>+10a</sup>] that defines the meaning of non-strict (**pragmatic**) MATHML expressions in terms of strict MATHML equivalents. Strict Content MATHML in turn obtains its meaning by being an encoding of OPENMATH Objects.

In this situation, the “meaning of OPENMATH (Objects)” obtains a completely new significance. The aim of this paper is to clarify the status of semantics in OPENMATH (and thus content MATHML3). We observe a presentational gap between how mathematical objects and theories are conventionally given a

meaning and the way OPENMATH answers the question. This leads to misunderstandings about the meaning of OPENMATH objects and its role in representing mathematical knowledge.

## 1.2. The Meaning of OpenMath

The OPENMATH standard actually gives two answers to the question about the meaning of OPENMATH expressions. The first one comes from the fact that OPENMATH is intended as a communication standard between mathematical software systems: OPENMATH envisions communication via *phrasebooks* (see [AvLS98] or [BCC<sup>+</sup>04, chapter 1]): Each mathematical software system  $S$  is equipped with an OPENMATH phrasebook that converts OPENMATH expressions from and to the internal representations of the system  $S$ . In this “system communication view”, the meaning of OPENMATH expressions is built into the phrasebooks that (purport to) understand the expression, and the meaning is whatever  $S$  (after conversion by the phrasebook) makes it to be. Clearly, this view of meaning is not very helpful, and taken in the radical simplicity we have formulated it here is not an adequate account. After all, the purpose of the OPENMATH standard is to synchronize the system-specific representations of objects, so that communication between systems is meaning-preserving. To attain this goal, OPENMATH gives a second answer: It defines the class of “OPENMATH objects” which acts as the model for encodings of mathematical formulae. OPENMATH objects are essentially labeled trees modulo  $\alpha$ -conversion for binding structures and flattening for nested semantic annotations. The OPENMATH standard considers OPENMATH objects as primary citizens and views the “OPENMATH XML encoding” as just an incidental design choice for an XML-based markup language. In fact OPENMATH specifies another encoding: the “binary encoding” designed to be more space efficient at the cost of being less human-readable. Furthermore, phrasebooks are supposed to reconstruct OPENMATH objects from the encodings and only then convert to internal representations.

*OpenMath objects do not specify any computational behavior, they merely represent mathematical expressions. Part of the OpenMath philosophy is to leave it to the application to decide what it does with an object once it has received it. OpenMath is not a query or programming language. Because of this, OpenMath does not prescribe a way of forcing “evaluation” or “simplification” of objects like  $2 + 3$  or  $\sin(\pi)$ . Thus, the same object  $2 + 3$  could be transformed to 5 by a computer algebra system, or displayed as  $2 + 3$  by a typesetting tool.* [BCC<sup>+</sup>04, section 1.5]

From a model-theoretic point of view it makes sense to view the set  $\mathcal{O}$  of OPENMATH objects as an initial algebra for OPENMATH expressions. This algebra is intentionally weak to make the OPENMATH format ontologically unconstrained and thus universally applicable. It basically represents the accepted design choice of representing objects as formulae.

For the internal representations in a system, OPENMATH stipulates that phrasebooks should be informed by (mathematical properties in) content dictionaries, rather than relying on mathematical intuitions:

*It is the OpenMath Content Dictionaries which actually hold the meanings of the objects being transmitted. For example if application A is talking to application B, and sends, say, an equation involving multiplication of matrices, then A and B must agree on what a matrix is, and on what matrix multiplication is, and even on what constitutes an equation. All this information is held within some Content Dictionaries which both applications agree upon. [...] The primary use of Content Dictionaries is thought to be for designers of Phrasebooks, the programs which translate between the OpenMath mathematical object and the corresponding (often internal) structure of the particular application in question.*

[BCC<sup>+</sup>04, section 4.1]

Note that since  $\mathcal{O}$  is initial, it is essentially unique and identifies (in the sense of “declares to be the same”) fewer objects than any other model. As a consequence two mathematical objects must be identical if their OPENMATH representations are, but not the other way around. Any further (meaning-giving) properties of an object  $o$  are relegated to the content dictionaries referenced in  $o$ , where they can be specified formally (as “Formal Mathematical Properties” in FMP elements containing XML-encoded OPENMATH objects) or informally (as “Commented Mathematical Properties” in CMP elements containing text). The precision of OPENMATH as a representation language can be adapted by supplying CDs to range from fully specific (by providing CDs based on some logical system and axioms that fix models up to isomorphism) to fully underspecified (where CDs are essentially empty except for declaring symbols).

### 1.3. Overview of the Paper

When designing a formal language, we have to make a trade-off between expressivity and interpretability. The more flexible we make the language, the more content can be expressed in it; but also, the harder it gets to interpret the language. OPENMATH systematically and intentionally errs on the side of expressivity imposing only minimal well-formedness constraints on syntactic objects in the form of a context-free grammar. Consequently, almost all well-formed OPENMATH objects do not denote a mathematical object and must be interpreted using special values denoting undefinedness. This is very common in logic and not a problem in itself, but it precludes full-coverage, symbol-oriented semantics.

In this paper, we will develop both

1. a full-coverage, construction-oriented semantics for OPENMATH objects, which solves the problem of specifying denotations for OPENMATH’s feature of arbitrary binders and attributions, and

2. a symbol-oriented semantics for a restricted subset of OPENMATH objects that can serve as a compromise between the spirit of OPENMATH and the elegance of a model-theoretical semantics.

The price for giving denotations for all OPENMATH objects (under 1.) is that giving individual algebras is extremely complicated and convoluted. The price for semantic elegance (under 2.) is that we need to find a way to restrict the flexibility of OPENMATH expressions without sacrificing (important) mathematical examples.

We will concentrate on the symbolic core of OPENMATH expressions which lends itself to methods from logic and model theory, excluding the basic OPENMATH objects for arbitrary-precision integers, IEEE floats, character strings, byte arrays, and the derived “foreign objects”, since they pose different (and completely independent) semantic challenges that have been covered elsewhere. Achieving full coverage for the symbolic core of OPENMATH is a challenge of its own as this paper shows.

We will develop the model theory for OPENMATH in two steps. In Section 3, we present an initial algebra semantics of OPENMATH objects, and then in Section 4 extend it to take mathematical properties in CDs into account. In particular, fixing a set of foundational content dictionaries that can take the place of a logic gives us a notion of consistency that can later be exploited to stipulate compliance criteria for OPENMATH-aware applications. In Section 5, we discuss how role/type systems can be used to single out subsets that afford symbol-oriented semantics. We show the feasibility of such an approach by providing a role system that strengthens the one specified in the OPENMATH 2 standard and exhibit a symbol-oriented semantics.

In Section 6.1 we take up the question of the semantics of OPENMATH and content MATHML expressions again and propose two extensions to the OPENMATH standard that would adopt a role system based on the one presented in Section 5, makes our symbol-oriented semantics the intended model for OPENMATH expressions, and extends the compliance chapter of the OPENMATH standard, which surprisingly does not mention content dictionaries so far. Section 7 concludes the paper.

**Acknowledgements.** The research reported in this paper was supported by the German Research Council (DFG) under grant KO 2428/9-1. The authors would like to thank an anonymous reviewer for extremely detailed and insightful comments to the submitted version of the paper.

## 2. Preliminaries and Related Work

### 2.1. Model Theoretical Semantics

Model theoretical semantics of logical languages go back to [TV56, Rob50], an overview is given in [BF85]. They are often based on a **universe**, a set that contains the objects the language is designed to talk about (its domain of discourse).

The denotation of an object of a formal language is usually defined via an **interpretation function**  $\llbracket - \rrbracket$ , an inductive, compositional function on the syntax. An **algebra** or (especially if propositions and truth values are involved) **model** is a universe together with an interpretation function.

Usually, almost all syntactic objects of the language are interpreted as **intra-universal** entities, i.e., as elements of the universe. Generally, **extra-universal** entities are not desirable because they complicate the semantics. Moreover, they can always be avoided by simply enlarging the universe to encompass them. Yet, there are two conditions under which they are useful. Firstly, there should only be few objects with extra-universal semantics. Usually, these objects are atomic, i.e., certain symbols; if they are composed objects, they should be subject to a comparatively simple language. Secondly, adding them to the universe would substantially complicate the universe. This usually means extending all operations that must be defined for all elements of the universe, which can be very inconvenient.

For instance, in first-order logic, the function and predicate symbols are interpreted extra-universally as functions and relations on the universe. Similarly, the truth values, i.e., the denotations of propositions are extra-universal. This is tolerated because there are only finitely many and only atomic objects with extra-universal semantics. Moreover, keeping them out of the universe is desirable because it permits universes without functions.

A contrasting example is the semantics of higher-order logic (a logic based on the simply typed  $\lambda$ -calculus) [Chu40]. There the universe contains truth values and (arbitrary orders) of functions, so that functions and predicates can be treated intra-universally (at the cost of having a much more complicated universe). Moreover, by using higher-order abstract syntax, even the quantifiers can be treated intra-universally as predicates on predicates, and the only remaining language-level constructions with an extra-universal interpretation are application and  $\lambda$ -abstraction.

The interpretation function is defined by induction on the syntax. Thus, at the very least, an algebra must provide one (intra- or extra-)universal entity for each symbol declared in the signature. If the interpretation of complex objects is defined generically in terms of the interpretations of their components (i.e., ultimately in terms of the interpretations of the symbols), we speak of a **symbol-oriented** semantics. The above semantics for first- and higher-order logic are examples, the former using extra-universal interpretations of the symbols.

Alternatively, algebras may additionally provide extra-universal operations that determine how the interpretation of complex objects is obtained from the interpretation of their components. In that case, we speak of a **construction-oriented** semantics. In this case, the universe is usually so rich that these operations are the only extra-universal entities needed. Consequently, the universe of the algebras is more complex, but once given, all symbols of the signature can easily be interpreted intra-universally.

For example, a construction-oriented semantics for higher-order logic can be given based on applicative structures, which consist of a universe together with

an application operator. Thus, the semantics of application can vary with the particular algebra. In categorical logic, this is taken to the extreme by searching for classes of categories that provide just enough structure to interpret all the constructions. For example, for higher-order logic, this leads to algebras based on arbitrary cartesian closed categories [LS86].

## 2.2. Role and Type Systems

In a **classified** language, the syntactic objects are grouped into classes, algebras provide one universe for each syntactic class, and objects are interpreted as elements of the universe associated with their class. Often some objects belong to no class (usually called ill-formed), which are not interpreted at all. We call the opposite case of a semantics with a single universe **monolithic**.

Classified languages are most common in type theory [WR13, Chu40], where the objects are classified by their type and algebras provide a different universe for every type.

A classified syntax can be achieved using role systems or type systems. Both assume a syntactic class assigned to each symbol (the symbol's role or type) and then extend this assignment to all objects. Moreover, objects in certain syntactic positions are constrained to have a certain role or type so that many objects become ill-formed. We speak of a **role system** if the syntactic classes are extraneous to the formal language and of a **type system** if they are themselves syntactic objects.

In order to discuss role and type system for OPENMATH, we distinguish positive and negative systems. A **negative** system defines some objects to be *ill-formed* and leaves the status of the remaining objects open, e.g., objects with a certain syntactic class may be forbidden in certain positions. A **positive** system, on the other hand, defines some objects to be *well-formed*, e.g., only objects with a certain syntactic class may be allowed in a certain position. A positive system permits restricting attention to the set of well-formed objects, which greatly simplifies the semantics. For example, if all well-formed binders are formed from certain symbols in a specific way, we can give a symbol-oriented semantics of binding objects.

We call a system **sound** if all well-formed objects are meaningful and **complete** if all meaningful objects are well-formed. In a non-sound system, the semantics must provide a special undefined value to interpret the meaningless objects. Incompleteness is often necessary to achieve decidability of well-formedness.

Furthermore, we distinguish universal and limited system. A **universal** system applies to arbitrary OPENMATH objects, and a **limited** system only to a subset of OPENMATH objects. In particular, any decidable type system limits attention to a decidable fragment of mathematical objects. For example, we might limit attention to the case where there is only one key (for type attributions) and only two objects permitted as binders (two symbols for  $\lambda$  and  $\Pi$ ).

These distinctions lead to a trade-off between the desirable properties of being positive and universal. Usually, role systems are relatively simple, making this trade-off easier. Type systems are usually much more fine-grained than role

systems, and a positive and universal type system for OPENMATH is virtually impossible.

**Role and Type Systems for OpenMath.** Moreover, the design of OPENMATH intentionally avoids a commitment to any particular role or type system. Still, both role and type systems have been given for OPENMATH.

The OPENMATH standard [BCC<sup>+</sup>04] already provides a role system. The possible roles are `constant`, `application`, `binder`, `key`.<sup>1</sup> The role system is negative: OM objects are ill-formed if their head does not have the respective role. In particular, no restriction is imposed for symbols without role or for composed objects. (Thus, the effect is marginal because the role restriction can be circumvented altogether by wrapping a symbol in a non-semantic attribution, which the standard guarantees not to change the semantics.) Therefore, the system is universal: If no roles are assigned, all objects are well-formed. Consequently, the system is complete but not sound.

In [RK09], we give a more fine-grained role system that uses arities for functions. The system is positive while being as universal as possible. For example, the default arity is that of a function with unlimited arity. Moreover, the binding objects are positively limited: The only permitted binders are symbols with that role and applications of such symbols to arguments.

Neither role system limits the bound variables of binding objects, which would be crucial for a symbol-oriented semantics.

In [Dav99], a simple sound type system is given. It is positive and intentionally limited to basic cases. It is essentially an extension of simple type theory with  $n$ -ary and associative-binary function arguments.

In [CC98], a stronger type system is given based on the calculus of constructions. It is also positive, sound, and limited – in particular, type checking is decidable – but much less limited than the above. It features all possible  $\Pi$ -types of the  $\lambda$ -cube as well as dependent product types. Attributions are limited to a single key for type attributions. Other binders than  $\Pi$ ,  $\lambda$ , and  $\Sigma$  are permitted, but these are considered abbreviations of the corresponding functions defined in terms of higher-order abstract syntax.

A similar type system based on categorial types is sketched in [Str04]. It uses a special key for type attributions and represents binding using higher-order abstract syntax. Keys are treated like binary functions.

### 3. A Construction-Oriented Semantics for OpenMath

We will now define a an algebraic semantics for OPENMATH objects building on ideas from [BBK04]. The difference to the situation there (giving a semantics for the simply typed  $\lambda$  calculus with a type of Booleans) is that OPENMATH allows

---

<sup>1</sup>We do not cover the roles `attribution` and `error` here, which go beyond the fragment of OPENMATH we consider; their treatment is analogous. Moreover, we write `key` instead of `semantic-attribution` for brevity.

$n$ -ary function application (rather than binary), arbitrary binding symbols (rather than just  $\lambda$ -abstraction), and arbitrary attributions (rather than just simple types), but only assumes  $\alpha$ -conversion (rather than  $\alpha\beta\eta$  conversion).

### 3.1. Syntax

We start out by fixing an abstract syntax of “OM objects”, which we will relate to OPENMATH objects in Section 3.3. We will call the objects specified in Definition 4 “*abstract OM Objects*” when we want to distinguish from the “*standard OPENMATH objects*” defined in the OPENMATH2 standard [BCC<sup>+</sup>04, section 2].

**Definition 1 (Symbols and Variables).** In all of the following, we will assume the existence of two disjoint, countably infinite sets: a set *Symbols* of **symbols** and a set *Variables* of **variables**. Furthermore, we assume a set *Keys*  $\subseteq$  *Symbols* of **keys**.<sup>2</sup>

As usual in formal languages we are a little more careful about the symbols and variables we use in the construction of complex objects. The notions of vocabularies and contexts help us do this:

**Definition 2 (OM Vocabulary).** An OM **vocabulary** is a set of symbols. For every OM vocabulary  $T$ , we denote by  $Symbols(T) := Symbols \cap T$  the **set of symbols of  $T$**  and by  $Keys(T) := Keys \cap T$  the **set of keys of  $T$** .

**Definition 3 (OM Context).** An OM **context**  $C$  is an  $n$ -tuple of variables which we will write as  $\langle x_1, \dots, x_n \rangle$ . We will use  $+$  for tuple concatenation and  $\in$  for tuple membership.

**Definition 4 (OM Objects).** Let  $T$  be an OM vocabulary. The set  $O(T, C)$  of **OM objects** over  $T$  in context  $C$  is the smallest set closed under the following operations

1. if  $s \in Symbols(T) \setminus Keys(T)$ , then  $\mathbb{S}(s) \in O(T, C)$ ,
2. if  $x \in C$ , then  $\mathbb{V}(x) \in O(T, C)$ ,
3. if  $f, o_1, \dots, o_n \in O(T, C)$  for  $n > 0$ , then  $\mathbb{A}(f, o_1, \dots, o_n) \in O(T, C)$ ,
4. if  $b \in O(T, C)$ ,  $X_1, \dots, X_n \in AttVar(T, C)$  for  $n \geq 0$ , and  $o \in O(T, C')$  where  $C' = C + \langle varname(X_1), \dots, varname(X_n) \rangle$ , then  $\mathbb{B}(b, [X_1, \dots, X_n], o) \in O(T, C)$ ,
5. if  $o \in O(T, C)$ ,  $k \in Keys(T)$ , and  $v \in O(T, C)$ , then  $\mathbb{K}(o|k := v) \in O(T, C)$ .

Here **attributed variables** are defined by:  $o \in AttVar(T, C)$ , iff  $o = \mathbb{V}(x)$  for some  $x \in C$  or  $o = \mathbb{K}(o'|k := v)$  for  $o' \in AttVar(T, C)$ ,  $k \in Keys(T)$  and  $v \in O(T, C)$ . We call OM objects in the empty context **closed**. The name of an attributed variable is defined by  $varname(\mathbb{K}(o'|k := v)) = varname(o')$  and  $varname(\mathbb{V}(x)) = x$ .

Note that in contrast to the OPENMATH2 standard we only consider “unary” attributions that associate an object with a single key/value pair. This allows us to build the “flattening of attributions” into the abstract representation of

---

<sup>2</sup>This assumption is strictly for convenience in theory development; actually the determination of a symbol being a key is made by ascribing the role “semantic-attribution” in an OPENMATH content dictionary. When we align content dictionaries with vocabularies in Definition 14, we make sure that the CD roles are respected.

OM Objects. We can regain the syntactic structure of OPENMATH2 objects by introducing  $n$ -ary attributions as an abbreviation for nested attributions:  $\mathbb{K}(o|k_1 := v_1, \dots, k_n := v_n) = \mathbb{K}(\mathbb{K}(o|k_1 := v_1)|k_2 := v_2, \dots, k_n := v_n)$  for  $n \geq 2$ . With this trick<sup>3</sup> we have fully covered the requirement of “attribution flattening equivalence” required in the OPENMATH standard.

Case 4 of Def. 4 reveals an underspecification in the OPENMATH standard: The standard does not specify whether a bound variable may occur in the attributions of itself or of other variables bound by the same binder. Our definition requires  $X_i \in AttVar(T, C)$ , i.e., no variable may occur in any variable’s attribution. While this is a perfectly reasonable choice, others are possible. For example,  $X_i \in AttVar(T, C + \langle varname(X_1), \dots, varname(X_{i-1}) \rangle)$  permits every variable to occur in attributions of later variables. That permits merging nested binding objects all using the  $\Pi$ -binder of dependent type theory into a single binding object.  $X_i \in AttVar(T, C')$  permits variables to occur in each other’s attributions, which permits to represent mutually recursive let bindings.

In Case 4, one should also note that OPENMATH permits bindings with 0 bound variables. These degenerate to unary functions.

Let us fortify our intuition with an example which we will use throughout the paper; we focus on binding objects, since they are the most problematic case:

**Example 1.** The untyped universal quantification  $\forall x.x = x$  is represented as  $\mathbf{U} = \mathbb{B}(\mathbb{S}(\forall), [\mathbb{V}(x)], \boxed{x = x})$ <sup>4</sup>, where  $\forall$  is a symbol. To show the interaction of attribution and binding, we use a typed identity function represented as a  $\lambda$ -abstraction:  $\lambda x : \iota \rightarrow \iota. x$  is represented as  $\mathbf{L} = \mathbb{B}(\mathbb{S}(\lambda), [\mathbb{K}(\mathbb{V}(x))|\tau := \boxed{\iota \rightarrow \iota}], \mathbb{V}(x))$ . We have  $\mathbf{U} \in O(\{\forall, =\}, \langle \rangle)$  and  $\mathbf{L} \in O(T, \langle \rangle)$ , where  $T = \{\lambda, \tau, \iota, \rightarrow\}$  and  $Keys(T) = \{\tau\}$ .

The use of attributed variables in binders can lead to somewhat awkward notations when accessing the keys and attributions present in abstract binding objects. Therefore, we use the auxiliary definition of binding signatures in the technical developments below. Intuitively, an OM binding object has binding signature  $\sigma$  if it binds  $l(\sigma)$  variables where the  $i$ -th variable has  $d^i(\sigma)$  attributions.

**Definition 5 (Binding Signature).** A **binding signature**  $\sigma$  consists of

- a natural number  $l(\sigma)$  (the **length** of  $\sigma$ ),
- natural numbers  $d^1(\sigma), \dots, d^{l(\sigma)}(\sigma)$  (the **depth of  $\sigma$  at  $i$** ).

We denote by  $\bar{\sigma}$  the set of pairs  $\langle i, j \rangle \in \mathbb{N} \times \mathbb{N}$  where  $1 \leq i \leq l(\sigma)$  and  $1 \leq j \leq d^i(\sigma)$ .

**Definition 6 (Abbreviated Binding Notation).** If  $\sigma$  is a binding signature with length  $n$ ,  $b \in O(T, C)$ , and  $K : \bar{\sigma} \rightarrow Keys(T)$  and  $V : \bar{\sigma} \rightarrow O(T, C)$ , as well as

---

<sup>3</sup>In fact we propose to follow this path in the next version of the OPENMATH standard as it simplifies the presentation. Note that we are only talking about (standard) OPENMATH objects, not their XML or binary encodings, where  $n$ -ary attributions make sense for notational convenience.

<sup>4</sup>Here and throughout the paper we will use boxed mathematical formulae to gloss OPENMATH objects (encoded, abstract, or standard); we assume that this distinction is either meaningless or clear from the context. Here,  $\boxed{x = x}$  stands for  $\mathbb{A}(\mathbb{S}(=), \mathbb{V}(x), \mathbb{V}(x))$ .

$o \in O(T, C + \langle x_1, \dots, x_n \rangle)$ , then we write

$$\mathbb{B}(b[x_1, \dots, x_n|K := V].o) \quad \text{for} \quad \mathbb{B}(b, [X_1, \dots, X_n], o) \in O(T, C)$$

where  $X_i = \mathbb{K}(\mathbb{V}(x_i)|K(i, 1) := V(i, 1), \dots, K(i, d^i(\sigma)) := V(i, d^i(\sigma)))$ .

**Example 2 (Continuing Example 1).** In the abbreviated syntax  $\forall x.x = x$  is represented as  $\mathbf{U} := \mathbb{B}(\mathbb{S}(\forall)[x|\emptyset := \emptyset] \boxed{x = x})$  and  $\lambda x : \iota \rightarrow \iota.x$  as  $\mathbf{L} := \mathbb{B}(\mathbb{S}(\lambda)[x|K := V].\mathbb{V}(x))$ , where in the latter case

- $l(\sigma) = 1$  and  $d^1(\sigma) = 1$ , and therefore  $\bar{\sigma} = \{\langle 1, 1 \rangle\}$
- $K = \{\langle 1, 1 \rangle \mapsto \tau\}$  and  $V = \{\langle 1, 1 \rangle \mapsto \boxed{\iota \rightarrow \iota}\}$

Clearly, every OM object of the form  $\mathbb{B}(b, [X_1, \dots, X_n], o)$  can be written uniquely as an expression of the form  $\mathbb{B}(b[x_1, \dots, x_n|K := V].o)$ , and we will use the latter notation in the future and abbreviate  $\mathbb{B}(b[x_1, \dots, x_n|\emptyset := \emptyset].o)$  with  $\mathbb{B}(b[x_1, \dots, x_n].o)$ .

**Definition 7 (Substitution).** For  $o \in O(T, \langle x_1, \dots, x_n \rangle)$ , we denote by  $Subs(o)$  the function that maps  $\langle o_1, \dots, o_n \rangle$  to the object arising from  $o$  by substituting all free occurrences of  $x_i$  with  $o_i$  and renaming captured variables as usual.

Because the definition of substitution application is straightforward and well-known, we omit it, and only mention one technical detail regarding the shadowing of bound variables: In the degenerate case of a binding  $\mathbb{B}(b[x_1, \dots, x_n|K := V].o)$  with  $x_i = x_j$  for some  $i < j$ , the OPENMATH standard defines that  $x_i$  is shadowed by  $x_j$ , i.e., free occurrences of  $x_i = x_j$  in  $o$  refer to  $x_j$ .

**Definition 8 ( $\alpha$ -Equality).** Two objects are said to be  $\alpha$ -equal iff they arise from one another by renaming bound variables.  $\equiv_\alpha$  denotes the induced equivalence relation, and  $[o]_\alpha$  denotes the equivalence class of  $o$ .

Finally, we define the head of an OM object as follows:

**Definition 9 (Head).** The **head** of an OM object  $o$  is

- $o$  if  $o$  is a symbol or variable,
- $f$  if  $o = \mathbb{A}(f, o_1, \dots, o_n)$ ,
- $b$  if  $o = \mathbb{B}(b, [X_1, \dots, X_n], o')$ ,
- $k$  if  $o = \mathbb{K}(o'|k := v)$ .

### 3.2. Semantics

We will now interpret OM objects in algebras and define the free OM algebra, in which the algebraic meaning of an abstract OPENMATH object is given by its  $\alpha$ -equivalence class.

In the following, we will use the notation  $\Lambda x \in A.f(x)$  for the set-theoretical function defined by  $\{\langle x, f(x) \rangle : x \in A\}$ .  $A$  may be omitted if it is clear from the context. We also write  $B^A$  for the set of functions from  $A$  to  $B$ .

**Definition 10 (OM Algebra).** Let  $T$  be an OM vocabulary. An **OM algebra**  $A$  over  $T$  consists of

1. a set  $U := U^A$  called the **universe of discourse**
2. a family of sets  $R_n^A \subseteq U^{(U^n)}$  for  $n \geq 1$ ; we also define  $R_0^A = U$ ,
3. an element  $s^A \in U$  for every  $s \in \text{Symbols}(T) \setminus \text{Keys}(T)$ ,
4. a family of mappings  $\@_n^A : U \times U^n \rightarrow U$  for  $n \geq 1$ ,
5. a family of mappings  $\beta_K^A : U \times U^{\bar{\sigma}} \times R_{l(\sigma)}^A \rightarrow U$  for every binding signature  $\sigma$  and mapping  $K : \bar{\sigma} \rightarrow \text{Keys}(T)$ ,
6. a family of mappings  $\alpha_k^A : U \times U \rightarrow U$  for every  $k \in \text{Keys}(T)$ .

$s^A$  interprets the symbols intra-universally, and  $\@^A$ ,  $\beta^A$ , and  $\alpha^A$  are extra-universal operations that yield a construction-oriented interpretation function. The sets  $R_n^A$  are special. Because OPENMATH permits arbitrary expressions as binders, it is not possible to define the interpretation of every binder separately as is common in both first-order and higher-order settings. Instead, we need to model variable binding explicitly in the semantics. Syntactically, binders are operators that take terms with free variables as arguments. It is well-understood in higher-order logic and type theory that terms with  $n$  free variables can be modeled as  $n$ -ary functions on the universe. Thus, we interpret binders as operators taking functions as arguments. These come from the  $R_{l(\sigma)}^A$  in the third argument of  $\beta$  operator (note that  $l(\sigma) = n$  here). The functions from  $\bar{\sigma}$  to  $U^A$  in the second argument are used for dealing with the keys of the attributed variables.

Since we can always write a binder like  $\mathbb{B}(b[x].\mathbb{V}(x))$ , the set  $R_1^A$  should at least contain the identity function. However, putting  $R_n = U^{(U^n)}$  in general would preclude free algebras in which  $R_n$  contains only those functions that arise from the interpretation of terms with  $n$  free variables. Since the sets of those functions depend on  $A$  itself, we permit an arbitrary set  $R_n^A$  here and leave it to Def. 12 to sort out when an OM algebra is well-defined.

**Definition 11 (Assignment).** Let  $A$  be an OM Algebra over  $T$ , and let  $C$  be an OM context with  $n$  variables. An  $A$ -**assignment**  $\varphi$  for  $C$  is a tuple in  $(U^A)^n$ . We denote the assignment  $\langle \varphi_1, \dots, \varphi_n, u \rangle$  for  $C + \langle x \rangle$  by  $\varphi, u$ .

**Definition 12 (Interpretation).** Let  $A$  be an OM Algebra over  $T$ , and let  $\varphi$  be an  $A$ -assignment for a context  $C$ . The **interpretation**  $\llbracket o \rrbracket_\varphi^A$  of  $o \in O(T, C)$  in  $A$  under  $\varphi$  is defined as follows:

1.  $\llbracket \mathbb{S}(s) \rrbracket_\varphi^A = s^A$ ,
2.  $\llbracket \mathbb{V}(x_i) \rrbracket_\varphi^A = \varphi_i$ ,
3.  $\llbracket \mathbb{A}(f, o_1, \dots, o_n) \rrbracket_\varphi^A = \@_n^A(\llbracket f \rrbracket_\varphi^A, \langle \llbracket o_1 \rrbracket_\varphi^A, \dots, \llbracket o_n \rrbracket_\varphi^A \rangle)$ ,
4.  $\llbracket \mathbb{B}(b[x_1, \dots, x_n]K := V).o \rrbracket_\varphi^A = \beta_K^A(\llbracket b \rrbracket_\varphi^A, \mathcal{V}, \mathcal{F})$  where
  - (a)  $\sigma$  is the binding signature of the binding (which must have length  $n$ ),
  - (b)  $\mathcal{V} = \Lambda p \in \bar{\sigma}. \llbracket V(p) \rrbracket_\varphi^A$ ,
  - (c)  $\mathcal{F} = \Lambda u \in (U^A)^n. \llbracket o \rrbracket_{\varphi, u_1, \dots, u_n}^A$
5.  $\llbracket \mathbb{K}(o|k := v) \rrbracket_\varphi^A = \alpha_k^A(\llbracket o \rrbracket_\varphi^A, \llbracket v \rrbracket_\varphi^A)$ .

Whether the case for bindings is well-defined, depends on the sets  $R_n^A$ . We call  $A$  **well-defined** if  $\Lambda u \in (U^A)^n . \llbracket o \rrbracket_{\varphi, u_1, \dots, u_n}^A \in R_n^A$  for all  $C, n, o \in O(T, C)$ , and all assignments  $\varphi$  for  $C$ .

**Example 3 (Continuing Example 2).** To interpret  $\mathbf{U}$  we use an OM Algebra  $A$  with

1.  $U^A := \mathbb{N} \cup \{\mathbf{q}, \mathbf{e}, \mathbf{t}, \mathbf{f}, \perp\}$
2.  $R_n^A = U^{(U^n)}$ ,
3.  $\forall^A := \mathbf{q}$  and  $=^A := \mathbf{e}$ ,
4.  $\text{@}_2^A(\mathbf{e}, u, v) = \mathbf{t}$  if  $u = v$ ;  $\text{@}_2^A(\mathbf{e}, u, v) = \mathbf{f}$  if  $u \neq v$ ; and  $\text{@}_n^A(u, \langle u_1, \dots, u_n \rangle) = \perp$  otherwise.
5.  $\beta_\emptyset^A(\mathbf{q}, \emptyset, \mathcal{F}) = \mathbf{t}$  if  $\mathcal{F}(u) = \mathbf{t}$  for all  $u \in \mathbb{N}$ ;  $\beta_\emptyset^A(\mathbf{q}, \emptyset, \mathcal{F}) = \mathbf{f}$  if  $\mathcal{F}(u) = \mathbf{f}$  for some  $u \in \mathbb{N}$ ; and  $\beta_K^A(u, \langle x_1, \dots, x_n \rangle, \mathcal{F}) = \perp$  otherwise.

Note that we only specify the parts of the algebra we actually need for our example, all others can be picked arbitrarily. If we want to evaluate  $\llbracket \forall x. x = x \rrbracket$  in  $A$ , recall that  $\bar{\sigma} = \emptyset$  and thus  $\mathcal{V} = \Lambda p \in \bar{\sigma}. \llbracket \emptyset(p) \rrbracket_\emptyset^A = \emptyset$ , so we have

$$\llbracket \mathbf{U} \rrbracket_\emptyset^A = \llbracket \mathbb{B}(\mathbb{S}(\forall) [x]. \llbracket x = x \rrbracket) \rrbracket_\emptyset^A = \beta_\emptyset^A(\mathbf{q}, \emptyset, \mathcal{F})$$

where  $\mathcal{F} = \Lambda u \in U^A. \llbracket x = x \rrbracket_{(u)}^A$ . So  $\llbracket \mathbf{U} \rrbracket_\emptyset^A = \mathbf{t}$ , iff  $\mathcal{F}(u) = \mathbf{t}$  for all  $u \in \mathbb{N}$ . But observe that we have  $\mathcal{F}(u) = \llbracket \mathbb{A}(=, \mathbb{V}(x), \mathbb{V}(x)) \rrbracket_{(u)}^A = \text{@}_2^A(\mathbf{e}, \langle u, u \rangle) = \mathbf{t}$  by definition, and thus  $\llbracket \mathbf{U} \rrbracket_\emptyset^A = \mathbf{t}$  as expected.

Extending  $A$  to an interpretation of the  $\lambda$ -binder is more complicated because we have to commit to a type theory.

**Example 4 (Continuing Example 2).** We extend  $U^A$  so that it contains all function sets that can be formed from the natural numbers, i.e.,  $\mathbb{N}^\mathbb{N}$   $\mathbb{N}^{(\mathbb{N}^\mathbb{N})}$ ,  $(\mathbb{N}^\mathbb{N})^\mathbb{N}$  and so on, as well as the functions they contain. We call this set  $\mathbb{N}^{**}$ . For this to be useful, we should also extend our vocabulary with symbols  $\iota$  and  $\rightarrow$ . We put

1.  $U := \mathbb{N}^{**} \cup \{\mathbf{l}, \mathbf{p}, \perp\}$
2.  $R_n^A = U^{(U^n)}$ ,
3.  $\lambda^A = \mathbf{l}$ ,  $\iota^A = \mathbb{N}$ , and  $\rightarrow^A = \mathbf{p}$ , and
4. interpret  $\text{@}_2^A(\mathbf{p}, \langle u, v \rangle)$  as the set of functions from  $v$  to  $u$  if  $u$  and  $v$  are sets and as  $\perp$  otherwise. Furthermore, we put  $\text{@}_1^A(f, \langle u \rangle) = f(u)$  whenever function application is defined. We put  $\text{@}^A(f, \langle u_1, \dots, u_n \rangle) = \perp$  otherwise.
5. Then for  $\bar{\sigma} = \{\langle 1, 1 \rangle\}$ ,  $K = \{\langle 1, 1 \rangle \mapsto \tau\}$ , we can put  $\beta_K^A(\mathbf{l}, \mathcal{V}, \mathcal{F})$  to be the function  $\Lambda u \in \mathcal{V}(\langle 1, 1 \rangle). \mathcal{F}(u)$ . We put  $\beta_L^A(u, \mathcal{V}, \mathcal{F}) = \perp$  in all other cases.
6.  $\alpha_\tau^A(u, v) = u$ .

Then we can interpret  $\llbracket \lambda x : \iota \rightarrow \iota. x \rrbracket$  as follows. We have  $\llbracket \mathbf{L} \rrbracket_\emptyset^A = \llbracket \mathbb{B}(\mathbb{S}(\lambda) [x] K := V. \mathbb{V}(x)) \rrbracket_\emptyset^A = \beta_K^A(\mathbf{l}, \mathcal{V}, \mathcal{F})$  where

- $\mathcal{V} = \Lambda p \in \{\langle 1, 1 \rangle\}. \llbracket V(p) \rrbracket_\emptyset^A = \Lambda p \in \{\langle 1, 1 \rangle\}. \llbracket \iota \rightarrow \iota \rrbracket_\emptyset^A = \{\langle 1, 1 \rangle \mapsto \mathbb{N}^\mathbb{N}\}$ ,
- $\mathcal{F} = \Lambda u \in U. \llbracket \mathbb{V}(x) \rrbracket_{(u)}^A = \Lambda u \in U. u$

And thus, we evaluate  $\beta_K^A(\mathbf{l}, \mathcal{V}, \mathcal{F})$  as the identity function on  $\mathbb{N}^\mathbb{N}$  as expected.

A simple induction over the construction of OPENMATH objects in Definition 4, using the respective clauses in Definition 12, gives us an OPENMATH version of the well-known substitution lemma:

**Lemma 1 (Substitution Value Lemma).** *If  $o \in O(T, C + \langle x \rangle)$  and  $o' \in O(T, C)$ , then  $\llbracket [x/o']o \rrbracket_{\varphi}^A = \llbracket o \rrbracket_{\varphi, \llbracket o' \rrbracket_{\varphi}^A}^A$*

This in turn can be specialized in the usual way to obtain:

**Corollary 1 (Soundness of  $\alpha$ -Equality).** *If  $o \equiv_{\alpha} o'$  then  $\llbracket o \rrbracket_{\varphi}^A = \llbracket o' \rrbracket_{\varphi}^A$ .*

So we have shown that OM algebras form a model class for OPENMATH objects. We will now show that they characterize them up to isomorphism. For that we need to consider initial models, which will function as canonical representatives in this model class.

**Definition 13 (Free OM Algebra).** Let  $T$  be an OM vocabulary. Then the **free OM algebra**  $I := I(T)$  over  $T$  is defined as follows.

1.  $U^I = O(T, \emptyset) / \equiv_{\alpha}$ , i.e. the quotient set of the closed OPENMATH objects modulo  $\alpha$ -conversion.
2.  $R_n^I$  is the set of functions  $\overline{Subs(o)}$  for  $o \in O(T, \langle x_1, \dots, x_n \rangle)$ , which are defined as follows:  $\overline{Subs(o)}(\langle [o_1]_{\alpha}, \dots, [o_n]_{\alpha} \rangle) = [Subs(o)\langle o_1, \dots, o_n \rangle]_{\alpha}$ .
3.  $s^I = [\mathbb{S}(s)]_{\alpha}$ ,
4.  $\mathbb{A}_n^I([f]_{\alpha}, \langle [o_1]_{\alpha}, \dots, [o_n]_{\alpha} \rangle) = [\mathbb{A}(f, o_1, \dots, o_n)]_{\alpha}$ ,
5. for a binding signature  $\sigma$ :  $\beta_K^I([b]_{\alpha}, \mathcal{V}, \mathcal{F}) = [\mathbb{B}(b[x_1, \dots, x_n | K := V].o)]_{\alpha}$  where
  - $V = \Lambda p \in \bar{\sigma}.v_p$  for some  $v_p \in \mathcal{V}(p)$ ,
  - $o \in O(T, \langle x_1, \dots, x_n \rangle)$  is some object such that  $\overline{Subs(o)} = \mathcal{F}$ .
6.  $\alpha_k^I([o]_{\alpha}, [v]_{\alpha}) = [\mathbb{K}(o|k := v)]_{\alpha}$ .

Technically,  $I(T)$  is the free OM algebra over the empty set of generators. It is straightforward to generalize Def. 13 to arbitrary sets of generators.

**Lemma 2.**  *$I(T)$  is well-defined.*

*Proof.* We need to show several well-definedness conditions.

$R_n^A$ :  $\overline{Subs(o)}$  is well-defined because substituting closed  $\alpha$ -equivalent objects for the free variables of  $o$  yields  $\alpha$ -equivalent objects.

$\mathbb{A}_n^I$ : If  $f \equiv_{\alpha} f'$ ,  $o_1 \equiv_{\alpha} o'_1, \dots, o_n \equiv_{\alpha} o'_n$ , then  $\mathbb{A}(f, o_1, \dots, o_n) \equiv_{\alpha} \mathbb{A}(f', o'_1, \dots, o'_n)$ . This follows directly from the definition of  $\alpha$ -equivalence.

$\beta_K^I$ : If  $b \equiv_{\alpha} b'$ ,  $v_p \equiv_{\alpha} v'_p$  for all  $p \in \bar{\sigma}$ , then there exists an  $o \in O(T, \langle x_1, \dots, x_n \rangle)$  such that  $\overline{Subs(o)} = \mathcal{F}$ , and for two such  $o, o'$  we have that

$$\mathbb{B}(b[x_1, \dots, x_n | K := \Lambda p.v_p].o) \equiv_{\alpha} \mathbb{B}(b'[x_1, \dots, x_n | K := \Lambda p.v'_p].o').$$

The existence follows from the definition of  $R_n^I$ . The  $\alpha$ -equivalence holds because non- $\alpha$ -equivalent objects induce non- $\alpha$ -equivalent substitution functions.

$\alpha_k^I$ : If  $o \equiv_\alpha o'$  and  $v \equiv_\alpha v'$ , then  $\mathbb{K}(o|k := v) \equiv_\alpha \mathbb{K}(o'|k := v')$ . This follows directly from the definition of  $\alpha$ -equivalence.

□

**Lemma 3.** Let  $T$  be an OM vocabulary. Then  $\llbracket o \rrbracket^{I(T)} = [o]_\alpha$  for every  $o \in O(T, \langle \rangle)$ .

*Proof.* This is proved by a straightforward induction on the structure of  $o$ . □

**Lemma 4 ( $I(T)$  is initial).** Let  $A$  be an OM algebra over  $T$ , and let  $I := I(T)$ . Then there is a unique mapping  $h : U^I \rightarrow U^A$  satisfying  $h(\llbracket o \rrbracket^I) = \llbracket o \rrbracket^A$  for all  $o \in O(T, \langle \rangle)$ .

*Proof.*  $h$  maps  $[o]_\alpha \in U^I$  to  $\llbracket o \rrbracket^A$ . The needed property follows directly from Cor. 1 and Lem. 3. □

**Corollary 2 (Completeness of  $\alpha$ -Equality).** If  $o, o' \in O(T, \langle \rangle)$  and  $\llbracket o \rrbracket^A = \llbracket o' \rrbracket^A$  for all OM algebras  $A$ , then  $o \equiv_\alpha o'$ .

*Proof.* This follows from Lem. 3 by putting  $A := I(T)$ . □

In the classification of denotational semantics from Section 2.1 our semantics from Def. 10 is construction-oriented using the four functions  $s^A$ ,  $@_n^A$ ,  $\beta_K^A$ , and  $\alpha_k^A$  for the four constructions: all of them come with the OM algebra, not the model class. As we have predicted in Section 2.1 the semantics is rather involved. In particular, giving individual OM algebras is ludicrously complicated due to the functions  $\beta_K^A$ : These must provide an interpretation for *any* object used as a binder, binding *any* number of bound variables, in which each variable carries *any* list of attribution keys, each attributing *any* value. This convoluted semantics is unavoidable and enforced by the generality of the OPENMATH standard; for a more elegant semantics of a slightly restricted subset of OPENMATH objects, see Section 5.

### 3.3. OpenMath Objects with Uninterpreted Symbols

The semantics discussed so far was based on the abstract notion of OM Vocabularies. To arrive at a semantics of OPENMATH objects we need to relate this to OPENMATH CDs.

The OPENMATH2 standard introduces “abstract content dictionaries” to abstract from the concrete XML encoding of content dictionaries. According to [BCC<sup>+</sup>04, section 4.2], (abstract) CDs have a **CD name**, a **CD base URI**, and contain **symbol definitions**, which in turn consist (among others) of a **symbol name**, an optional **symbol role** (one of “binder”, “attribution”, “semantic-attribution”, “application”, “constant”, and “error”), and a set of **mathematical properties**.

**Definition 14 (OpenMath Symbols).** We say that a CD  $C$  **declares** an OPENMATH symbol  $\langle n, c, u, r \rangle$ , iff the CD base of  $C$  is  $u$ , the CD name of  $C$  is  $c$ , and  $C$  has a symbol definition with symbol name  $n$  and symbol role  $r$  (note that the role can be undefined as it is optional). We define the set *Symbols* to be the set of symbols

declared by some OPENMATH CD and the set *Keys* to be those with symbol role “semantic-attribution”.

There are three differences between abstract OM Objects and standard OPENMATH objects; all three are related to symbols and keys:

1. We do not take keys to be abstract OM objects by themselves (see clause 1 in Definition 4). We claim that there are no mathematically meaningful situations where keys can appear except in attributions. This design decision should not be perceived as a serious impediment for our semantics, since keys can be added analogously to the treatment below at the cost of adding an additional case everywhere.
2. The OPENMATH2 [BCC<sup>+</sup>04] “role system”, poses some additional restrictions on where symbols can occur, but not enough to simplify our construction of binding signatures. Therefore, we disregard it here and refer the reader to [RK09] for details and an extended role system proposal that would.
3. We do not consider attributions with symbols that are not in *Keys*, in particular symbols with roles “attribution” which are intended by the OPENMATH2 standard for just this purpose. However the standard states

*This form of attribution may be ignored by an application, so should  
be used for information which does not change the meaning of the  
attributed OpenMath object.* [BCC<sup>+</sup>04, clause 2.1.4.i]

and therefore it is necessary to disregard these attributions in the construction of a semantics for OPENMATH. In the mapping from standard OPENMATH objects to abstract ones, we strip attributions with non-*Keys* symbols.

This allows us to define the meaning of an OPENMATH object. As we are not taking mathematical properties in CDs into account, we will think of these symbols as uninterpreted, therefore we will call it the “algebraic meaning”.

**Definition 15 (Algebraic Meaning).** Let  $o$  be an OPENMATH object, then we call the set of symbols  $s$  such that  $\mathbb{S}(s)$  occurs in  $o$  the **OM vocabulary induced by  $o$** .

If  $o$  is a closed OM Object,  $T$  its induced vocabulary, and  $A$  an OM algebra over  $T$ , then the **algebraic meaning of  $o$  in  $A$**  is  $\llbracket o \rrbracket^A$  and the **algebraic meaning of  $o$**  is  $\llbracket o \rrbracket^{I(T)}$ .

Note that the algebraic meaning of an abstract OPENMATH object is just an  $\alpha$ -equivalence class of (standard) OPENMATH objects.

As discussed in the introduction, the algebraic semantics only gives us a rather weak and syntactic concept of meaning of the OPENMATH language. To understand the full meaning of OPENMATH objects we need to take CDs into account, which we do in the next section.

## 4. OpenMath Models

If we want to understand mathematical properties in OPENMATH content dictionaries, we need to have a notion of “truth” — after all, the properties are assumed

to hold true. Furthermore, we need to take into account the mathematical properties themselves.

#### 4.1. Theories and Satisfaction

As formal mathematical properties are expressed as OPENMATH objects, we will need to build the required notions of truth, its opposite falsity, and equality into an OM vocabulary. This is rather simple.

**Definition 16 (OM Logic).** An OM vocabulary  $L$  with distinguished symbols  $\top$ ,  $\perp$ , and  $=$  is called an **OM logic**.

In a logic, we can define negation as follows:

**Definition 17 (OM Logic).** Given a logic  $L$ , we write  $\neg o$  for  $\mathbb{A}(=, o, \perp)$ .

In OPENMATH CDs, (formal) mathematical properties are expressed as statements in some foundational logical system, thus the OM Objects representing them will in general contain symbols from the foundation and the CD itself. For instance, the `arith1` CD [CDa04] contains an FMP with the object  $\boxed{\forall a, b. a + b = b + a}$  to express commutativity of addition. The symbols  $\boxed{\forall}$  and  $\boxed{=}$  are from the vocabulary of the foundational system and the symbol  $\boxed{+}$  is from the CD itself.

We will treat OPENMATH content dictionaries as logical theories, which are determined by their vocabularies and axioms, and model them using institutions (see [Rab08] for an introduction to both).

**Definition 18 (Theory).** Let  $L$  be an OM logic and  $T$  an OM vocabulary. An **OM theory**  $\Theta$  for  $L$  is a pair  $\langle T, Ax \rangle$  where  $Ax \subseteq O(L \cup T, \langle \rangle)$ .<sup>5</sup> We will denote  $Ax$  with  $Axioms(\Theta)$  and use  $O(\Theta, C) := O(L \cup T, C)$  and take an OM algebra over  $\Theta$  to be an OM algebra over  $L \cup T$ .

Note that  $\langle \emptyset, \emptyset \rangle$  is a theory for any OM logic  $L$ , we call  $\langle \emptyset, \emptyset \rangle$  the **empty theory over  $L$** .

In this setting we can define OM models as those algebras that respect equality and in which the axioms hold.

**Definition 19 (Model).** Let  $L$  be an OM logic and  $\Theta$  be an OM theory for  $L$ . An OM algebra  $M$  over  $\Theta$  is a **model** of  $\Theta$  if

1.  $\llbracket \top \rrbracket^M \neq \llbracket \perp \rrbracket^M$ ,
2. for all  $C$ ,  $o, o' \in O(T, C)$ , and  $\varphi$ , we have that  $\llbracket \mathbb{A}(=, o, o') \rrbracket_\varphi^M = \llbracket \top \rrbracket^M$  iff  $\llbracket o \rrbracket_\varphi^M = \llbracket o' \rrbracket_\varphi^M$ ,
3. for all  $A \in Axioms(\Theta)$ , we have that  $\llbracket A \rrbracket^M = \llbracket \top \rrbracket^M$ .

The **Model Class  $\mathcal{M}(C)$  of  $\Theta$**  is the set of OM Models of  $\Theta$ .

Def. 19 gives us the OM versions of the standard notions of satisfaction and semantic entailment.

---

<sup>5</sup>There are content dictionaries with formal mathematical properties that are not closed. We follow common mathematical practice and assume they are implicitly closed universally.

**Definition 20 (Satisfaction).** Let  $L$  be an OM logic,  $\Theta$  be an OM theory for  $L$ ,  $o \in O(\Theta, C)$ ,  $M$  an OM model of  $\Theta$ , and  $\varphi$  an assignment for  $C$  into  $M$ . Then we say that  $M$  **satisfies**  $o$  **under**  $\varphi$  (which we denote as  $M, \varphi \models o$ ), iff  $\llbracket o \rrbracket_{\varphi}^M = \llbracket \top \rrbracket_{\varphi}^M$ . We write  $M \models o$  if  $M, \varphi \models o$  holds for all assignments  $\varphi$  and say that  $o$  is *valid* in  $M$ .

The first condition in Def. 19 guarantees that the universe of a model must have at least two elements. This excludes inconsistent models in which the truth values collapse and every formula is true. More precisely, we have:

**Lemma 5.** In the situation of Def. 20, if  $M, \varphi \models \neg o$ , then  $M, \varphi \not\models o$ . In particular,  $M \not\models \perp$ .

*Proof.* This follows immediately from Def. 19 and Def. 20.  $\square$

**Definition 21 (Entailment).** Let  $\Theta$  be an OM theory and  $o$  a closed object. Then we say that  $\Theta$  **entails**  $o$  ( $\Theta \models o$ ), iff  $M \models o$  for all  $M \in \mathcal{M}(\Theta)$ .

**Example 5 (Continuing Example 3).** We can extend the vocabulary  $\{\forall, =\}$  into an OM logic  $Q = \{\forall, =, \top\}$ . Then the OM algebra  $A$  from 3 becomes a model for the empty theory over  $Q$  by putting  $\top^A := \text{t}$ . Note that  $\mathbf{U}$  is entailed by the empty theory over  $Q$ .

We will now turn to the initial semantics again, this time to build initial OM models.

**Definition 22 (Congruence Relation).** Let  $T$  be an OM vocabulary and  $A$  an OM algebra over  $T$ . A **congruence relation** on  $A$  is a family of equivalence relations on  $U^A$  and  $R_n^A$  all denoted by  $\equiv$  such that (whenever applicable)<sup>6</sup>

1. if  $u \equiv u'$  and  $u_i \equiv u'_i$  for  $i = 1, \dots, n$ , then

$$\text{@}_n^A(u, \langle u_1, \dots, u_n \rangle) \equiv \text{@}_n^A(u', \langle u'_1, \dots, u'_n \rangle),$$

2. for  $l(\sigma) = n$ , if  $u \equiv u'$ ,  $\mathcal{V}(p) \equiv \mathcal{V}'(p)$  for all  $p \in \bar{\sigma}$ , and  $\mathcal{F} \equiv \mathcal{F}'$ , then

$$\beta_K^A(u, \mathcal{V}, \mathcal{F}) \equiv \beta_K^A(u', \mathcal{V}', \mathcal{F}'),$$

3. if  $u \equiv u'$  and  $v \equiv v'$ , then  $\alpha_k(u, v) \equiv \alpha_k(u', v')$ ,
4. if  $\mathcal{F} \equiv \mathcal{F}'$  and  $u_i \equiv u'_i$  for  $i = 1, \dots, n$ , then

$$\mathcal{F}(\langle u_1, \dots, u_n \rangle) \equiv \mathcal{F}'(\langle u'_1, \dots, u'_n \rangle).$$

**Definition 23 (Quotient Algebra).** Let  $T$  be an OM vocabulary,  $A$  an OM algebra over  $T$ , and  $\equiv$  a congruence relation on  $A$ . Then the OM algebra  $Q := A/\equiv$  over  $T$  is defined by:

1.  $U^Q = U^A/\equiv$ ,

---

<sup>6</sup>Note that we do not have to consider a congruence on keys because equations between keys are not well-formed objects.

2.  $R_n^Q$  is the set of all functions of the form

$$f : (U^Q)^n \rightarrow U^Q, f([u_1]_\equiv, \dots, [u_n]_\equiv) = [F(u_1, \dots, u_n)]_\equiv$$

for some  $F \in R_n^A$ ,

3.  $\@_n^Q$ ,  $\beta_K^Q$ , and  $\alpha_k^Q$  are induced by their analogues in  $A$ .

**Lemma 6.** *In the situation of Def. 23,*

- $Q$  is a well-defined OM algebra if  $A$  is,
- for all  $o \in O(T, C)$  and all  $A$ -assignments  $\varphi = (\varphi_1, \dots, \varphi_n)$ , it holds that  $[\llbracket o \rrbracket_\varphi^A]_\equiv = [\llbracket o \rrbracket_{\varphi'}^Q$  where  $\varphi'$  is the  $Q$ -assignment given by  $\varphi'_i = [\varphi_i]_\equiv$  for  $i = 1, \dots, n$ .

*Proof.* We prove the lemma by induction on  $o$  and its context  $C$ . The first part of the lemma is proved in the induction step for binders.

- $\mathbb{S}(s)$ : Trivial.
- $\mathbb{V}(x)$ : Immediately from the relation between  $\varphi$  and  $\varphi'$ .
- $\mathbb{A}(f, o_1, \dots, o_n)$ : Immediately from the definition of congruence.
- $\mathbb{K}(o|k := o')$ : Immediately from the definition of congruence.
- $\mathbb{B}(b[x_1, \dots, x_n|K := V].o)$ : If  $Q$  is well-defined, this follows immediately from the definition of congruence. To show well-definedness, we have to show that  $f = \Lambda(v_1, \dots, v_n) \in (U^Q)^n$ .  $\llbracket o \rrbracket_{\varphi', v_1, \dots, v_n}^Q \in R_n^Q$ . Due to the well-definedness of  $A$ , we know that  $F = \Lambda(u_1, \dots, u_n) \in (U^A)^n$ .  $\llbracket o \rrbracket_{\varphi, u_1, \dots, u_n}^A \in R_n^A$ . Thus, due to the definition of  $R_n^Q$ , we know that  $f' : ([u_1]_\equiv, \dots, [u_n]_\equiv) \mapsto [F(u_1, \dots, u_n)]_\equiv \in R_n^Q$ . The induction hypothesis for  $o$  shows that  $f = f'$ .

□

**Definition 24 (Induced Congruence).** Let  $\Theta = \langle T, Ax \rangle$  be an  $L$ -theory, then we define a congruence relation  $\equiv_\Theta$  on  $I(L \cup T)$  as follows:

$$[o]_\alpha \equiv_\Theta [o']_\alpha \quad \text{iff} \quad \Theta \models \mathbb{A}(=, o, o') \quad \text{for } o, o' \in O(L \cup T, \langle \rangle)$$

and

$$\overline{\text{Subs}(o)} \equiv_\Theta \overline{\text{Subs}(o')} \quad \text{iff} \quad \Theta \models \mathbb{A}(=, o, o') \quad \text{for } o, o' \in O(L \cup T, C).$$

We call  $\equiv_\Theta$  the **congruence induced by  $\Theta$** .

**Lemma 7.** *Let  $\Theta = \langle T, Ax \rangle$  be an  $L$ -theory, then  $\equiv_\Theta$  is indeed a congruence relation. Moreover, if  $\Theta \models o$  iff  $[o]_\alpha \equiv_\Theta [\top]_\alpha$ ; and  $\Theta \models \neg o$  iff  $[\neg o]_\alpha \equiv_\Theta [\perp]_\alpha$ .*

*Proof.* For the first part, we need to show the properties of a congruence relation. All cases are straightforward. We prove the case of application as an example.

Assume  $[o_i]_\alpha \equiv_\Theta [o'_i]_\alpha$  for  $i = 0, \dots, n$ . Then  $\Theta \models \mathbb{A}(=, o_i, o'_i)$ , and thus  $\llbracket o_i \rrbracket^A = \llbracket o'_i \rrbracket^A$  in every  $\Theta$ -model  $A$ . In that case, it follows that

$$\@_n^A(\llbracket o_0 \rrbracket^A, \langle \llbracket o_1 \rrbracket^A, \dots, \llbracket o_n \rrbracket^A \rangle) = @_n^A(\llbracket o'_0 \rrbracket^A, \langle \llbracket o'_1 \rrbracket^A, \dots, \llbracket o'_n \rrbracket^A \rangle);$$

and therefore,  $\Theta \models \mathbb{A}(=, \mathbb{A}(o_0, o_1, \dots, o_n), \mathbb{A}(o'_0, o'_1, \dots, o'_n))$ ; and therefore,

$$[\mathbb{A}(o_0, o_1, \dots, o_n)]_\alpha \equiv_\Theta [\mathbb{A}(o_0, o_1, \dots, o_n)]_\alpha.$$

Finally using Lem. 3 yields

$$@_n^{I(L \cup T)}([o]_\alpha, \langle [o_1]_\alpha, \dots, [o_n]_\alpha \rangle) \equiv_\Theta @_n^{I(L \cup T)}([o]_\alpha, \langle [o_1]_\alpha, \dots, [o_n]_\alpha \rangle)$$

as needed.

The second part follows after observing that for every  $M, \varphi$ , we have (by Def. 19 and 20)  $M, \varphi \models o$  iff  $M, \varphi \models \mathbb{A}(=, o, \top)$  as well as (by Def. 17)  $M, \varphi \models \neg o$  iff  $M, \varphi \models \mathbb{A}(=, o, \perp)$ .  $\square$

As a consequence, we can construct the initial model as follows:

**Definition 25 (Initial Model).** Let  $\Theta = \langle T, Ax \rangle$  be an  $L$ -theory.  $\Theta$  is called **semantically consistent** if  $\Theta$  has some model. In that case, we put  $I(\Theta) := I(L \cup T) / \equiv_\Theta$  and call it the **initial model for  $\Theta$** .

And that finally yields

**Theorem 1 (Completeness).** If  $\Theta$  is semantically consistent,  $I(\Theta)$  is indeed a  $\Theta$ -model, and for all  $o \in O(\Theta, \langle \rangle)$  we have  $I(\Theta) \models o$  iff  $\Theta \models o$ .

*Proof.* We know  $I(\Theta) \models o$  iff  $\llbracket o \rrbracket^{I(\Theta)} = \llbracket \top \rrbracket^{I(\Theta)}$ . Using Lem. 6, this is equivalent to  $\llbracket \llbracket o \rrbracket^{I(T)} \rrbracket_{\equiv_\Theta} = \llbracket \llbracket \top \rrbracket^{I(T)} \rrbracket_{\equiv_\Theta}$  where  $T$  is the vocabulary of  $\Theta$ . The latter is equivalent to  $\Theta \models \mathbb{A}(=, o, \top)$  by Lem. 3 and Def. 24. And that is equivalent to  $\Theta \models o$ .

To show that  $I(\Theta)$  is a model, we have to show the three properties of Def. 19. Regarding the first property, because there is some model  $M$ , we know  $\llbracket \top \rrbracket^M \neq \llbracket \perp \rrbracket^M$  and therefore  $\llbracket \top \rrbracket_\alpha \not\equiv_\Theta \llbracket \perp \rrbracket_\alpha$  and therefore  $\llbracket \top \rrbracket^{I(\Theta)} \neq \llbracket \perp \rrbracket^{I(\Theta)}$ . The second property holds because  $I(\Theta) \models \mathbb{A}(=, o, o')$  is equivalent to  $[o]_\alpha \equiv_\Theta [o']_\alpha$ , and that is equivalent to  $\llbracket o \rrbracket^{I(\Theta)} = \llbracket o' \rrbracket^{I(\Theta)}$ . The third property follows from the first part of this theorem.  $\square$

From this result we directly obtain the main theorem of this section as a corollary:

**Theorem 2 (Herbrand Theorem).** Every semantically consistent OM theory has a model that arises as a quotient of the free OM algebra.

#### 4.2. The Meaning of OpenMath CDs and Objects

Note that the definitions above are still abstract in the sense that they refer to OM vocabularies, and OM theories, and not OPENMATH CDs. So as in section 3.3 we have to relate abstract OM objects to standard ones and in particular to answer the question: what is the theory of an OPENMATH content dictionary? The OPENMATH2 standard leaves this information under-defined, so we propose an interpretation that allows us to define an adequate notion of mathematical semantics<sup>7</sup>.

Note that OPENMATH CDs need not be self-contained, i.e. their FMPs can contain symbols that are neither introduced in the CD nor from the foundational

---

<sup>7</sup>Arguably the OPENMATH standard cannot fix this fully, since it intends to support all mathematical software systems including such that are “semantics-independent” like mathematical editing systems.

system. Of course, these symbols (and thus the mathematical properties in CDs that introduce them) should have an effect on the meaning of the symbols described by the FMP, so they need to be taken into account; naturally this process must be iterated until a fixed point has been reached.

**Definition 26 (CD References).** Let  $C$  be an OPENMATH content dictionary, then we say that  $C$  **references**  $D$ , iff  $C \neq D$  and some FMP element in  $C$  contains a symbol with CD  $D$ . We call a CD **basic**, iff it does not reference other CDs.

OPENMATH used to explicitly annotate the “CDuses relation”, which is the transitive closure of the “references relation”, but has deprecated this in OPENMATH 2, since it can be automatically extracted in the way defined above. Note that OPENMATH does not make any assumptions about the absence of cycles in the references relation. In this respect, the references relation differs from the “imports relation” employed in module systems for mathematics (see [RK08, RK11] for an overview).

**Definition 27 (Signature and Property set of a CD).** The **signature** of a CD  $C$  is the set of symbols it declares in union with the signatures of all CDs referenced by  $C$ . Similarly, the **property set** of a CD  $C$  is the set of OPENMATH objects in FMP elements in  $C$  (these are called the **local properties** of  $C$ ) in union with all the property sets of all CDs referenced by  $C$ .

With this, we can directly define the OM theory induced by a CD.

**Definition 28 (Theory of a CD).** We call the pair  $\langle S, P \rangle$ , where  $S$  is the signature of  $C$  and  $P$  is the property set of  $C$  the **OM theory of  $C$** .

In essence, the OM theory of a content dictionary is the union of all symbol declarations and mathematical properties from all theories from which a symbol is used in the CD. There is no problem with the (implicit) references relation being cyclic, since all morphisms (in the terminology of [RK11]) are the identity and we are constructing the (iterated) union. Note furthermore that OPENMATH only supports literal CD names, and we can assume the set of CDs to be finite, therefore, the signature and axiom set of a CD are finite.

Definition 28 allows us to define the meaning of a CD as a class of OM models.

**Definition 29 (Model Class and Entailment for CDs).** Let  $C$  be an OPENMATH CD and  $\Theta$  the OM theory of  $C$ , then the **Model Class of  $C$**  is  $\mathcal{M}(\Theta)$  and  $C \models o$ , iff  $\Theta \models o$ .

## 5. A Symbol-Oriented Semantics for OpenMath

In this section, we will present a symbol-oriented semantics for a subset of OPENMATH expressions defined by a novel role/type system that strengthens the one specified in the OPENMATH 2 standard. We use a well-chosen trade-off in order to

simplify the definition of algebras significantly while preserving the simplicity and flexibility of OPENMATH.

Considering our construction-oriented semantics, we observe that from the four constructions of OPENMATH (symbols, application, binding, and attribution) two (symbols and attributions) can be easily rephrased as a symbol-oriented semantics: Then  $s^A$  and  $\alpha_k^A$  become the semantic entities assigned to a symbol  $s \in \text{Symbols}(T) \setminus \text{Keys}(T)$  or  $k \in \text{Keys}(T)$ , respectively. Note that in the case of attributions, this is only possible because the OPENMATH standard requires the head of an attribution to be a symbol.

For the heads of applications and bindings, on the other hand, OPENMATH permits arbitrary complex objects. Consequently, the semantics must account for any object of the universe acting as a function or binder. Therefore, a separation between extra-universal functions/binders and intra-universal arguments/scopes is not possible.

For OPENMATH applications, this cannot be remedied. Type systems have been used successfully for languages with function application. But for a general purpose mathematical language like OPENMATH, where it is undecidable whether an object denotes a function, sound type systems cannot be both complete and decidable.

The binding construction is an intermediate case. On the one hand, OPENMATH permits arbitrary bindings. On the other hand, this freedom is rarely exploited: almost every binders used in practice is a symbol or possibly an application of a symbol to some arguments, and they expect a specific list of attributions to the bound variables.

Therefore, our role system focuses on restricting the permitted binding objects. This leads to a symbol-oriented semantics with a single extra-universal application operator at a comparatively small price.

### 5.1. A Role System

We will now define a positive role system that provides a reasonable trade-off between being universal and being strict to prepare for a symbol-oriented semantics. Most characteristically, our role system generalizes the concept of functions with arities to binders.

The basic definitions are as follows:

**Definition 30 (Arities).** Consider a vocabulary  $T$ . An **application-arity** is a finite sequence of either  $_-$  or  $_+^+$ , the latter occurring at most once. A **binding-arity** is a finite sequence of either  $K$  or  $K^+$  for  $K \subseteq \text{Keys}(T)$ , the latter occurring at most once.

We will write sequences using the notation  $[e_1, \dots, e_n]$  and  $E@e$  for the sequence  $E$  extended with the element  $e$ .

**Definition 31 (Roles).** A **role** is either **obj**, **key**, or a pair  $(a, b)$  of an application-arity  $a$  and a binding-arity  $b$ . A **roled vocabulary**  $(T, r)$  is vocabulary  $T$  together with a function  $r$  assigning a role to each symbol.

Intuitively, objects with role **obj** are the “usual” objects, i.e., values (including all functions) in the universe of discourse. These exclude the keys, which have role **key**, and the binders, which have the complex roles  $(a, b)$ . In the latter roles,  $\_$  stands for an argument, and  $\_+$  for a finite non-empty sequence of arguments. Similarly,  $K$  stands for an attributed variable whose attribution keys are exactly the ones in  $K$ , and  $K^+$  stands for a finite non-empty sequence of such attributed variables. We use the following auxiliary definition to replace occurrences of  $\_+$  and  $K^+$  with the needed number of repetitions of  $\_$  and  $K$ , respectively:

**Definition 32.** Given an application or a binding-arity  $s$  of length  $m$ , and a natural number  $n \geq m$ , we define  $s^n$  as follows:

- if  $s$  does not contain  $\_+$  or  $K^+$ , then  $s^n = s$ ,
- if  $s$  contains  $\_+$  or  $K^+$ , respectively, then  $s^n$  arises from  $s$  by replacing  $\_+$  or  $K^+$  with  $n - m + 1$  repetitions of  $\_$  or  $K$ , respectively.

Then we can make the meaning of binding arities more precise by:

**Definition 33.** An attributed variable  $X$  **matches** the set of keys  $\{k_1, \dots, k_n\}$  if it is of the form

$$X = \mathbb{K}(x | k_1 := v_1, \dots, k_n := v_n)$$

for some ordering  $k_1, \dots, k_n$ . In that case, we define

$$X(k_i) = v_i.$$

A sequence  $X_1, \dots, X_n$  of attributed variables **matches** the binding-arity  $b$  if  $b^n = [b_1, \dots, b_n]$  and each  $X_i$  matches  $b_i$ .

**Example 6.** Binders binding a single variable with a **type** attribution can be declared with the role  $([], [\{\text{type}\}])$ . More complex binders arise by combining application-arity and binding-arity such as in  $([\_, \_], [\emptyset])$  for the definite integral  $\int_a^b$ , which takes two arguments and then binds one variable without attributions.

Then we can define the well-roled objects:

**Definition 34.** Assume a roled OM vocabulary  $(T, r)$ . An object  $o \in O(T, C)$  for some context  $C$ . We define the **well-roled** objects  $o$  and their roles  $R(o)$ .  $o$  is well-roled if one of the following holds:

1.  $o = \mathbb{S}(s)$ . In that case,  $R(o) = r(s)$ .
2.  $o = \mathbb{V}(x)$  for  $x \in C$ . In that case,  $R(o) = \text{obj}$ .
3.  $o = \mathbb{A}(f, o_1, \dots, o_n)$ ,  $R(o_1) = \dots = R(o_n) = \text{obj}$ , and
  - (a)  $R(f) = \text{obj}$ , in which case  $R(o) = \text{obj}$ , or
  - (b)  $R(f) = (a, b)$  and the length of  $a$  is  $n$  or  $a$  contains  $\_+$ , in which case  $R(o) = ([], b)$ .
4.  $o = \mathbb{B}(o_1, [X_1, \dots, X_n], o_2)$ ,  $R(o_1) = ([], b)$ ,  $X_1, \dots, X_n$  matches  $b$ , all attribution values in all  $X_i$  have role **obj**, and  $R(o_2) = \text{obj}$ .  
In that case,  $R(o) = \text{obj}$ .
5.  $o = \mathbb{K}(o' | k := v)$  and  $R(k) = \text{key}$ , and  $R(o') = R(v) = \text{obj}$ .  
In that case  $R(o) = \text{obj}$ .

**Limitations.** As expected, our role system is not quite universal. In the following, we discuss the limitations that our implicit or explicit assumptions impose when we restrict attention to well-roled objects. We assume that all unroled symbols are assigned the role `obj`.

Firstly, since our role system is positive, symbols with a role `key` or  $(a, b)$  may not occur in general OPENMATH objects. The former may only occur as the head of an attribution. The latter may only occur as the head of a binding (if  $a = []$ ) or as the head of an application that itself occurs as the head of a binding (if  $a \neq []$ ). Conversely, only such objects may occur as the heads of attributions and bindings.

Secondly, since the keys of an attributed variable in a binding form a set, bound variables may not have two attributions with the same key, and the order of attributions does not matter. Moreover, the attributions on a bound variable must match the role of the binder.

For practical purposes, this means that all symbols intended to be used as binders or keys must be assigned a role and may then not be used for anything else. Moreover, for each binder, an *a priori* commitment is needed what attributions its bound variables will carry.

The latter limitation is the only one that may cause concern: It is conceivable that the same binder should be used with different binding-arithies in different situations. For such purposes, a reasonable extension of our role system would be to permit optional keys, e.g., a binding-arity  $\{\text{type}?\}^+$  for arbitrarily many bound variables, which may carry a `type` attribution. Another possible generalization is to permit and then ignore attributions to a bound variable that are not required by the binder. To the best of our knowledge, no reasonable representation is excluded by these limitations.

## 5.2. A Roled Semantics

For roled vocabularies, we can simplify the definition of OM algebras significantly. In particular, we can make it symbol-oriented. We will still use a monolithic universe, but we will interpret both binders and keys extra-universally.

First, we need some auxiliary definitions:

**Definition 35.** Given a set  $U$ , we define

- For a set  $K = \{k_1, \dots, k_n\}$  of keys, we call the elements of the set  $U^K$  *K-records* over  $U$ .
- For an application-arity  $a$ , the set  $U^a$  is given by

$$\begin{aligned} U^{[-, \dots, -]} &= U \times \dots \times U \\ U^{[-, \dots, -, +, \dots, -]} &= U \times \dots \times U \times \bigcup_{i=1}^{\infty} U^i \times U \times \dots \times U \end{aligned}$$

- For a binding-arity  $b$ , the set  $U^b$  is given by

$$\begin{aligned} U^{[K_1, \dots, K_n]} &= U^{K_1} \times \dots \times U^{K_n} \\ U^{[K_1, \dots, K_m, K^+, L_1, \dots, L_n]} &= U^{K_1} \times \dots \times U^{K_m} \times \bigcup_{i=1}^{\infty} (U^K)^i \times U^{L_1} \times \dots \times U^{L_n} \end{aligned}$$

- Given an element  $r \in U^b$ , then  $|r|$  is the number of records in  $r$ .

These definitions are used to capture the semantics of binders with binding-arity  $b$ . If  $b = [K_1, \dots, K_n]$  does not contain an occurrence of  $K^+$ , the binder expects  $n$  bound variables. Then the set  $U^b$  contains tuples  $r = \langle r_1, \dots, r_n \rangle$  such that  $r_i$  is a  $K_i$ -record providing an attributed value  $r_i(k)$  for each key  $k \in K_i$ . In these cases, we always have  $|r| = n$ . If  $b$  contains  $K^+$ , then  $U^b$  contains tuples of arbitrary length corresponding to an arbitrary number of bound variables. Therefore, we use  $|r|$  to obtain the actual number of records in a tuple  $r$ . Finally, pairs  $(r, f)$  where  $r \in U^b$  and  $f : U^{|r|} \rightarrow U$  provide a record of attributions and a function  $f$  on  $U$  in as many arguments as there are attributed variables.

**Definition 36 (Applicative Structure).** An **applicative structure** is a pair  $(U, @)$  where  $U$  is a set and  $@$  is a family of mappings  $@_n : U \times U^n \rightarrow U$  for  $n \geq 1$ .

We still have to use the general application operator  $@$  from Def. 10. This is necessary because we have to permit arbitrary objects with functional behavior. But apart from that, we can finally give a symbol-oriented semantics.

Given a universe  $U$ , the general structure of the semantics is as follows:

| Syntactic role   | Semantic domain  |
|------------------|--|
| <code>obj</code> | $U$  |
| <code>key</code> | $U \times U \rightarrow U$   |
| $(a, b)$         | $U^a \rightarrow (\{(r, f) \mid r \in U^b, f : U^{ r } \rightarrow U\} \rightarrow U)$ |

In particular, binders and keys are interpreted in certain extra-universal domains.

**Definition 37 (Roled Algebra).** Let  $(T, r)$  be a roled OM vocabulary. A **roled OM algebra**  $A$  over  $(T, r)$  consists of

- an applicative structure  $(U, @)$ ,
- a family of sets  $R_n^A \subseteq U^{(U^n)}$  for  $n \geq 1$ ; we also define  $R_0^A = U$ ,
- for every  $s \in T$  an element  $s^A$  of the domain corresponding to  $r(s)$ .

**Definition 38 (Interpretation).** Let  $A$  be a roled OM Algebra over  $(T, r)$ , and let  $\varphi$  be an  $A$ -assignment for a context  $C$ . The **interpretation**  $\llbracket o \rrbracket_\varphi^A$  of a well-roled  $o \in O(T, C)$  in  $A$  under  $\varphi$  is an element of the domain corresponding to  $R(o)$ . It is defined by induction on  $o$  according to Def. 34:

- $\llbracket S(s) \rrbracket_\varphi^A = s^A$  (for any role  $r(s)$ ),
- $\llbracket V(x_i) \rrbracket_\varphi^A = \varphi_i$ ,
- (a) if  $R(f) = \text{obj}$ , then  $\llbracket A(f, o_1, \dots, o_n) \rrbracket_\varphi^A = @_n^A(\llbracket f \rrbracket_\varphi^A, \langle \llbracket o_1 \rrbracket_\varphi^A, \dots, \llbracket o_n \rrbracket_\varphi^A \rangle)$ ,  
(b) if  $R(f) = (a, b)$ , then  $\llbracket A(f, o_1, \dots, o_n) \rrbracket_\varphi^A = f^A(\langle \llbracket o_1 \rrbracket_\varphi^A, \dots, \llbracket o_n \rrbracket_\varphi^A \rangle)$ ,
- if  $R(o_1) = ([], b)$  and  $b^n = [K_1, \dots, K_n]$ , then  $\llbracket B(o_1, [X_1, \dots, X_n], o_2) \rrbracket_\varphi^A = \llbracket o_1 \rrbracket_\varphi^A(\langle V_1, \dots, V_n \rangle, F)$  where
  - $V_i \in U^{K_i}$  with  $V_i(k) = \llbracket X_i(k) \rrbracket_\varphi^A$  for  $i = 1, \dots, n$  and  $k \in K_i$ ,
  - $F = \Lambda u \in U^n \cdot \llbracket o_2 \rrbracket_{\varphi, u_1, \dots, u_n}^A$
- $\llbracket K(o | k := v) \rrbracket_\varphi^A = k^A(\llbracket o \rrbracket_\varphi^A, \llbracket v \rrbracket_\varphi^A)$ .

As in Def. 12, we call  $A$  **well-defined** if  $R_n^A$  contains all functions of the form  $\Lambda u \in (U^A)^n \cdot \llbracket o \rrbracket_{\varphi, u_1, \dots, u_n}^A$ .

Roled algebras are indeed a special case of Def. 10:

**Theorem 3.** *Given a roled vocabulary  $(T, r)$ , every roled algebra  $A$  over  $(T, r)$  induces an algebra  $B$  over  $T$  such that for all objects  $o \in O(T, C)$  with  $R(o) = \text{obj}$  and all assignments  $\varphi$  for  $C$  and  $A$ , we have  $\llbracket o \rrbracket_\varphi^A = \llbracket o \rrbracket_\varphi^B$ .*

*Proof.* Let  $U$  be the universe of  $A$ . The universe  $V$  of  $B$  is given by the closure of  $U \cup \mathcal{P}(\text{Keys}(T))$  under the formation of  $n$ -ary tuples and functions. We put  $R_n^B = U^{(V^n)}$ .

Then we define the necessary construction-oriented interpretation in  $B$  in the straightforward way:

- $s^B$  is given by  $s^A$  for  $s \in \text{Symbols}(T)$ ,
- $\@_n^B(f, u)$  is given
  - by  $\@_n^A(f, u)$  if  $f \in U$ ,  $u \in U^n$ ,
  - otherwise, by  $f(u)$  if  $f \in V \setminus U$  and  $f(u) \in V$  is defined,
  - otherwise, by  $\perp$ ,
- $\beta_K^B(u, v, w)$  is given
  - by  $u(r, w)$  if  $u(r, w) \in V$  is defined and where  $r = (r_1, \dots, r_n)$  is obtained from  $v : \bar{\sigma} \rightarrow U$  and  $K : \bar{\sigma} \rightarrow \text{Keys}(T)$  by  $r_i(K(i, j)) = v(i, j)$ ,
  - otherwise, by  $\perp$ ,
- $\alpha_k^B(u, v)$  is given by  $k^A(u, v)$  for  $k \in \text{Keys}(T)$ .

It is straightforward to show the two interpretation functions agree for objects with role **obj**.  $\square$

Here the construction of the universe  $V$  shows once again how much simpler algebras becomes if a role system is used.

Finally, it is straightforward to transfer the constructions of initial and quotient algebras from the roled to the unroled case:

**Definition 39 (Free Roled OM Algebra).** Let  $(T, r)$  be a roled OM vocabulary. Then the **free roled OM algebra**  $I := I(T)$  over  $T$  is defined by:

1.  $U^I = \{o \in O(T, \emptyset) \mid R(o) = \text{obj}\} / \equiv_\alpha$ ,
2.  $R_n^I$  is the set of functions  $\overline{\text{Subs}(o)}$  for  $o \in O(T, \langle x_1, \dots, x_n \rangle)$  with  $R(o) = \text{obj}$ , which are defined as follows:  $\overline{\text{Subs}(o)}([o_1]_\alpha, \dots, [o_n]_\alpha) = [\text{Subs}(o)\langle o_1, \dots, o_n \rangle]_\alpha$ ,
3.  $\@_n^I([f]_\alpha, \langle [o_1]_\alpha, \dots, [o_n]_\alpha \rangle) = [\mathbb{A}(f, o_1, \dots, o_n)]_\alpha$ ,
4. for  $r(s) = \text{obj}$ ,  $s^I = [\mathbb{S}(s)]_\alpha$ ,
5. for  $r(s) = \text{key}$ ,

$$s^I([o]_\alpha, [v]_\alpha) = [\mathbb{K}(o|k := v)]_\alpha$$

6. for  $r(s) = (a, b)$ ,

$$s^I(\langle [o_1]_\alpha, \dots, [o_n]_\alpha \rangle)(r, f) = [\mathbb{B}(\mathbb{A}(\mathbb{S}(s), o_1, \dots, o_n), [X_1, \dots, X_n], o)]_\alpha$$

where

- $b^{|r|} = [K_1, \dots, K_n]$

- $X_i$  is the variable  $x_i$  carrying for each  $k \in K_i$  one attribution with key  $k$  and some value  $v$  for which  $r_i(k) = [v]_\alpha$ ,
- $o \in O(T, \langle x_1, \dots, x_n \rangle)$  is some object such that  $\overline{Subs(o)} = f$ .

**Definition 40.** Given an equivalence relation  $\equiv$  on a set  $U$ , we extend it as follows

- For  $u, u' \in U^n$ , we write  $u \equiv u'$  if  $u_i \equiv u'_i$  for  $i = 1, \dots, n$ .
- For  $r, r' \in U^K$ , we write  $r \equiv r'$  if  $r(k) \equiv r'(k)$  for all  $k \in K$ .
- For  $r, r' \in U^b$ , we write  $r \equiv r'$  if  $|r| = |r'| = n$  and  $r_i \equiv r'_i$  for  $i = 1, \dots, n$ .
- For  $f, f' : U^n \rightarrow U$ , we write  $f \equiv f'$  if  $f(u) \equiv f(u')$  whenever  $u \equiv u'$ .

These are all equivalence relations, and we extend the definition of equivalence classes accordingly:

- For  $u \in U^n$ , we write  $[u]_\equiv = \{u' \in U^n | u \equiv u'\}$ .
- For  $r \in U^K$ , we write  $[r]_\equiv = \{r' \in U^K | r \equiv r'\}$ .
- For  $r \in U^b$ , we write  $[r]_\equiv = \{r' \in U^b | r \equiv r'\}$ .
- For  $f : U^n \rightarrow U$ , we write  $[f]_\equiv = \{f' : U^n \rightarrow U | f \equiv f'\}$ .

**Definition 41 (Congruence Relation).** Let  $(T, r)$  be a roled OM vocabulary and  $A$  a roled algebra over it with universe  $U$ . A **congruence relation** on  $A$  is an equivalence relation on  $U$  such that whenever applicable

1. if  $u \equiv u'$  and  $u_i \equiv u'_i$  for  $i = 1, \dots, n$ , then

$$@_n^A(u, \langle u_1, \dots, u_n \rangle) \equiv @_n^A(u', \langle u'_1, \dots, u'_n \rangle),$$

2. if  $r(s) = \text{key}$ , and  $u \equiv u'$  and  $v \equiv v'$ , then  $s^A(u, v) \equiv s^A(u', v')$ ,
3. if  $r(s) = (a, b)$ , and  $u \equiv u'$  for  $u, u' \in U^a$ , and  $r \equiv r'$  for  $r, r' \in U^b$  with  $|r| = |r'| = n$ , and  $f \equiv f'$  for  $f, f' : U^n \rightarrow U$ , then  $s^A(u)(r, f) \equiv s^A(u')(r', f')$ .

**Definition 42 (Quotient Algebra).** Let  $(T, r)$  be a roled OM vocabulary,  $A$  an OM algebra over  $(T, r)$ , and  $\equiv$  a congruence relation on  $A$ . Then the roled OM algebra  $Q := A/\equiv$  over  $T$  is defined by:

1. the applicative structure  $(U^Q, @_Q)$  where  $U^Q = U/\equiv$  and

$$@_Q([u]_\equiv, \langle [u_1]_\equiv, \dots, [u_n]_\equiv \rangle) = @_A(u, \langle u_1, \dots, u_n \rangle)]_\equiv$$

2.  $R_n^Q$  is the set of all functions of the form

$$f : (U^Q)^n \rightarrow U^Q, f([u_1]_\equiv, \dots, [u_n]_\equiv) = [F(u_1, \dots, u_n)]_\equiv$$

for some  $F \in R_n^A$ ,

3. for  $r(s) = \text{obj}$ ,  $s^Q = [s^A]_\equiv$ ,
4. for  $r(s) = \text{key}$ ,  $s^Q([u]_\equiv, [v]_\equiv) = [s^A(u, v)]_\equiv$ ,
5. for  $r(s) = (a, b)$ ,

$$s^Q(\langle [u_1]_\equiv, \dots, [u_n]_\equiv \rangle)([r]_\equiv, [f]_\equiv) = [s^A(\langle u_1, \dots, u_n \rangle)(r, f)]_\equiv$$

**Lemma 8.** In the situation of Def. 42,

- $Q$  is a well-defined roled OM algebra,

- for all  $o \in O(T, C)$  with  $R(o) = \text{obj}$  and all  $A$ -assignments  $\varphi = (\varphi_1, \dots, \varphi_n) \in U^n$ , it holds that  $[\![o]\!]_{\varphi}^A = [\![o]\!]_{\varphi'}^Q$  where  $\varphi'$  is the  $Q$ -assignment given by  $\varphi'_i = [\varphi_i]_{\equiv}$  for  $i = 1, \dots, n$ .

*Proof.* The proof proceeds analogous to the one of Lem. 6.  $\square$

The remaining constructions, in particular roled models, the induced congruence relation of a theory, and the Herbrand theorem are obtained analogously.

## 6. The Meaning of OpenMath and Content MathML

Now that we have defined a both full-coverage and a symbol-oriented semantics for OPENMATH objects, we are in a position to address the question of meaning of OPENMATH expressions. Concretely we propose a definition of the “meaning of OPENMATH expressions” in terms of our model theory for the OPENMATH standard and propose added levels of OPENMATH compliance. In the rest of the section we discuss the consequences for OPENMATH, MATHML and semantic questions of content dictionaries.

### 6.1. Proposals and Clarifications for the OpenMath Standard.

While the notion of meaning induced by phrasebooks must remain vague, the algebraic models for OPENMATH objects are now precisely defined. Moreover, we have shown that the symbol-oriented semantics from Section 5 is much simpler and more natural than the full-coverage OM models without losing legitimate representations.

In the light of this development we propose to adopt the role system from Section 5.1 and roled OM Models as the official meaning of OPENMATH (see Section 5.2). And with that we can finally define the meaning of OPENMATH objects.

**Definition 43 (The Meaning of an OpenMath Object).** Let  $o$  be a roled OPENMATH object, then we call the union of the theories of the CDs CDused in  $o$  the **Theory** of  $o$ . If  $o \in O(T, \langle \rangle)$  is a closed OM Object,  $\Theta$  its theory, and  $M$  a roled OM model of  $\Theta$ , then the **meaning of  $o$  in  $M$**  is  $[\![o]\!]^M$ .

The main advantage of fixing a semantics for OPENMATH is that we finally have a basis to define a non-trivial notion of OPENMATH compliance for mathematical software systems. The notion of compliance in the OPENMATH 2 standard is very syntactic and exhausts itself in requirements for signaling lexical errors. We propose to refine the definition of OPENMATH compliance into four levels (which include the previous ones):

1. The OPENMATH 2 is divided into two levels the syntactic rules for encodings from [BCC<sup>+</sup>04, Section 5.1/2] retained under the name of **syntactic compliance**. This is the lowest level of compliance.

2. The additional lexical rules about supporting the errors1 content dictionary and about signaling lexical errors from [BCC<sup>+</sup>04, Section 5.3/4] retained as **lexical compliance**.
3. A system  $S$  is **semantically compliant**, iff none of the manipulations performed by the system is inconsistent with the mathematical properties stated in the CDs.
4. A system  $S$  is **transformationally compliant**, iff all manipulations in the system are directly justified by a mathematical property in one of the applicable content dictionaries.

Note that we can also view the stipulations in the last two consistency levels as soundness and completeness conditions on the system  $S$ . Soundness is the minimal notion of “semantic” compliance we can stipulate, whereas completeness (together with soundness) the maximal; Note that transformational OPENMATH compliance can only be claimed by symbolic software systems like computer algebra systems or automated reasoning systems, and will be relatively hard to verify.

## 6.2. OM Models and OpenMath

We have shown that the free algebra of OPENMATH objects forms an initial algebra for “formulae with uninterpreted symbols” which is syntactic in nature as all initial algebras are. Indeed, for OPENMATH and content MATHML expressions that do not contain symbols — and are thus unrestricted by content dictionaries — this is the best meaning we can hope for: OPENMATH cannot impose more restrictions than  $\alpha$ -equivalence and flattening of attributions without losing coverage. This is captured by the algebraic semantics of OPENMATH expressions in Section 5.

But the meaning of an OPENMATH object comes mainly from the mathematical properties in the content dictionaries of its symbols. In section 4 we have been able to show that this can be grafted onto the algebraic semantics by interpreting OPENMATH CDs as logical theories over a foundational system (an OM logic).

## 6.3. OM Models and MathML

Note that there has been a substantial change to the way the meaning of content MATHML expressions are given meaning in MATHML 3.

*MathML 3 assigns semantics to content markup by defining a mapping to Strict Content MathML. Strict MathML, in turn, is in one-to-one correspondence with OpenMath, and the subset of OpenMath expressions obtained from content MathML expressions in this fashion all have well-defined semantics via the standard OpenMath Content Dictionary set.*

Before, the meaning of the content MATHML elements had been given by informal statements of the form

*The quotient element is the operator used for division modulo a particular base. When the quotient operator is applied to integer arguments  $a$  and  $b$ , the result is the “quotient of  $a$  divided by  $b$ ”. That is, quotient returns the unique integer  $q$  such that  $a = qb + r$ . (In common usage,  $q$  is called the quotient and  $r$  is the remainder.)*

Nonetheless, the MATHML 3 recommendation states that

*The meaning of the actual content remains as before in principle, but a lot of work has been done on expressing it better.*

And indeed, earlier versions of MATHML had an appendix (Appendix C in [ABC<sup>+</sup>03]) that stated “default definitions” for the concepts in a way that is structurally very similar to OPENMATH CDs (symbols have a description, properties, examples, and sometimes type information). Furthermore, all content MathML elements can carry a `definitionURL` attribute that serves as a link to a specification of the meaning, which defaulted to “Appendix C”. So while the target of these links changed to OPENMATH content dictionaries, the mechanism essentially stays the same. Moreover, the MATHML and OPENMATH groups have aligned the OPENMATH CDs with Appendix C, so arguably the meaning of MATHML expressions has only been clarified in MATHML 3. Most of the clarifications have actually been to the meaning of non-strict MATHML expressions which were not mentioned in Appendix C, there the “strict content MATHML translation” [ABC<sup>+</sup>10b, Section 4.6] had to make choices about previously under-defined edge-cases, which now get their meaning via the translation. Definition 43 above would further clarify the meaning of MATHML expressions.

#### 6.4. Descriptions and Examples in OpenMath CDs

Note that our semantic analysis of OPENMATH content dictionaries has only taken into account symbol names, roles, and mathematical properties. The former two are relevant for the OM vocabularies and the latter for the OM theories that give OPENMATH symbols their meaning. In particular, we did not look at descriptions (for symbols or whole CDs) or examples. The status of these CD parts is left unspecified by the OPENMATH2 standard, and usage in actual CDs is non-uniform. Symbol descriptions reach from appealing to the folklore — e.g. “*This symbol represents the Boolean value true.*” [CDI04] to specific literature references e.g. “*See CRC Standard Mathematical Tables and Formulae, editor: Dan Zwillinger, CRC Press Inc., 1996, (7.7.11) section 7.7.1.*” [CDs04]. Arguably both forms “mean” something to the human reader, and especially the latter should surely contribute to the theory. The case of examples in CDs is similarly unclear: if they were uninformative to the human reader, nobody would put them in. But again practice in published CDs is no help: examples are often statements — and thus in principle mathematical properties — about (mathematical objects constructed by) the symbols they illustrate, and — if they are — they tend to be valid, but it would be incautious to assume this to be generally the case or even a normative part of the CD. The next version of the OPENMATH standard could of course clarify these issues at the cost of making it more heavyweight and thus arguably less useful. We propose to use the OMDOC format [Koh06] that already addresses these issues for specifying content dictionaries instead if the additional functionality is desired.

### 6.5. Informal Parts of OpenMath CDs

In OPENMATH there are two kinds of mathematical properties: “commented mathematical properties” (encoded as **CMP** elements which contain mathematical vernacular) and “formal mathematical properties” (encoded as **FMP** elements that contain XML encodings of OPENMATH objects). We have concentrated on the latter in this paper since they provide more structure. This is no loss of generality, given the assumption in mathematical practice that any rigorously stated property can be fully formalized.

### 6.6. Underspecification in OpenMath Content Dictionaries

At this it is useful to distinguish the concepts of informality (as discussed above) and underspecification. We call a language informal, if we have no (simple/algorithmic) way of finding out grammaticality and the internal structure. In this sense natural language texts are informal, and therefore not open to the methods. Underspecification is a different matter, it comments on the precision of description, and thus how many (intrinsically different) models a formula has.

An objection often brought up against the “semantics of OPENMATH” is that the standard CDs maintained by the OPENMATH society are very weak, and do not give a clear and unambiguous meaning for K-14 mathematics. The OPENMATH society (and the W3C Math Working Group for that matter) view<sup>8</sup> the weakness of the standard OPENMATH/MATHML CD group as a feature and not a bug. These CDs contain fewer mathematical properties to allow them to describe larger model classes. For instance the CD **arith1** [CDa04] (somewhat) corresponds to the class of (Abelian) semigroups. This is supposed to capture the usage in K-14, where addition is employed in many contexts. We need the flexibility offered by the OPENMATH/MATHML CDs so that we do not over-specify the meaning. We would probably not want to scare elementary school children who are struggling with long division with the Peano Axioms or teenagers in high school with the subtle differences between Riemann and Lebesque integration. Essentially, the OPENMATH content dictionaries make use of underspecification to cover more use cases at the cost of precision.

The main point of the OPENMATH philosophy however is that the set of CDs is open-ended, and that we can build CDs to suit all our communication and representation needs. In particular it is possible (and in fact rather simple) to build a CD **NatArith** for natural numbers and arithmetic by encoding the Peano Axioms and recursive equations for the arithmetical operators in OPENMATH objects so that its theory  $\Theta = \Theta(\text{NatArith})$  determines the class of  $\Theta$ -models up to isomorphism (and all are isomorphic to  $\mathbb{N}$ ). To see this just use the standard proof with our notion of OM models from section 4. Note that this argument reveals another source of underspecification we have not discussed yet: the implicit foundational

---

<sup>8</sup> Statements about the views of these bodies are made to the best knowledge of the first author who has a co-author of the OPENMATH 2 standard, an active member of the W3C Math WG for more than a decade, and currently serves as the president of the OPENMATH society.

system. Following accepted mathematical practice we might assume the foundation to be first-order logic (with a choice operator) and a version of axiomatic set theory as a theory of first-order logic, e.g. Zermelo-Fraenkel set theory with choice [Zer08, Fra22] since this is the best-known one. But in OPENMATH practice, commented mathematical properties seem to assume FOL+ZFC as a foundational system, whereas FMPs make due with less: they usually only use symbols from the following CDS:

- `logic1` [CDl04] supplies the symbols `true`, `false` — which we take as the distinguished symbols  $\top$  and  $\perp$  — and the usual propositional connectives,
- `relation1` [CDr04] supplies the symbol `eq`, which we take as  $=$ ,
- `quant1` [CDq04] supplies the first-order quantifiers.

The first two together form a logic<sup>9</sup> in the sense of Definition 16; an underspecified version of first-order logic (since not all the properties are encoded in these CDs). So to make the models of  $\Theta(\text{NatArith})$  unique up to isomorphisms, we would need two basic CDs for first-order logic (declaring connectives, quantifiers, equalities, and choice) and ZFC (declaring membership and axioms). Note that any other foundation of mathematics would serve equally well for our purposes. OPENMATH does not supply these, but our LATIN logic atlas [CHK<sup>+</sup>11, KMR] has OMDoc content dictionaries that would fit the bill.

We see that it is possible to specify the meaning of mathematical objects and formulae at many levels of flexibility and rigorousness and extend the invitation to our readers to do just that: to contribute content dictionaries to the community of mathematicians (by way of the OPENMATH society CD site [OMC]).

## 7. Conclusion

In this paper we have tried to clarify the meaning of OPENMATH (and thus MATHML3) expressions by giving an algebraic model theory. We have presented a full-coverage construction-based semantics as well as a symbol-based semantics for a subset of OPENMATH expressions given by a role system and propose the latter as the (core of) the official semantics for OPENMATH expressions.

Note that this stipulation only takes up on one of the two intuitions about meaning from the OPENMATH standard (see Section 1.2 for a discussion). Indeed we claim that it is the more direct and simpler one. The system-interoperability intuition behind the integration-via-phrasebooks can be mapped to a general integration problem between symbolic software systems. We have mapped out a foundational approach at solving these in [KRSC11], which would be applicable, since it is based on theories (CDs) and theory morphisms (the translations induced by phrasebooks). But the representational system used there is based on OPENMATH and thus pre-supposes the work reported in this paper.

---

<sup>9</sup>Note that in contrast to our definitions from section 4.1, the signature of a CD will already contain the OM logic, as OPENMATH does not distinguish OM logics from other CDs.

## References

- [ABC<sup>+</sup>03] Ron Ausbrooks, Stephen Buswell, David Carlisle, Stéphane Dalmas, Stan Devitt, Angel Diaz, Max Froumentin, Roger Hunter, Patrick Ion, Michael Kohlhase, Robert Miner, Nico Poppelier, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. Mathematical Markup Language (MathML) version 2.0 (second edition). W3C recommendation, World Wide Web Consortium (W3C), 2003.
- [ABC<sup>+</sup>10a] Ron Ausbrooks, Stephen Buswell, David Carlisle, Giorgi Chavchanidze, Stéphane Dalmas, Stan Devitt, Angel Diaz, Sam Dooley, Roger Hunter, Patrick Ion, Michael Kohlhase, Azzeddine Lazrek, Paul Libbrecht, Bruce Miller, Robert Miner, Murray Sargent, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. Mathematical Markup Language (MathML) version 3.0. W3C Recommendation, World Wide Web Consortium (W3C), 2010.
- [ABC<sup>+</sup>10b] Ron Ausbrooks, Stephen Buswell, David Carlisle, Giorgi Chavchanidze, Stéphane Dalmas, Stan Devitt, Angel Diaz, Sam Dooley, Roger Hunter, Patrick Ion, Michael Kohlhase, Azzeddine Lazrek, Paul Libbrecht, Bruce Miller, Robert Miner, Murray Sargent, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. Mathematical Markup Language (MathML) version 3.0. W3C Recommendation, World Wide Web Consortium (W3C), 2010.
- [AvLS98] John Abbott, Andre van Leeuwen, and Andreas Strotmann. Openmath: Communicating mathematical information between co-operating agents in a knowledge network. *Journal of Intelligent Systems*, 8, 1998.
- [BBK04] Christoph Benzmüller, Chad Brown, and Michael Kohlhase. Higher order semantics and extensionality. *Journal of Symbolic Logic*, 69:1027–1088, 2004.
- [BCC<sup>+</sup>04] Stephen Buswell, Olga Caprotti, David P. Carlisle, Michael C. Dewar, Marc Gaëtano, and Michael Kohlhase. The Open Math standard, version 2.0. Technical report, The OpenMath Society, 2004.
- [BF85] J. Barwise and S. Feferman, editors. *Model-Theoretic Logics*. Springer-Verlag, 1985.
- [CC98] O. Caprotti and A. Cohen. A type system for OpenMath. Technical report, Esprit Project OpenMath, 1998.
- [CDa04] **arith1**. Openmath content dictionary, The OpenMath Society, 2004.
- [CDl04] **logic1**. Openmath content dictionary, The OpenMath Society, 2004.
- [CDq04] **quant1**. Openmath content dictionary, The OpenMath Society, 2004.
- [CDr04] **relation1**. Openmath content dictionary, The OpenMath Society, 2004.
- [CDs04] **sdata1**. Openmath content dictionary, The OpenMath Society, 2004.
- [CHK<sup>+</sup>11] Mihai Codescu, Fulvia Horozal, Michael Kohlhase, Till Mossakowski, and Florian Rabe. Project abstract: Logic atlas and integrator (latin). In Davenport et al. [DFRU11], pages 289–291.
- [Chu40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.
- [Dav99] James H. Davenport. A small OPENMATH type system. Technical report, The OPENMATH Esprit Project, 1999.

- [DFRU11] James Davenport, William Farmer, Florian Rabe, and Josef Urban, editors. *Intelligent Computer Mathematics*, number 6824 in LNAI. Springer Verlag, 2011.
- [Fra22] Adolf Abraham Fraenkel. Der Begriff “definit” und die Unabhängigkeit des Auswahlsaxioms. *Sitzungsberichte der Preussischen Akademie der Wissenschaften, Physikalisch-mathematische Klasse*, 1922. reprinted as [Fra67].
- [Fra67] Adolf Abraham Fraenkel. The notion of “definite” and the independence of the axiom of choice. Source books in the history of the sciences series, pages 284–289. Harvard Univ. Press, Cambridge, MA, 3<sup>rd</sup> printing, 1997 edition, 1967.
- [KMR] Michael Kohlhase, Till Mossakowski, and Florian Rabe. Latin: Logic atlas and integrator. <http://latin.omdoc.org>.
- [Koh06] Michael Kohlhase. OMDOC – *An open markup format for mathematical documents [Version 1.2]*. Number 4180 in LNAI. Springer Verlag, August 2006.
- [KRSC11] Michael Kohlhase, Florian Rabe, and Claudio Sacerdoti Coen. A foundational view on integration problems. In Davenport et al. [DFRU11], pages 107–122.
- [LS86] J. Lambek and P. Scott. *Introduction to Higher-Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986.
- [OMC] OPENMATH content dictionaries. web page at <http://www.openmath.org/cd/>. seen June2008.
- [Rab08] Florian Rabe. *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen, 2008.
- [RK08] Florian Rabe and Michael Kohlhase. An exchange format for modular knowledge. In G. Sutcliffe, P. Rudnicki, R. Schmidt, B. Konev, and S. Schulz, editors, *Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and The 7<sup>th</sup> International Workshop on the Implementation of Logics*, number 418 in CEUR Workshop Proceedings, pages 50–68, Aachen, 2008.
- [RK09] Florian Rabe and Michael Kohlhase. A better role system for OpenMath. In James H. Davenport, editor, *22<sup>nd</sup> OpenMath Workshop*, July 2009.
- [RK11] Florian Rabe and Michael Kohlhase. A scalable module system. Manuscript, submitted to Information & Computation, 2011.
- [Rob50] A. Robinson. On the application of symbolic logic to algebra. In *Proceedings of the International Congress of Mathematicians*, pages 686–694. American Mathematical Society, 1950.
- [Str04] Andreas Strotmann. The categorial type of openmath objects. In Andrea Asperti, Grzegorz Bancerek, and Andrej Trybulec, editors, *Mathematical Knowledge Management, MKM’04*, number 3119 in LNAI, pages 378–392. Springer Verlag, 2004.
- [TV56] A. Tarski and R. Vaught. Arithmetical extensions of relational systems. *Compositio Mathematica*, 13:81–102, 1956.
- [WR13] A. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, 1913.

- [Zer08] E. Zermelo. Untersuchungen ber die Grundlagen der Mengenlehre I. *Mathematische Annalen*, 65:261–281, 1908. English title: Investigations in the foundations of set theory I.

Michael Kohlhase  
Computer Science, Jacobs University Bremen, Germany  
<http://kwarc.info/kohlhase>

Florian Rabe  
Computer Science, Jacobs University Bremen, Germany  
<http://kwarc.info/frabe>

# A Scalable Module System

Florian Rabe<sup>a</sup>, Michael Kohlhase<sup>a</sup>

<sup>a</sup>*Jacobs University Bremen, Computer Science, Germany*

---

## Abstract

Symbolic and logic computation systems ranging from computer algebra systems to theorem provers are finding their way into science, technology, mathematics and engineering. But such systems rely on explicitly or implicitly represented mathematical knowledge that needs to be managed to use such systems effectively.

While mathematical knowledge management (MKM) “*in the small*” is well-studied, scaling up to large, highly interconnected corpora remains difficult. We hold that in order to realize MKM “*in the large*”, we need representation languages and software architectures that are designed systematically with large-scale processing in mind.

Therefore, we have designed and implemented the MMT language – a module system for mathematical theories. MMT is designed as the simplest possible language that combines a module system, a foundationally uncommitted formal semantics, and web-scalable implementations. Due to a careful choice of representational primitives, MMT allows us to integrate existing representation languages for formal mathematical knowledge in a simple, scalable formalism. In particular, MMT abstracts from the underlying mathematical and logical foundations so that it can serve as a standardized representation format for a formal digital library. Moreover, MMT systematically separates logic-dependent and logic-independent concerns so that it can serve as an interface layer between computation systems and MKM systems.

---

---

*Email addresses:* f.rabe@jacobs-university.de (Florian Rabe),  
m.kohlhase@jacobs-university.de (Michael Kohlhase)  
*URL:* <http://kwarc.info/frabe/> (Florian Rabe), <http://kwarc.info/kohlhase/> (Michael Kohlhase)

## Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>                                   | <b>4</b>  |
| <b>2</b> | <b>Features of Knowledge Representation Languages</b> | <b>6</b>  |
| 2.1      | Packages and Modules . . . . .                        | 6         |
| 2.2      | Inheritance . . . . .                                 | 7         |
| 2.3      | Realizations . . . . .                                | 9         |
| 2.4      | Semantics . . . . .                                   | 11        |
| 2.5      | Genericity . . . . .                                  | 11        |
| 2.6      | Degree of Formality . . . . .                         | 12        |
| 2.7      | Scalability . . . . .                                 | 13        |
| <b>3</b> | <b>Central Features of MMT</b>                        | <b>14</b> |
| <b>4</b> | <b>Syntax</b>   | <b>23</b> |
| 4.1      | MMT Theory Graphs . . . . .                           | 23        |
| 4.1.1    | Grammar . . . . .                                     | 23        |
| 4.1.2    | Identifiers . . . . .                                 | 24        |
| 4.1.3    | The Object Level . . . . .                            | 25        |
| 4.1.4    | The Symbol Level . . . . .                            | 27        |
| 4.1.5    | The Module Level . . . . .                            | 29        |
| 4.2      | Realizations . . . . .                                | 30        |
| 4.3      | Valid Declarations . . . . .                          | 33        |
| 4.4      | Normal Terms . . . . .                                | 38        |
| <b>5</b> | <b>Well-formed Expressions</b>                        | <b>39</b> |
| 5.1      | Judgments . . . . .                                   | 40        |
| 5.2      | Foundations . . . . .                                 | 41        |
| 5.3      | Inference Rules for the Structural Levels . . . . .   | 43        |
| 5.4      | Inference Rules for Morphisms . . . . .               | 45        |
| 5.5      | Inference Rules for Terms . . . . .                   | 46        |
| 5.6      | Module-Level Reasoning . . . . .                      | 47        |
| <b>6</b> | <b>Formal Properties</b>                              | <b>48</b> |
| 6.1      | Theory Graphs . . . . .                               | 48        |
| 6.2      | Properties of Morphisms . . . . .                     | 49        |
| 6.3      | Structural Well-Formedness . . . . .                  | 52        |
| 6.4      | Flattening . . . . .                                  | 55        |
| <b>7</b> | <b>Specific Foundations</b>                           | <b>58</b> |
| 7.1      | OpenMath . . . . .                                    | 58        |
| 7.2      | The Edinburgh Logical Framework (LF) . . . . .        | 59        |

|           |   |           |
|-----------|---|-----------|
| <b>8</b>  | <b>Web-Scalability</b>                                | <b>60</b> |
| 8.1       | Documents and Libraries . . . . .                     | 60        |
| 8.2       | XML-based Concrete Syntax . . . . .                   | 61        |
| 8.3       | URI-based Addressing . . . . .                        | 65        |
| 8.4       | An API for Knowledge Management . . . . .             | 67        |
| <b>9</b>  | <b>Implementations</b>                                | <b>69</b> |
| 9.1       | The MMT Reference Implementation . . . . .            | 69        |
| 9.2       | TNTbase – a Scalable MMT-Compliant Database . . . . . | 70        |
| 9.3       | Twelf – an MMT-Compliant Logical Framework . . . . .  | 71        |
| <b>10</b> | <b>Related Work</b>                                   | <b>71</b> |
| <b>11</b> | <b>Conclusion and Future Work</b>                     | <b>79</b> |
| 11.1      | The MMT Language . . . . .                            | 80        |
| 11.2      | Beyond MMT . . . . .                                  | 81        |
| 11.3      | Applying MMT . . . . .                                | 85        |

## 1. Introduction

Mathematics is one of the oldest areas of human knowledge and provides science with modeling tools and a knowledge representation regime based on rigorous language. However, mathematical knowledge is far too vast to be understood by one person – it has been estimated that the total amount of published mathematics doubles every ten to fifteen years [Odl95]. Indeed, for example, Zentralblatt Math [ZBM31] maintains a database of 2.9 million reviews for articles from 3500 journals from 1868 to 2010.

The currently practiced way to organize mathematical knowledge is to have humans build a cognitive representation of the contents in their minds and to communicate their results in natural – i.e., informal – language with interspersed formulas. This process is well-suited for doing mathematics “in the small” where human creativity is needed to create new mathematical insights. But the sheer volume of mathematical knowledge precludes this approach from organizing mathematics “in the large”: Except for prestige projects such as the classification of finite simple groups [Sol95], collaboration in mathematics is largely small-scale.

But this leads to increasing specialization and missed opportunities for knowledge transfer, and the question of supporting the management and dissemination of mathematical knowledge in the large remains difficult. This problem has been tackled in the field of *mathematical knowledge management* (MKM), which uses explicitly annotated content as the basis for mathematical software services such as semantics-based searching and navigation. MKM in the large has been pioneered in the field of *formal methods in software engineering*, where a sound logical foundation and the incorruptibility of computers are combined to verify computer systems. These computer-aided proofs rely on large amounts of formal knowledge about the programming language constructs and data structures, and the productivity of formal methods is restricted in practice by the effectiveness of managing this knowledge.

We currently see five obstacles for large scale computerized MKM:

**Informality** As computer programs still lack any real understanding of mathematics, human mathematicians must make structures in mathematical knowledge sufficiently explicit. This usually means that the knowledge has to be formalized, i.e., represented in a formal, logical system. While it is generally assumed that all mathematical knowledge can in principle be formalized, this is so expensive that it is seldom even attempted.

**Logical Heterogeneity** One of the advantages of informal, but rigorous mathematics is that it does not force the choice of a formal system. There are many formal systems, each optimized for expressing and reasoning about different aspects of mathematical knowledge. All attempts to find the “mother of all logical systems” (and convince others to use it) have failed. Even though logics themselves can be made the objects of mathematical investigation and even of formalization (in logical frameworks), we do not have scalable methods for

efficiently dealing with heterogeneous, i.e., multi-logic, presentations of mathematical knowledge.

**Foundational Assumptions** Logical heterogeneity is not only a matter of optimization because different developments of mathematical knowledge make different foundational assumptions. For example, classical mathematics usually assumes some kind of set theory as a foundation and embraces a platonist philosophy. But there are different ones of differing expressivity, such as those with and those without the axiom of choice. Other mathematicians even reject the law of excluded middle or insist on constructive witnesses for existential theorems. Corresponding developments often take a more formalist stance and use type theoretic foundations.

**Modularity** Modern developments of mathematical knowledge are highly modular. They take pains to identify minimal sets of assumptions so that results are applicable at the most general possible level. This modularity and the mathematical practice of “framing”, i.e., of viewing objects of interest in terms of already understood structures, must be supported to even approach human capabilities of managing mathematical knowledge in computer systems.

**Global Scale** Mathematical research and applications are distributed globally, and mathematical knowledge is highly interlinked by explicit and implicit references. Therefore, a computer-supported management system for mathematical knowledge must support global interlinking and framing as well as management algorithms that scale up to very large (global) data sets.

In this paper we contribute to a uniform solution of four of the five challenges: We give a globally scalable module system for mathematical theories (MMT) that abstracts from and mediates between different logics and foundations.<sup>1</sup> With this, we lay a conceptual and technical foundation for formal MKM in the large.

Because our solution draws intuitions from the fields of mathematics, formal methods, and knowledge management, we give a comprehensive overview over the relevant language features and introduce a terminology for them in Sect. 2. This gives us a solid footing to describe the central design choices underlying MMT in Sect. 3. Then we describe the formal syntax of MMT in Sect. 4 and an inference system that defines the well-formed expressions in Sect. 5. In Sect. 6, we discuss the meta-theoretical properties of MMT, which include a flattening algorithm that defines the semantics of modular MMT-expressions. The semantics of MMT is parametric in what we call foundations, and we look at particular foundations in Sect. 7. Then we discuss the web scalability of MMT and our implementations in Sect. 8 and 9. Finally we give an extensive discussion of related representation languages in Sect. 10 and conclude in Sect. 11.

---

<sup>1</sup>We have already solved the integration of formal and informal mathematical knowledge in the OMDoc format, whose formal part is a predecessor of the work presented in this paper. We plan to integrate this solution with the much stronger formal basis of MMT in the future.

## 2. Features of Knowledge Representation Languages

In order to compare MMT to other representation languages for mathematical knowledge, we will first develop a classification vocabulary that will allow us to place MMT in the taxonomy of modular, machine-processable knowledge representation languages. We will also use this vocabulary in Sect. 10 to compare MMT to other module systems.

By a **module system**, we mean a formal language that provides constructs to express high-level design patterns such as namespaces, imports, parametricity, encapsulation, etc. Very often the modular features of a language can be separated from the non-modular ones. In that case, we call the fragment containing no modularity the **base language**. Typical base languages are logics, type theories, or programming languages. Base language and module system can be designed together or independently, and in the latter case the module system may be designed before or after the base language. We will sometimes use the phrases **modular expression** and **base expression** to distinguish expressions of the module system and the base language.

### 2.1. Packages and Modules

Module systems typically feature one or both of two main scoping devices. Unfortunately, these overlap, and even when they are distinguished, there is no universal convention on how to name them. We will use the names *package* and *module*. Other names in common use for package are “library”, “namespace”, and “module”; the latter is the one used in Modula [Wir77], one of the first systems with this functionality. Other names used instead of *module* are “theory”, “signature”, “specification”, “(type) class”, “(module) type”, and “locale”.

**Packages** provide scopes for the grouping of related toplevel declarations into – possibly nested – components. The main purpose of packages is **namespace management**: Packages have names, and their named toplevel declarations are identified by a qualified name: a pair of a package name and a declaration name. This facilitates reuse and distribution of declarations over files and networks. Often the packaging structure is transparent to the semantics of the language; in that case the semantics of packages is that they identify and locate the available toplevel declarations.

When identifying these declarations, we distinguish **open** and **closed** packaging. With open packages, all packages can refer all toplevel declarations in all other packages via qualified names. With closed packages, import declarations are necessary as only explicitly imported declarations are accessible. In both cases, import declarations are often used to make the imported declarations available without qualification.

When locating these declarations, we speak of **logical** package identifiers if package identifiers are different from physical locations as given by file systems, databases, and networks; otherwise, we speak of **physical** identifiers. With logical identifiers, the location of resources requires a resolution algorithm that maps logical identifiers to physical locations. This resolution can be relegated to an extra-linguistic **catalog**. Catalogs provide an abstraction layer that makes

the distribution of resources over physical locations transparent to the language and avoids conflicts due to naming conventions of operating systems and storage solutions. Using URI-based package identifiers, logical identifiers can be made globally unique to support global interlinking.

Typically, the declarations in a package are module declarations, and a package can be seen as a group of modules. But there are also languages featuring only packages or only modules. In the former case, every package can be considered to contain a single unnamed module; this is the case in many XML-related languages where the packages are called namespaces such as in XQuery [W3C07]. In the latter case, all modules share the same namespace, which can be considered to form a single unnamed package; this is the case in SML where a configuration file is used to list the files over which the modules are distributed. We will call the latter **single-package** module systems.

Like packages, **modules** are scoped groups of declarations. But contrary to packages, modules are opaque to the language semantics and are used to realize modular design patterns such as inheritance, instantiation, and hiding. For example, moving a declaration between packages has no semantic consequences except that references to the moved declaration must be updated. But a module has a meaning itself that will be affected if a declaration is removed or added. For example, a mathematical theory should be represented as a module because moving axioms between theories changes the semantics; a mathematical paper should be represented as a package because some parts may be relegated to other papers.

Typically, languages provide a number of different types of declarations that may occur in a module. The most typical declarations are sorts and types, constants and values, operations and functions, and predicates. These are usually named. Further examples of named or unnamed declarations are axioms, theorems, inference rules, abbreviations, or notations for parsing and printing. A named declaration within a module can often be identified as a triple of package name, module name, and declaration name.

## 2.2. Inheritance

In the simplest case, **inheritance** is a binary relation between modules, which is usually seen as an inheritance graph whose nodes are the modules and whose edges make up the inheritance relation. The individual edges are called **imports**: If  $T$  inherits from  $S$ , then  $T$  imports all knowledge items of  $S$ , which then become available in  $T$ . An important distinction is whether the individual imports are **named** or **unnamed**. In the former case, the name of the import is available to refer to (*i*) the imported module as a whole, or (*ii*) the imported knowledge items via qualified names.

Other names for named imports are “structure” and “instance”. Other names for unnamed imports are “mixin”, “inclusion”, “inheritance”, and “definitional morphism/link”.

Inheritance leads to a **diamond situation** when the same module is imported in two different ways. The language may **identify** multiple imports of

the same module or **distinguish** them. For named imports, the distinguish-semantics is natural because the multiply imported knowledge items can have different qualified names. But then **sharing** declarations are necessary to force the identification of these items. The identify-semantics is more natural with unnamed imports. Then **renaming** declarations are needed to force the distinction of multiply imported knowledge items.

A related problem is the **import name clash**, which arises when unnamed imports import from different modules which happen to contain knowledge items with the same local name. In large-scale developments, this is a very typical situation, which can be difficult to detect. Here module systems may signal an **error**, the knowledge item imported first can be **shadowed** by the one imported later, the name of the module can be used to form a unique **qualified** name, or **overload/identify**-semantics can be used. In the latter case, overloading resolution is used to disambiguate a reference to a knowledge item; and knowledge items that cannot be distinguished in this way (e.g., because they have the same types) are identified.

A more complex form of inheritance is **instantiation**. It means that when importing  $S$  into  $T$ , some names declared in  $S$  may be mapped to expressions of  $T$ . This set of mappings can be seen as the passing of argument values over  $T$  to parameters of  $S$ . If instantiations are possible, multiple imports of the same module with different instantiations should be distinguished. Therefore, the distinguish-semantics is more natural. But it is also possible to identify two imports iff they use the same instantiations.

Module systems differ as to what kind of mappings are allowed. Some systems only allow the map of  $S$ -symbols to  $T$ -symbols. This has the advantage that it is easier to check whether a map is well-typed. Other systems allow mapping symbols to composed expressions. And systems with named imports, can permit the map of an import itself to a realization (see below).

Another difference is which symbols or imports may be instantiated: We speak of a **free** instantiation if arbitrary symbols or imports can be instantiated. Free instantiations must **explicitly** associate some names of  $S$  with expressions of  $T$ . And we speak of **interfaced** instantiation if the declarations of  $S$  are divided into two blocks, and only the declarations in the first block — the interface — are available for instantiations. Interfaced instantiations are often **implicit**: The order of declarations in the interface of  $S$  must correspond to the order of provided  $T$ -expressions. Furthermore, instantiations may be **total** or **partial**: Total instantiations provide expressions for all symbols or imports in (the interface of)  $S$ . Finally, some systems restrict inheritance to axioms; in such systems, imports must carry instantiations for all symbols; we speak of **axiom-inheritance**.

A further distinction regards the relation between the imports and the other declarations. We speak of **separated** imports if all imports must be given at the beginning of the module; otherwise, we call them **interspersed** imports. Separated imports are conceptually easier, but less expressive: At the beginning of a module, less syntactic material is available to form expressions that can be used in instantiations.

More general forms of imports permit **hiding** and **filtering** of declarations. Both are similar syntactically but not semantically. When importing from  $S$  to  $T$ , filtering a declaration of  $S$  means to exclude that declaration from the import. Hiding is more complicated – one way to think of it is that if a declaration is hidden, it is still imported but rendered inaccessible. In both cases, it is necessary to maintain a dependency relation between declarations: If a declaration is hidden or filtered, so must be all declarations that depend on it.

Hiding can be quite difficult to formalize but has an elegant interpretation in the context of algebraic specification. There, it is used to represent the hiding of implementation details or auxiliary constants. For example, implementations of a specification  $S$  must also implement the hidden functions of  $S$ , but are considered equal if they differ only in the implementation of hidden functions. More precisely, we speak of **simple** hiding.

**Complex** hiding arises if not only declarations, i.e., atomic expressions, can be hidden but composed expressions as well. Syntactically, a complex hiding from  $S$  to  $T$  can be seen as a morphism from  $T$  to  $S$  in a category of specifications. Then simple hiding is the special case where this morphism is an inclusion. Complex hiding has the appeal that instantiation and hiding become dual to each other.

### 2.3. Realizations

Many module systems use a concept that we will call **realization**. Its treatment can vary substantially between systems, which makes it more difficult to describe abstractly. The common intuition is that we can often think of a module as a specification, an interface, or a behavioral description. Then the realizations are the objects that conform to such a specification. Further names used instead of “realization” are “interpretation”, “structure”, “instance”, and “(module) term/value/expression”. Very often it is fruitful to consider modules as types and apply the intuitions of type theory to them. Then a realization is a value that is typed by a module.

For example, in SML, the structures are the realizations of the signatures. In Java, the instances of concrete classes are the realizations of the abstract classes and the interfaces. In logic, the models are the realizations of the theories. In formal specification, the implementations are the realizations of the specifications.

More concretely, a realization of a module  $S$  in terms of some context  $C$  must provide values over  $C$  for all symbols declared in the module  $S$ . Two special cases are of particular importance.

Firstly, if  $C$  is the empty context – or more precisely: the global environment implicitly determined by the base language – we speak of **grounded** realizations. For example, in formal specification, the grounded realizations of  $S$  are the programs implementing  $S$ ; the implicit global environment is given by the built-in datatypes and values of the programming language. In logic, the grounded realizations are the models of  $S$ ; the implicit global environment is given by the foundation of mathematics, e.g., set theory.

Secondly, if  $C$  is another module  $T$ , we obtain the notions of “views from  $S$  to  $T$ ” and of “functors from  $T$  to  $S$ ”. They are dual in the sense that a view from  $S$  to  $T$  is a functor from  $T$  to  $S$  and vice versa. But because they are often associated with very different intuitions, this duality is rarely explicated. We can also recover imports as a special case of realizations akin to views.

**Functors** are associated with the intuitions of type theory: If modules are seen as types and realizations as values, then functors are the module-level analogue of functions. If  $r(x)$  is a realization of  $T$  that is given in terms of a realization  $x$  of  $S$ , then  $\lambda$ -abstraction yields a functor  $\lambda x : S.r(x)$ . For such a functor, **functor application** maps a realization of  $S$  to a realization of  $T$  by  $\beta$ -reduction. A module system is **higher-order** if functors may take other functors as arguments.

**Views** are associated with the intuitions of category theory: Many declarative languages can be naturally formulated as categories with modules as objects and views as morphisms. A view from  $S$  to  $T$  interprets all declarations of  $S$  in terms of  $T$ , and this often yields a **homomorphic extension** that maps expressions over  $S$  to expressions over  $T$ : All symbols in an  $S$ -expression are replaced with their  $T$ -definition provided by  $r$ . Other names used instead of “view” are “signature/theory/specification morphism” and “postulated morphism/link”.

**Imports** are similar to views in that every import from  $S$  to  $T$  yields a realization of  $S$  in terms of  $T$ . Using the intuitions of type theory, declaring a named import from  $S$  can be seen as the declaration of a symbol of type  $S$ .

The duality between views and functors is connected to a duality of two important translation functions. Consider a realization  $r$  of  $S$  in terms of  $T$ . The **syntactic translation** maps  $S$ -expressions to  $T$ -expressions using the homomorphic extension of  $r$ . This is closely related to the intuition of  $r$  as a view from  $S$  to  $T$ . The **semantic translation** maps realizations of  $T$  to realizations of  $S$  by functor application. This is closely related to the intuition of  $r$  as a functor from  $T$  to  $S$ .

For example, let  $S$  be the theory of monoids and  $T$  the theory of groups, and let  $r$  realize every symbol of the language of monoids by its analogue in the language of groups. Then the syntactic translation maps an expression in the language of monoids to the corresponding expression in the language of groups. And the semantic translation maps every group to itself seen as a monoid.

A language that features realizations may or may not provide concrete syntax for these two translations. A language that can talk about both translations may also state the duality between them as an adjunction between two functors in the sense of category theory.

Finally, we have the notion of **subtyping** between modules. If every expression over  $S$  is also an expression over  $T$ , then  $S$  is a **syntactic subtype** of  $T$ . Dually, if every realization of  $S$  is also a realization of  $T$ , then  $S$  is a **semantic subtype** of  $T$ . If both subtyping relations are present in a language, then they are usually opposites of each other.

More concretely,  $S$  is a syntactic subtype of  $T$  iff there is a realization  $r$  of  $S$  in terms of  $T$  whose syntactic and semantic translations are inclusions. Then we

speak of **nominal subtyping** if  $r$  is an import, and of **structural subtyping** if  $r$  is a view.

#### 2.4. Semantics

There are two ways to give a formal semantics of modular expressions. We speak of a **model theoretical** semantics if models are used to interpret modules. This is typical in the algebraic specification community. We speak of a **proof theoretical** semantics if the semantics is given by typing judgments and inference rules.

Often for some or all modular expressions, there is an expression of the base language with the same semantics. In the type and proof theory community, this is often built-in: The semantics of a modular expression is defined by transforming it into a non-modular one; this is called **elaboration**. In contrast, in languages with a model theoretical semantics, it is a theorem about the semantics and often called **flattening**.

We say that a module system is **conservative** if every modular expression can be flattened or elaborated into an expression of the base language. Language features that typically prevent conservativity are higher-order functors and hiding.

Similar to conservativity is the **internalization** of a module system. For certain languages, it is usually possible to represent the module level judgment  $s$  as a realization of the module  $S$  as a typing judgment of the base language. This is possible if the base language features record types, in which all declarations that can occur in a module may also occur as fields in a record. Then modules are records, realizations are values, and functors are functions. However, such expressive record types are often not present and can often only be added at great cost, e.g., an internalized module system for simple type theory requires type polymorphism. Moreover, in languages where declarations build on each other, dependent record types are needed.

#### 2.5. Genericity

A **logical framework** is a formal representation system that provides an uncommitted set of primitives. Such a framework can be used as a meta-language to define other languages. We call a module system **generic** if it is not specific to a certain base language, but defined within a logical framework. A generic module system is parametrized by an arbitrary base language defined within the logical framework.

We distinguish further whether the logical framework is based on **set theory** or **type theory**. The former typically has a model theoretical, the latter a proof theoretical semantics. The choice of framework often implies a foundational commitment because the framework must make some assumptions about the base language.

For example, set/model theoretical module systems may assume the semantics of the base language as an institution. An example is ASL based on the framework of institutions [GB92, SW83]. This implies a commitment to a certain axiomatic set theory in which models and institutions are given. But for

example, if the foundation includes axioms for choice or large cardinals, the models of the same module differ.

Similarly, a type/proof theoretical module system may assume the semantics of the base language as a system of judgments and inference rules. An example is the locale module system based on the logical framework Isabelle [Pau94, KWP99]. This implies a commitment to a formal language in which judgments and inference rules are described. But different logical frameworks permit the representation of different object logics.

We use the term **foundation** to refer to the mathematical theory that formalizes this implicit commitment: the axiomatic set theory in the former, and the logical framework in the latter case. We call a module system **foundation-independent** if it avoids such a commitment. This can for instance be achieved by explicitly representing the foundation itself as a module. Foundation-independent module systems are not only parametric in the base language but also in the foundation used to express the semantics of the base language.

### 2.6. Degree of Formality

Mathematics has traditionally been written in natural language with interspersed formulas. This is different from the fully formal style that is often used in computer-supported mathematics. Even though the focus of MMT is on formal languages, it is worthwhile to discuss informal languages as well because many aspects of module systems are independent of the degree of formality.

**Formal** languages are based on a formal syntax with a precisely defined semantics. The syntax is based on a formal grammar that can be implemented so that computers can parse and understand it. A typical service that a computer can offer for a formal language is the **validation** of knowledge to guarantee correctness. Computers can also automatically generate knowledge, such as in automated theorem proving where the generated knowledge item is a proof. This category also includes controlled grammars of natural language that are used to give formal representations a more human-friendly appearance.

**Informal** languages do not have a formal syntax and are based on unrestricted natural language. While mathematicians use informal language rigorously to obtain an unambiguous semantics, this semantics can only be understood by humans but not by machines. Therefore, only shallow machine-processing services are available such as authoring, storing, and distributing papers and books.

But mathematicians frequently use formal objects within natural language. This has motivated the design of **semi-formal** representation languages that combine formal and informal representations and degrade gracefully when the latter is used. The automated type-setting provided by L<sup>A</sup>T<sub>E</sub>X is a simple example; here the formal representation aspects include the structuring of text into, e.g., definitions, theorems, and formulas.

Note that in the example of L<sup>A</sup>T<sub>E</sub>X, the formulas themselves are not formal in our sense: While formal symbols are used, the representation is still human-oriented, and machines can usually not determine the syntax tree of a formula

from its  $\text{\LaTeX}$  representation. Such representations are called **presentation-based** and distinguished from **content-based** representations that make the syntax tree accessible to machines.

### 2.7. Scalability

For machine-processable representation languages, performance and language design are not always orthogonal. We are specifically interested in language aspects that affect scalability.

We call a module system **web standard-compliant** if it provides a concrete syntax that uses XML [W3C98] for all language expressions and URIs [BLFM05] for all identifiers. XML enables standardized document fragment access by technologies like XPath [W3C99] and document fragment aggregation by XQuery [W3C07]. Deployment on web servers allows distributed storage and flexible access methods. URIs provide a standardized and flexible language for logical identifiers. They support the unambiguous identification of all meaningful components of modular theories and provide an abstraction layer over physical locations. An **XML catalog** can translate URIs into their physical locations represented as URLs.

A common feature in implementations of formal languages is a distinction between **internal** and **external** syntax. The latter is more relaxed in order to ease reading and writing for humans, whereas the former is stricter and fully disambiguated to ease machine-processing. A **reconstruction** algorithm is used to obtain the internal representation from the external one. For programming languages, this is usually called **compilation**. Typical steps of the reconstruction algorithm are parsing of infix operators using precedences, disambiguation of overloaded symbol names, inference of omitted types, and automated proof search to discharge incurred proof obligations. Moreover, often the internal syntax is non-modular, and the reconstruction includes the elaboration or flattening.

If different systems are to communicate mathematical knowledge, a complex reconstruction algorithm can be problematic. If internal syntax is communicated, human-oriented information is lost; when external syntax is communicated, the receiving system must implement the costly reconstruction. Therefore, we speak of **authoring-oriented** languages if the reconstruction algorithm is complex and of **interchange-oriented** languages if it is simple (or even the identity).

We speak of **incremental** processing if modular expressions can be processed step-wise. We say that a language is **decomposable** if there is an algorithm that decomposes a modular declaration into a sequence of **atomic** declarations with an acyclic dependency relation. We say that a language is **order-invariant** if the semantics is independent of the order of declarations as long as the order respects the dependency relation. Any decomposable, order-invariant language permits streaming of documents and optimized storage in databases.

The flattening (elaboration) operation is usually defined by induction on expressions and leads to an exponential increase in size. We speak of **eager**

flattening if every induction step requires the recursive flattening of all sub-expressions. If we regard flattening as the evaluation of a modular expression, this corresponds to call-by-value evaluation. We speak of **lazy** flattening if a corresponding call-by-reference evaluation is possible. In the latter case, the exponential blow-up may be avoided.

### 3. Central Features of MMT

We will now discuss the central design goals that have guided the development of MMT in terms of the concepts introduced above. For other systems with different applications and design choices see Sect. 10.

*A Generic Formal Module System.* MMT is a generic, formal module system for mathematical knowledge. It is designed to be applicable to a large collection of declarative formal base languages, and all MMT notions are fully abstract in the choice of base language.

MMT is designed to be applicable to all base languages based on **theories**. Theories are modules in the sense of Sect. 2, in the simplest case they are defined by a set of typed **symbols** (the signature) and a set of **axioms** describing the properties of the symbols. A **signature morphism**  $\sigma$  from a theory  $S$  to a theory  $T$  translates or interprets the symbols of  $S$  in  $T$ .

If we have entailment relations for the formulas of  $S$  and  $T$ , a signature morphism is particularly interesting if it translates all theorems of  $S$  to theorems of  $T$ ; this is called a **theory morphism**. Using the **Curry-Howard representation**, MMT drops the distinction between symbols and axioms and between signatures and theories altogether, and only uses theories. Axioms are constants whose type is the asserted proposition, and theorem are defined constants whose definiens is a proof.

The flat fragment of MMT provides a generic syntax for theories and theory morphisms (called *views* in MMT). A view from  $S$  to  $T$  is a list of **assignments**  $c \mapsto \omega$  where  $c$  is an  $S$ -constant (axiom) and  $\omega$  is a  $T$ -term (proof). Such a list of assignments induces a **homomorphic** translation of  $S$ -terms to  $T$ -terms by replacing every  $c$  with the corresponding  $\omega$ . Such translations are often called *structural*, *recursive*, or *compositional*.

Full MMT adds the most general form of inheritance: interspersed named imports (called *structures* in MMT) carrying free, explicit, and partial instantiations. In particular, we choose named imports to avoid the problems caused by the diamond situation and import name clashes, which occur frequently in large-scale developments.

MMT has been designed in the tradition of the semi-formal OMDoc language, and an extension of MMT to cover informal knowledge is poised to culminate in a successor to OMDOC. But in this paper, we will focus on the formal aspects only. We will nonetheless discuss the relation to semi-formal languages below. To ensure machine-processing MMT uses a content-oriented representation building on OPENMATH [BCC<sup>+</sup>04] and akin to OMDoc [Koh06]. We have designed and implemented an extension of MMT with notation definitions

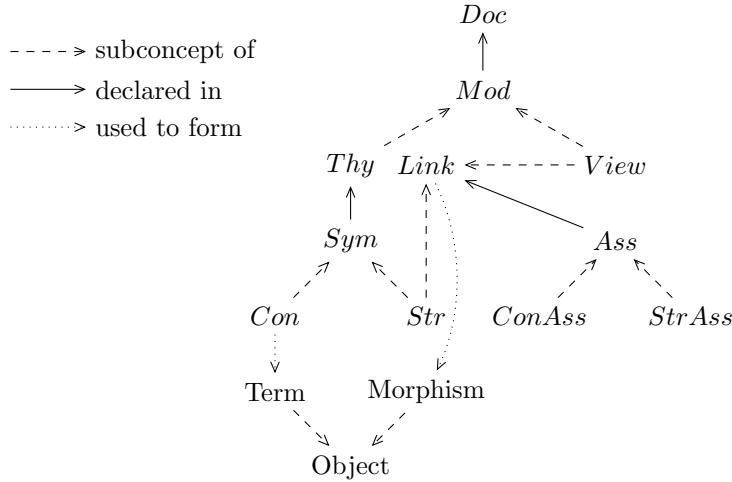


Figure 1: MMT Ontology

that transform MMT-content representations into presentation-oriented formats [Rab08b], but this will not be the focus of this work.

*A Simple Ontology.* A scalable module system must be both expressive and simple, which forms a difficult trade-off. Therefore, MMT carefully picks only a few primitive language features: The ontology of MMT language features is so simple that it can be visualized in a single graph, see Fig. 1. MMT concepts are distinguished into four levels: the document, module, symbol, and object level.

Expressions at the document level are the **documents**, which act as packages. MMT systematically follows the intuition that documents are transparent to the semantics. Therefore, scalable knowledge management services can be implemented easily at the document level. Documents are open packages – every document may refer to every other document as long as the dependency relation is acyclic – and the distribution of modules into documents is transparent. Logical identifiers are used for all knowledge items and are given as MMT URIs, and the translation of URIs into URLs is relegated to an extra-linguistic catalog; thus, MMT documents provide namespace management and abstract from physical locations.

Documents contain **modules**, and MMT uses only two kinds of module declarations: **theories** and **views**. MMT does not need other module declarations because both grounded realizations and functors can be represented as views. Most declarative languages, can be stated naturally as a category. The objects are sets of declarations and are represented as MMT theories. And the morphisms are translations between theories, which are represented as MMT views.

More precisely, MMT theories contain **symbol declarations**, and views

contain **symbol assignments**. A view from theory  $S$  to theory  $T$  must realize all  $S$ -symbols in terms of  $T$ -objects. Consequently, for every kind of symbol declaration, there is a corresponding kind of objects. MMT uses only two kinds of symbol declarations: **Constants** represent all declarations of the base language, and **structures** represent inheritance between theories (see below). A constant assignment provides a  $T$ -term for an  $S$ -constant, and structure assignments provide a  $T$ -morphism for an  $S$ -structure.

Objects are complex expressions that represent mathematical expressions, formulas, etc. MMT only uses two kinds of objects: terms and morphisms. Constants occur as the atomic terms, and structures and views as the atomic morphisms. The grammar for **terms** is motivated by the OPENMATH grammar [BCC<sup>+</sup>04]. It uses generic constructs for application and binding to form complex terms in a way that is general enough to represent most mathematical languages. MMT achieves this by relegating the semantics of terms to a foundation (see below).

**Morphisms** from  $S$  to  $T$  are realizations of  $S$  over  $T$ . We take the concept of **links** from development graphs [AHMS99] to unify the two atomic morphisms: Structures are morphisms induced by imports, views are morphisms declared (and proved) explicitly. Complex morphisms are formed by composition. The representation of realizations as morphisms has the advantage that MMT can easily provide concrete syntax for the two translations induced by a realization: The syntactic translation is given by applying morphisms to terms, and the semantic translation by composition of morphisms. Thus, MMT can capture the semantics of realizations while being parametric in the semantics of terms.

*A Simple Semantics using Theory Graphs.* The semantics of a collection of MMT documents is given as a **theory graph**, which serves as a compact specification of a collection of mathematical theories and their relations. The nodes of a theory graph are the theories; the edges are the links. Each path in a theory graph yields a theory morphism. In particular, if a declarative language is given as a category whose components are represented as MMT theories and morphisms, then diagrams in that category are represented as MMT theory graphs. It is a crucial observation that theory graphs are universal in the sense that they arise naturally and in the same way in any declarative language. Using theory graphs, MMT can capture the semantics of modular theories generically.

*Example 1* (Running Example: Elementary Algebra) For a simple example, consider the theory graph on the right with nodes for the theories of monoids, commutative groups, and rings, and three structures between them. The theory **Monoid** might declare symbols for composition and unit, and axioms for associativity and neutrality. The theory of commutative groups is an extension of the theory of monoids: it arises by adding symbols and axioms to **Monoid**. Therefore, we only need to represent those added symbols and axioms in **CGroup** and add a structure **mon** importing from **Monoid**.

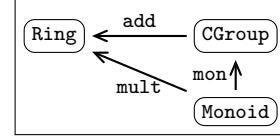


Fig. 2 gives a more detailed view of the theory graph adding the symbols in the theory nodes, but eliding the axioms. `Ring` declares two structures for addition and multiplication, and the distinguish-semantics yields two different monoid operations for addition and multiplication.

Our running example shows a slight complication in the case of first-order logic: We can declare a symbol for the first-order universe either in `Monoid` or in the theory `FOL`, which we will introduce in Ex. 5. Both choices are justified, and we will assume the latter for the sake of our example.

Structures in MMT are always named and the distinguish-semantics is used in the case of diamonds. Qualified identifiers for the imported constants are formed by concatenating the structure name and the name of the imported symbols. For example, the theory `Ring` from Fig. 2 can access the symbols `add/mon/comp` (addition), `add/mon/unit` (zero), `add/inv` (additive inverse), `mult/comp` (multiplication), and `mult/unit` (one).

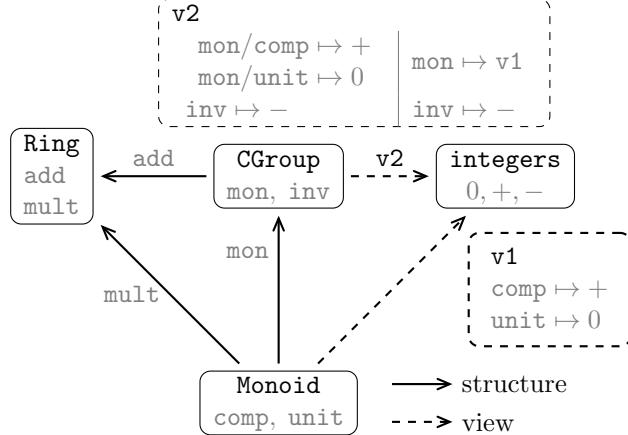


Figure 2: A Theory Graph for Elementary Algebra

Both structures and views from  $S$  to  $T$  are defined by a list of assignments  $\sigma$  that assigns  $T$ -objects to  $S$ -symbols, and both induce theory morphisms from  $S$  to  $T$  that map all  $S$ -objects to  $T$ -objects. This can be utilized to obtain the identify-semantics: sharing declarations are special cases of assignments in structures.

*Example 2 (Sharing via Instantiation)* Consider Ex. 1 but with the change that we declare a symbol `univ` in `Monoid` for the first-order universe. Then `univ` must be shared between the two imports from `Monoid` to `Ring`. We obtain an asymmetric sharing declaration by first declaring the import `add` and then adding the assignment `univ ↦ add/mon/univ` to the structure `mult` in order to identify the two copies of the universe. Alternatively, we can give a symmetric sharing declaration by declaring `univ` in `Ring` as well and adding the assignment `univ ↦ univ` to both structures.

We will see that whole structures can be shared in the same way.

While a view relates two fixed theories without changing either one, structures from  $S$  to  $T$  occur within  $T$  and change  $T$  by including a copy of  $S$ . Thus, structures induce theory morphisms by definition, and views correspond to representation theorems.

*Example 3* (Views (continued from Ex. 1)) The node on the right side of the graph in Fig. 2 represents a theory for the integers declaring the constants 0, +, and -. The fact that the integers are a monoid is represented by the view v1. It is a theory morphism that explicitly gives the interpretations of all symbols: `comp`  $\mapsto +$  and `unit`  $\mapsto 0$ . If we did not omit axioms, this view would also have to interpret all the axioms of `Monoid` as proof terms.

The view v2 is particularly interesting because there are two ways to represent the fact that the integers are a commutative group. In the first variant, all constants of `CGroup` are interpreted separately: `inv` as - and the two imported constants `mon/comp` and `mon/unit` as + and 0, respectively. In the second variant v2 is constructed modularly by importing the existing view v1: The MMT structure assignment `mon`  $\mapsto$  v1 maps all symbol imported by `mon` according to v1. The intuition behind a structure assignment is that it makes the right triangle commute: v2 is defined such that  $v2 \circ \text{mon} = v1$ . Clearly, both variants lead to the same theory morphism; the second one is conceptually more complex but eliminates redundancy because it is structured.

*Partial Morphisms.* The assignments defining a structure may be (and typically are) partial whereas a view should be total. In order to treat structures and views uniformly, we admit **partial views** as well. This is not only possible, but in fact desirable. A typical scenario when working with views is that some of the specific assignments making up the view constitute proof obligations and must be found by costly procedures. Therefore, it is reasonable to represent partial views, namely views where some proof obligations have already been discharged whereas others remain open.

*Example 4* (Partial Morphisms (continued from Ex. 3)) Consider for instance the situation in Fig. 2 but this time taking axioms into account. Recall that under the Curry-Howard correspondence, axioms are just symbols whose types is given by the asserted formula. So we would have additional constants `assoc` and `neut` for associativity and the properties of the neutral element in `Monoid`, the constants `inv_ax` and `comm` for the properties of the inverse element and commutativity in `CGroup`, and finally the constant `dist` for distributivity in `Ring`.

Thus, the views v1 and v2 are clearly partial views, and the missing assignments for `assoc` and `neut` in v1 and for `inv_ax` and `comm` in v2 are proof obligations that need to be discharged by proving the translated axioms in theory `integers`. If these proof terms are known, they can be added to the views as assignments to the respective (axiom) constants. In this situation, the structured view v2 shows its strength: It imports the constant assignments from v1 that discharge proof obligations so that these proofs do not have to be repeated.

Partial morphisms also arise when representations are inherently partial. For example, we can give a one-sided inverse to the structure `mon` in Fig. 2 by mapping `mon/comp` and `mon/unit` to `comp` and `unit`.

MMT introduces **filtering** to obtain a semantics for partial morphisms: All constants for which a view does not provide an assignment are implicitly filtered, i.e., are mapped to a special term  $\top$ . If a link  $l$  from  $S$  to  $T$  filters a  $S$ -constant that has a definiens, this is harmless because the filtered constant can be replaced with its definiens. But if undefined constants are filtered, MMT enforces the strictness of filtering: All terms depending on a filtered constant, are also filtered. In that case, we speak of filtered terms, which are also represented by  $\top$ .

*A Foundation-Independent Semantics.* Mathematical knowledge is described using very different foundations. Most of them can be grouped into set theory and type theory. Within each group there are numerous variants, e.g., Zermelo-Fraenkel [Zer08, Fra22] or Gödel-Bernays set theory [Göd40, Ber37], or set theories with or without the axiom of choice. Therefore, scalability across semantic domains requires a foundation-independent representation language. It is a unique feature of MMT to provide such a high level of genericity and still be able to give a rigorous semantics in terms of theory graphs and a foundation-independent **flattening theorem**.

The semantics of MMT is given proof theoretically by flattening in order to avoid a commitment to a particular model theory. This also makes MMT conservative over the base language so that we can combine MMT with arbitrary base languages without affecting their semantics. Therefore, we have to exclude non-conservative language features, but we have shown in [Rab10, HR11] that despite the proof theoretical semantics of MMT, model theoretical module systems can be represented in MMT. Moreover, we have given an extension of MMT with hiding in [CHK<sup>+</sup>11a].

Foundation-independence is achieved by representing all logics, logical frameworks, and the foundational languages themselves simply as theories. For example, an MMT theory graph based on ZFC set theory starts with a theory that declares the symbols of ZFC such as  $\in$  and  $\subset$ . Moreover, MMT does not prescribe a set of well-typed terms. Instead, MMT uses generic term formation operators, and any term may occur as the type of any other term.

We recover this loss of precision by formalizing the notion of *meta-languages*, which pervades mathematical discourse. Let us write  $M/T$  to express that we work in the object language  $T$  using the meta-language  $M$ . For example, most of mathematics is carried out in FOL/ZFC, i.e., first-order logic is the meta-language, in which set theory is defined. FOL itself might be defined in a logical framework such as LF [HHP93], and within ZFC, we can define the language of natural numbers, which yields LF/FOL/ZFC/Nat. In MMT, all of these languages are represented as theories. In many ways  $M/T$  behaves like an import from  $M$  to  $T$ , but using only an import would fail to describe the meta-relationship. Therefore, MMT uses a binary **meta-theory** relation between theories.

In the example in Fig 3 and generally in this paper, the meta-theory relation is visualized using dotted inclusion morphisms. The theory **FOL** for first-order logic is the meta-theory for **Monoid** and **Ring**. And the theory **LF** for the logical framework **LF** is the meta-theory of **FOL** and the theory **HOL** for higher-order logic. Note how the meta-theory can indicate both to humans and to machines how  $T$  is to be interpreted. For example, interpretations of **Monoid** are always stated relative to a fixed interpretation of **FOL**.

The importance of meta-theories  $M/T$  in MMT is that  $M$  defines the semantics of  $T$ . More precisely, a **foundational theory** declares all primitive concepts and axioms of the foundational language and occurs as the upper-most meta-theory – like **LF** and **Isabelle** in the example in Fig 3. The semantics of the foundational theory is called the **foundation**; it is given externally and assumed by MMT, and it induces the semantics of all other theories. Formally, MMT assumes that the foundation for the foundational theory  $M$  defines typing and equality judgments for arbitrary theories  $T$  with (possibly indirect) meta-theory  $M$ .

The choice of typing and equality is motivated by their universal importance in the formal languages of mathematics and computer science. Here we should clarify that, from an MMT perspective, languages like untyped set theory are in fact typed languages, if only coarsely-typed: For example, typical formalizations of set theory at least distinguish types for sets, propositions, and proofs, and a concise definition of axiom schemes naturally leads to a notion of function types.

#### Example 5 (Meta-Theories (continued from Ex. 4))

We can add meta-theories by adding a theory **FOL** for first-order logic, which occurs as the meta-theory of monoids, groups, and rings. In particular, **FOL** declares symbols for the first-order universe and the connectives and quantifiers. We use a theory for ZFC as the meta-theory of the integers. In that case the views  $v1$  and  $v2$  are only meaningful relative

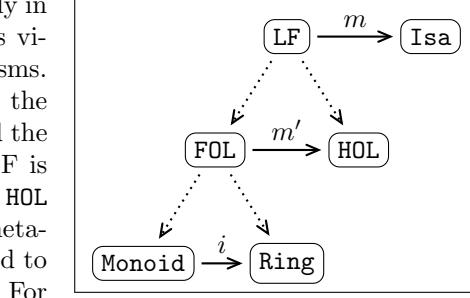
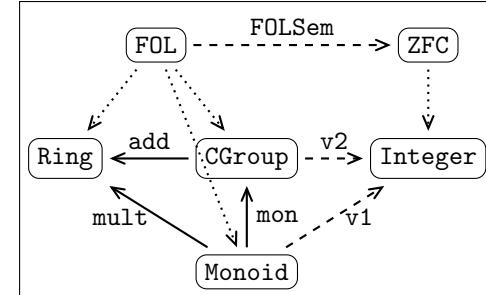


Figure 3: Meta-Theories



to an interpretation of first-order logic in set theory. In MMT, this interpretation is given as a view **FOLSem** from **FOL** to **ZFC** which is attached to  $v1$  and  $v2$  as a meta-morphism. **FOLSem** represents the inductive interpretation function that defines the semantics of first-order logic in set theory.

*Little Logics and Little Foundations.* The little theories methodology [FGT92] strives to state every mathematical theorem in the theory with the smallest possible set of axioms in order to maximize theorem reuse. Using the **foundations-as-theories** approach of, we can extend it to the **little logics** and **little foundations** methodology.

MMT provides a uniform module system for theories, logics, and foundational languages. Thus, we can use structures to represent inheritance at the level of logical foundations and views to represent formal translations between them. For example, the morphisms  $m$  and  $m'$  in Fig. 3 indicate possible translations on the levels of logical frameworks and logics, respectively. Therefore, just like in the little theories approach, we can prove meta-logical results in the simplest logic or foundation that is expressive enough and then use views to move results between foundations.

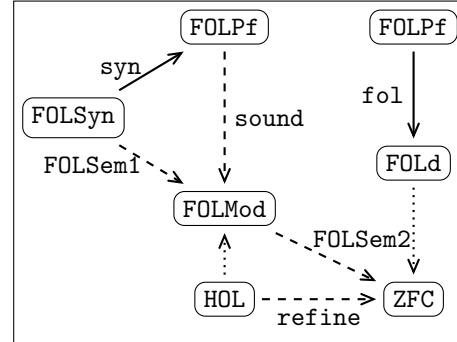
*Example 6* (Proof and Model Theory of First-Order Logic) In [HR11], we formalize the syntax, proof theory, and model theory and prove the soundness of first-order logic in MMT/LF. We use the theory graph given in the commutative diagram on the right. We represent the syntax – i.e., the connectives and quantifiers – in the theory **FOLSyn**. (This theory was called **FOL** in Ex. 5.) For the proof theory, the theory **FOLPf** imports **FOLSyn** and adds constants for the rules of a calculus for first-order logic encoded via the Curry-Howard correspondence.

For the definition of the model theory in **FOLMod**, we should use set theory as the meta-theory. However, doing proofs in set theory as needed for the soundness proof is tedious. Therefore, we use higher-order logic **HOL** as the meta-theory of **FOLMod**; it is expressive enough to carry out the soundness proof but permits typed reasoning. The syntax is interpreted in the model theory by a view **FOLSem1**.

Then the view **refine** from **HOL** to **ZFC** proves that **ZFC** is a refinement of **HOL**. We reuse **refine** to give a morphism **FOLSem2** that interprets **FOLMod** in **ZFC**. The composition of **FOLSem1** and **FOLSem2** yields the view **FOLSem** from Ex. 5.

Finally the soundness proof – which shows that all proof terms over **FOLPf** induce valid statements over **FOLMod** – is represented as the view **sound**. **sound** is given as a structured view: It imports the view **FOLSem1** using the structure assignment **syn**  $\mapsto$  **FOLSem1**.

To establish the views into **ZFC**, it must have proof rules of its own. We use a variant of first-order logic as the meta-theory of **ZFC**, namely **FOLD**. It arises by importing **FOLPf** and then adding a description operator  $\iota$ . This yields two morphisms from **FOLPf** to **ZFC**: one via the import **fol** and the meta-theory relation, and one as the composition of **sound** and **FOLSem2**. These morphisms are not equal: If



$\circ$  is the type of formulas in **FOLsyn**,  
then the former morphism maps  $\circ$  to the type of propositions of ZFC; but the latter maps  $\circ$  to the set of boolean truth values. Note that to express this difference in our commutative diagram, we must use two copies of the node **FOLPf**.

Moreover, in [HR11], all theories have LF as the ultimate meta-theory, which we omitted from the diagram on the right. In addition, [HR11] gives all theories and views using our little logics approach, e.g., using separate theories for each connective. Thus, we can reuse these fragments to define other logics as we do in [KMR09].

*Built-in Web-Scalability.* Most module systems in mathematics and computer science are designed with the implicit assumption that all theories of a graph are retrieved from a single file system or server and are processed by loading them into the working memory of a single process. These assumptions are becoming increasingly unrealistic in the face of the growing size of both mathematical knowledge and formalized mathematical knowledge. Moreover, this mathematical knowledge is represented in different formal languages, which are processed with different implementations.

MMT is designed as a representation language that scales well to large inter-linked document collections that are processed with a wide variety of systems across networks and implementation languages. Therefore, MMT offers integration support through web standards-compliance, incremental processing of large theory graphs, and an interchange-oriented fully disambiguated external syntax.

Scalable transport of MMT documents must be mediated by standardized protocols and formats. While the use of XML as concrete syntax is essentially orthogonal to the language design, the use of **URIs as identifiers** is not because it imposes subtle constraints that can be hard to meet a posteriori. In MMT, all constants, including imported ones, that are available in a theory have canonical URIs. MMT uses tripartite URIs  $doc?mod?sym$  formed from a document URI  $doc$ , a module name  $mod$ , and a qualified symbol name  $sym$ . For example, if the theory graph from Fig. 3 is given in a document with URI <http://cds.omdoc.org/mmt/paper/example>, then the constant **unit** imported from **Monoid** into **CGroup** has the URI <http://cds.omdoc.org/mmt/paper/example?CGroup?mon/unit>.

Note that theories are containers for declarations, and relations between theories define the declarations that are available in a given theory. Therefore, if every available constant has a canonical identifier, the syntax of identifiers is inherently connected to the possible relations between theories. Consequently, and maybe surprisingly, defining the canonical identifiers is almost as difficult as defining the semantics of the whole language.

All MMT definitions and algorithms are designed with incremental processing in mind. In particular, MMT is decomposable and order-invariant. For example, the declaration  $T = \{s_1 : \tau_1, s_2 : \tau_2\}$  of a theory  $T$  with two typed symbols yields the atomic declarations  $T = \{\}, T?s_1 : \tau$ , and  $T?s_2 : \tau_2$ . Documents, views,

and structures are decomposed accordingly. This “unnesting” of declarations is possible because every declaration has a canonical URI so that declarations can be taken out of context for transport and storage and re-assembled later.

The understanding of structures and their induced declarations is crucial to achieve web-scalability. Languages with imports and instantiations tend to be much more complex than flat ones making them harder to specify and implement. Therefore, the semantics of modularity must often remain opaque to generic knowledge management services, an undesirable situation. Because MMT has a simple and foundation-independent flattening semantics, modularity can be made transparent whenever a system is unable to process it.

Moreover, the flattening of MMT is lazy: Every structure declaration can be eliminated individually without recursively flattening the imported theory. Thus, systems gain the flexibility to flatten MMT documents partially and on demand.

## 4. Syntax

We will now develop the abstract syntax of MMT, our formal module system that realizes the features described in the last section. We introduce the syntax in Sect. 4.1. Then we use MMT to give a precise definition of the concept of “realizations” in Sect. 4.2. In Sect 4.3 and 4.4, we introduce auxiliary functions for lookup and normalization that are used to talk about MMT theory graphs.

### 4.1. MMT Theory Graphs

The MMT syntax for theory graphs distinguishes the **module**, **symbol**, and **object** level. We defer the description of the document level to Sect. 8 because documents are by construction transparent to the semantics.

#### 4.1.1. Grammar

The MMT grammar is given in Fig. 4 where  $^+$ ,  $|$ , and  $[-]$  denote non-empty repetition, alternative, and optional parts, respectively. Note that several non-terminal symbols correspond directly to concepts of the MMT ontology given in Sect. 3. In order to state the flattening theorem below, we also introduce the flat MMT syntax; it arises by removing the productions given in gray. We will call a theory graph, module, or definiens **flat**, iff it can be expressed in the flat MMT syntax.

The meta-variables we will use are given in Fig. 5. References to named MMT knowledge items are Latin letters, MMT objects and lists of knowledge items are Greek letters. We will occasionally use  $_$  as an unnamed meta-variable for irrelevant values.

In the following we describe the syntax of MMT and its intended semantics in a bottom-up manner, i.e., identifiers, object level, symbol level, and module level. Alternatively, the following subsections can be read in top-down order.

|                     |              |  |
|---------------------|--------------|--|
| Theory graph        | $\gamma$     | $::= \cdot   \gamma, Thy   \gamma, View$   |
| Theory              | $Thy$        | $::= T \stackrel{[M]}{=} \{\vartheta\}$  |
| View                | $View$       | $::= l : S \rightarrow T \stackrel{[\mu]}{=} \{\sigma\}   l : S \rightarrow T = \mu$   |
| Theory body         | $\vartheta$  | $::= \cdot   \vartheta, Con   \vartheta, Str$  |
| Constant            | $Con$        | $::= c : \omega = \omega   c : \omega   c = \omega   c$                                |
| Structure           | $Str$        | $::= s : S \stackrel{[\mu]}{=} \{\sigma\}   s : S = \mu$                               |
| Link body           | $\sigma$     | $::= \cdot   \sigma, ConAss   \sigma, StrAss$  |
| Ass. to constant    | $ConAss$     | $::= c \mapsto \omega$   |
| Ass. to structure   | $StrAss$     | $::= s \mapsto \mu$  |
| Variable context    | $\Upsilon$   | $::= \cdot   \Upsilon, x[:\omega][=\omega]$  |
| Term                | $\omega$     | $::= T   T?c   x   \omega^\mu   @(\omega, \omega^+)   \beta(\omega; \Upsilon; \omega)$ |
| Morphism            | $\mu$        | $::= id_T   l   \mu \mu$   |
| Document identifier | $g$          | URI, no query, no fragment   |
| Module identifier   | $S, T, M, l$ | $g?I$  |
| Symbol identifier   |              | $T?I$  |
| Local identifier    | $c, s, I$    | $i[/i]^+$  |
| Names               | $i, x$       | pchar <sup>+</sup>   |
| URI, pchar          |              | see RFC 3986 [BLFM05]  |

Figure 4: The Grammar for MMT Expressions

| Level  | Declaration                      | Expression  |
|--------|----------------------------------|---|
| Module | theory $T, S, R, M$<br>link $l$  | theory graph $\gamma$ (set of modules)  |
| Symbol | constant $c$<br>structure $r, s$ | theory body $\vartheta$ (set of symbols)<br>link body $\sigma$ (set of assignments) |
| Object | variable $x$                     | term $\omega$<br>morphism $\mu$   |

Figure 5: Meta-Variables

#### 4.1.2. Identifiers

All MMT identifiers are URIs and the productions for URIs given in RFC 3986 [BLFM05] are part of the MMT grammar. We distinguish identifiers of documents, modules, and symbols.

Document identifiers  $g$  are URIs without queries or fragments (The query and fragment components of a URI are those starting with the special characters ? and #, respectively.).

Module identifiers are formed by pairing a document identifier  $g$  with a local module identifier  $I$  valid in that document. We use ? as a separating character. Similarly, symbol identifiers  $T?c$  arise by pairing a theory identifier with an local identifier valid in that theory.

Local identifiers may be qualified and are thus lists of names separated by /. Finally, names are non-empty strings of pchars. pchar is defined in RFC 3986 and produces any Unicode character where certain reserved characters must be %-encoded; reserved characters are ?/#[]% and all characters generally illegal in URIs.

*Example 7* (Continued from Ex. 1) We assume that the MMT theory graph for the running example is located in a document with some URI  $e$ . Then the MMT URIs of theories and views are for example  $e?\text{Ring}$  and  $e?v1$ . The MMT URIs of the constants available in the theory  $e?\text{Ring}$  are

- $e?\text{Ring}?\text{add}/\text{mon}/\text{comp}$ ,
- $e?\text{Ring}?\text{add}/\text{mon}/\text{unit}$ ,
- $e?\text{Ring}?\text{add}/\text{inv}$ ,
- $e?\text{Ring}?\text{mult}/\text{comp}$ ,
- $e?\text{Ring}?\text{mult}/\text{unit}$ .

The identifiers of structures are special because they may be considered both as symbol level and as module level knowledge items. This is reflected in MMT by giving structures two identifiers. Consider the structure that imports `Monoid` into `CGroup`: If we want to emphasize its nature as a declaration within `CGroup`, we use the symbol identifier  $e?\text{CGroup}?\text{mon}$ ; if we want to emphasize its nature as a morphism, we use the module identifier  $e?\text{CGroup}/\text{mon}$ . Consequently, the non-terminal symbol  $l$  for links may refer both to a view and to a structure (as expected).

#### 4.1.3. The Object Level

Following the OpenMath approach, MMT objects are distinguished into terms and morphisms. **Terms**  $\omega$  are formed from:

- **constants**  $T?c$  referring to constant  $c$  declared in theory  $T$ ,
- **variables**  $x$  declared in an enclosing binder,
- **applications**  $@(\omega, \omega_1, \dots, \omega_n)$  of  $\omega$  to arguments  $\omega_i$ ,
- **bindings**  $\beta(\omega_1; \Upsilon; \omega_2)$  by a binder  $\omega_1$  of a list of variables  $\Upsilon$  with body  $\omega_2$ ,
- **morphism applications**  $\omega^\mu$  of  $\mu$  to  $\omega$ ,
- a **special term**  $\top$  for filtered terms (see below).

Variable contexts are lists of variable declarations. Parallel to constant declarations, variables carry an optional type and an optional definiens. The scope of a bound variable consists of the types and definitions of the succeeding variable declarations and the body of the binder.

For every occurrence of a term, there is a **home theory** against which the term is checked. For occurrences in constant declarations, this is the containing theory. For occurrences in assignments, this is the codomain of the containing link. We call a term  $t$  that is well formed in a theory  $T$  a **term over  $T$** . Terms over  $T$  may use  $T?c$  to refer to a previously declared  $T$ -constant  $c$ . And if  $s$  is a previously declared structure instantiating  $S$ , and  $c$  is a constant declared in  $S$ , then  $T$  may use  $T?s/c$  to refer to the copy of  $c$  induced by  $s$ . Note that MMT assumes that the declarations occur in an order that respects their dependencies; we will see later that the precise order chosen does not matter. MMT does not impose a specific typing relation between terms. In particular, well-formed terms may be untyped or may have multiple types.

*Example 8* (Continued from Ex. 7) The running example only contains constants. Complex terms arise when types and axioms are covered. For example, the type of the inverse in a commutative group is  $@(\rightarrow, \iota, \iota)$ . Here  $\rightarrow$  represents the function type constructor and  $\iota$  the carrier set. These two constants are not declared in the example. Instead, we will add them in Ex. 12 by giving **CGroup** a meta-theory, in which these symbols are declared. A more complicated term is the axiom for left-neutrality of the unit:

$$\omega_e := \beta(\forall; x : \iota; @(=, @((e?\text{Monoid}?\text{comp}, e?\text{Monoid}?\text{unit}, x), x))).$$

Here  $\forall$  and  $=$  are further constants that are inherited from the meta-theory.

**Morphisms** are built up from links and compositions. If  $s$  is a structure declared in  $T$  that imports from  $S$ , then  $T/s$  is a link from  $S$  to  $T$ . Similarly, every view  $m$  from  $S$  to  $T$  is a link. Composition is written  $\mu\mu'$  where  $\mu$  is applied before  $\mu'$ , i.e., composition is in diagrammatic order. The identity morphism of the theory  $T$  is written  $id_T$ . A morphism application  $\omega^\mu$  takes a term  $\omega$  over  $S$  and a morphism  $\mu$  from  $S$  to  $T$ , and returns a term over  $T$ .

Just like a structure declared in  $T$  is both a symbol of  $T$  and a link into  $T$ , a morphism from  $S$  to  $T$  can be regarded as a composed object over  $T$ . To stress this often fruitful perspective, we also call the codomain of a morphism its **home theory**, and the domain its **type**. Then morphism composition  $\mu'\mu$  can be regarded as the application of  $\mu$  to  $\mu'$ : It takes a morphism  $\mu'$  with home theory  $S$  and type  $R$  and returns a morphism with home theory  $T$  of the same type.

*Example 9* (Continued from Ex 8) In the running example, an example morphism is

$$\mu_e := e?\text{CGroup}/\text{mon}\,e?\text{v2}.$$

It has domain  $e?\text{Monoid}$  and codomain  $e?\text{integers}$ . The intended semantics of the term  $\omega_e^{\mu_e}$  is that it yields the result of applying  $\mu_e$  to  $\omega_e$ , i.e.,

$$\beta(\forall; x : \iota; @(=, @((+, 0, x), x))).$$

Here, we assume  $\mu_e$  has no effect on those constants that are inherited from the meta-theory. We will make that more precise below by using the identity as a meta-morphism.

We define a straightforward abbreviation for the application of morphisms to whole contexts:

**Definition 10.** We define  $\Upsilon^\mu$  by

$$\cdot^\mu := \cdot \quad \text{and} \quad (\Upsilon, x : \tau = \delta)^\mu := \Upsilon^\mu, x : \tau^\mu = \delta^\mu$$

Here we assume  $\perp^\mu = \perp$  to avoid case distinctions.

The analogy between terms and morphisms is summarized in Fig. 6.

|           | Atomic object | Complex object | Type   | Checked relative to |
|-----------|---------------|----------------|--------|---------------------|
| Terms     | constant      | term           | term   | home theory         |
| Morphisms | link          | morphism       | domain | codomain            |

Figure 6: The Object Level

#### 4.1.4. The Symbol Level

We distinguish four symbol level concepts as given in Fig. 7: constants and structures, and assignments to them.

|           | Declaration          | Assignment                       |
|-----------|----------------------|----------------------------------|
| Terms     | of a constant $Con$  | to a constant $c \mapsto \omega$ |
| Morphisms | of a structure $Str$ | to a structure $s \mapsto \mu$   |

Figure 7: The Symbol Level

A **constant declaration** of the form  $c : \tau = \delta$  declares a constant  $c$  of type  $\tau$  with definition  $\delta$ . Both the type and the definition are optional yielding four kinds of constant declarations. If both are given, then  $\delta$  must have type  $\tau$ . In order to unify these four kinds, we will sometimes write  $\perp$  for an omitted type or definition.

Recall that via the Curry-Howard representation, a theorem can be declared as a constant with the asserted proposition as the type and the proof as the definiens. Similarly, (derived) inference rules are declared as (defined) constants.

A **structure declaration** of the form  $s : S \stackrel{[\mu]}{=} \{\sigma\}$  in a theory  $T$  declares a structure  $s$  instantiating the theory  $S$  defined by assignments  $\sigma$ . Such structures can have an optional meta-morphism  $\mu$  (see below). Alternatively, structures may be introduced as an abbreviation for an existing morphism:  $s : S = \mu$ . While the domain of a structure is given explicitly (in the style of a type), the codomain is the theory in which the structure is declared. Consequently, if  $s : S = \mu$  is declared in  $T$ ,  $\mu$  must be a morphism from  $S$  to  $T$ .

Just like symbols are the constituents of theory bodies, assignments are the constituents of link bodies. Let  $l$  be a link from  $S$  to  $T$ . A **assignment to a constant** of the form  $c \mapsto \omega$  in the body of  $l$  expresses that  $l$  maps the constant

$c$  of  $S$  to the term  $\omega$  over  $T$ . Assignments of the form  $c \mapsto \top$  are special: They express that the constant  $c$  is **filtered**, i.e.,  $l$  is undefined for  $c$ .

If  $s$  is a structure declared in  $S$  and  $\mu$  a morphism over (i.e., into)  $T$ , then an **assignment to a structure** of the form  $s \mapsto \mu$  expresses that  $l$  maps  $s$  to  $\mu$ . This means that the triangle  $S/s \xrightarrow{l} l = \mu$  commutes.

Both kinds of assignments must type-check to ensure that typing is preserved by theory morphisms. In the case of constants, this means that the term  $\omega$  must type-check against  $\tau^l$  where  $\tau$  is the type of  $c$  declared in  $S$ . In the case of structures, it means that  $\mu$  must be a morphism from  $R$  to  $T$  where  $R$  is the type, i.e., the domain, of  $s$ .

*Induced Symbols.* Intuitively, the semantics of a structure  $s$  with domain  $S$  declared in  $T$  is that all symbols of  $S$  are copied into  $T$ . For example, if  $S$  contains a constant  $c$ , then an induced constant  $s/c$  is available in  $T$ . In other words,  $/$  is used as the operator that dereferences structures.

Similarly, every assignment to a structure induces assignments to constants. Continuing the above example, if a link with domain  $T$  contains an assignment to  $s$ , this induces assignments to the induced constants  $s/c$ . Furthermore, assignments may be **deep** in the following sense: If  $c$  is a constant of  $S$ , a link with domain  $T$  may also contain assignments to the induced constant  $s/c$ . Of course, this can lead to clashes if a link contains assignments for both  $s$  and  $s/c$ ; links with such clashes will not be well-formed.

*Example 11* (Continued from Ex. 9) The symbol declarations in the theory **CGroup** are written formally like this:

$$\text{mon} : e?\text{Monoid} = \{\} \quad \text{and} \quad \text{inv} : @(\rightarrow, \iota, \iota).$$

The former induces the constants  $e?\text{CGroup?mon/comp}$  and  $e?\text{CGroup?mon/unit}$ . **Ring** contains only the two structures

$$\text{add} : e?\text{CGroup} = \{\} \quad \text{and} \quad \text{mult} : e?\text{Monoid} = \{\}.$$

Instead of inheriting a symbol  $\iota$  for the first-order universe from the metatheory, we can declare a symbol **univ** in **Monoid**. Then **Ring** would inherit two instances of **univ**, which must be shared. **Ring** would contain the two structures

$$\begin{aligned} \text{add} &: e?\text{CGroup} = \{\} \\ \text{mult} &: e?\text{Monoid} = \{\text{mon/univ} \mapsto e?\text{Ring?add/mon/univ}\} \end{aligned}$$

Using an assignment to a structure, the assignments of the view **v2** look like this:

$$\text{inv} \mapsto e?\text{integers?} - \quad \text{and} \quad \text{mon} \mapsto e?\text{v1}.$$

The latter induces assignments for the induced constants  $e?\text{CGroup?mon/comp}$  as well as  $e?\text{CGroup?mon/unit}$ . For example,  $e?\text{CGroup?mon/comp}$  is mapped to  $e?\text{Monoid?comp}^{e?\text{v1}}$ .

The alternative formulation of the view `v2` arises if two deep assignments to the induced constants are used instead of the assignment to the structure `mon`:

$$\text{mon/comp} \mapsto e?\text{integers?} + \quad \text{and} \quad \text{mon/unit} \mapsto e?\text{integers?}0$$

#### 4.1.5. The Module Level

On the module level a **theory declaration** of the form  $T \stackrel{[M]}{=} \{\vartheta\}$  declares a theory  $T$  defined by a list of symbol declarations  $\vartheta$ , which we call the **body** of  $T$ . Theories have an optional meta-theory  $M$ . A **View declarations** of the form  $m : S \rightarrow T \stackrel{[\mu]}{=} \{\sigma\}$  declares a view  $m$  from  $S$  to  $T$  defined by a list of assignments  $\sigma$  and by an optional meta-morphism  $\mu$ . Just like structures, views may also be defined by an existing morphism:  $m : S \rightarrow T = \mu$ .

*Meta-Theories.* Above, we have already mentioned that theories may have meta-theories and that links may have meta-morphisms. Meta-theories provide a second dimension in the theory graph. If  $M$  is the meta-theory of  $T$ , then  $T$  may use all symbols of  $M$ .  $M$  provides the syntactic material that  $T$  can use to define the semantics of its symbols.

Because a theory  $S$  with meta-theory  $M$  implicitly imports all symbols of  $M$ , a link from  $S$  to  $T$  must provide assignments for these symbols as well. This is the role of the meta-morphism: Every link from  $S$  to  $T$  must provide a meta-morphism from  $M$  to  $T$  (or any meta-theory of  $T$ ).

*Example 12* (Continued from Ex. 11) We can now combine the situations from Ex. 5 and 6 in one big MMT theory graph. In a document with URI  $m$ , we declare an MMT theory for the logical framework as

$$m?\text{LF} = \{\text{type}, \rightarrow, \dots\}$$

where we only list the constants that are relevant for our running example: `type` represents the kind of types, and  $\rightarrow$  is the function type constructor.

We declare a theory for first-order logic in a document with URI  $f$  like this:

$$f?\text{FOLSyn} \stackrel{m?\text{LF}}{=} \left\{ \begin{array}{l} \iota : m?\text{LF?type}, o : m?\text{LF?type}, \\ \text{equal} : @(\text{m?LF}\rightarrow, ??\iota, ??\iota, ??o), \dots \end{array} \right\}$$

Here we already use relative identifiers (see Sect. 8.3) in order to keep the notation readable: Every identifier of the form  $??c$  is relative to the enclosing theory: For example,  $??\iota$  resolves to  $f?\text{FOLSyn}?\iota$ . Again we restrict ourselves to a few constant declarations: The types  $\iota$  and  $o$  represent terms and formulas, and the equality operation takes two terms and returns a formula.

Then the theories `Monoid`, `CGroup`, and `Ring` are declared using  $f?\text{FOLSyn}$  as their meta-theory. For example, the declaration of the theory `CGroup` finally looks like this:

$$e?\text{CGroup} \stackrel{f?\text{FOLSyn}}{=} \left\{ \begin{array}{l} \text{mon} : e?\text{Monoid} \stackrel{id_{f?\text{FOLSyn}}}{=} \{\}, \\ \text{inv} : @(\text{m?LF}\rightarrow, f?\text{FOLSyn}?\iota, f?\text{FOLSyn}?\iota) \end{array} \right\}$$

Here the structure `mon` must have a meta-morphism translating from the meta-theory of `Monoid` to the current theory, and that is simply the identity morphism of  $f?\text{FOLSyn}$  because `Monoid` and  $e?\text{CGroup}$  have the same meta-theory. If the meta-theory of `integers` is ZFC, then the meta-morphism of `v1` and `v2` is `FOLSem`.

#### 4.2. Realizations

MMT permits an elegant and precise formulation of a general theory of realizations, which formalizes the intuitions we introduced in Sect. 2.3. The central idea is to formalize the implicit global environment as an MMT theory  $D$ . In particular, MMT naturally provides concrete syntax for both the syntactic and the semantic translations associated with views and functors.

We will consider examples from programming languages and logic. In the former case, we use SML and  $D$  is a theory for the global environment of SML. In the latter case,  $D$  is a theory for ZFC set theory.

**Definition 13** (Grounded Realizations). An MMT-theory with meta-theory  $D$  is called a “ $D$ -theory”. Then a **grounded realization** of the  $D$ -theory  $S$  is a morphism from  $S$  to  $D$  that is the identity on  $D$ .

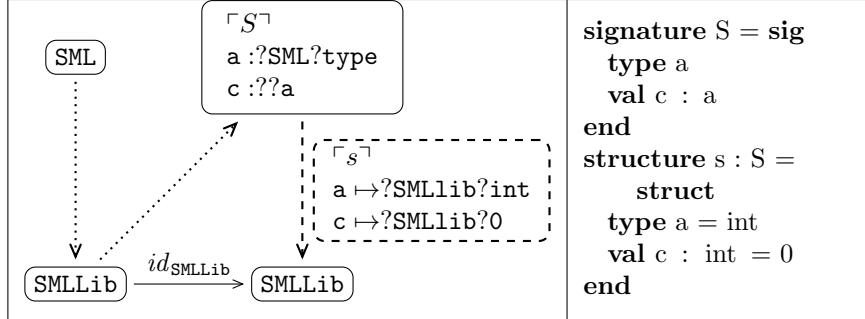


Figure 8: Implementations in SML

**Example 14** (Realizations in SML) The theory `SML` contains declarations for all primitives of the simple type theory underlying SML, such as `->`, `fn`, and `type`. These constants are untyped and undefined. `SML` is used as the meta-theory of the theory  $D = \text{SMLLib}$ , which extends `SML` with typed constants for all declarations of the SML basis library [SML97].

Now we represent SML signatures  $S$  as `SMLib`-specifications and SML structures  $s$  realizing  $S$  as grounded realizations of  $S$ . For example, consider the simple SML signature  $S$  and the structure  $s$  realizing it given on the right of Fig. 8. Its representation in MMT is given by the commutative theory graph on the left side of the same figure.  $\lceil S \rceil$  contains one MMT constant declaration for every declaration in  $S$ . These constants have a type according to  $S$  but no definiens. The view  $\lceil s \rceil$  maps every declaration of  $\lceil S \rceil$  to its value given by  $s$ .

More generally, SML structures  $s$  may also contain declarations that do not correspond to declarations present in  $S$ . In that case, an auxiliary theory  $T$  with meta-theory **SMllib** is used that contains one declaration for every declaration in  $s$ . Then the view  $\lceil s \rceil$  arises as the partial view from  $T$  to **SMllib**.

In Ex. 14, a single meta-theory **SMllib** is used because both SML signatures and SML structures may use the SML basis library. A common alternative is that the specification and the implementation language are separated into two different languages. We encounter this, for example, in logic where specifications (i.e., theories) are written using only the syntax of the logic whereas implementations (i.e., models) are given in terms of the semantic domain – in our example ZFC set theory.

*Example 15 (Realizations in Logic)*

The theory graph on the right continues Ex. 5. The theory of monoids is represented as a FOL-theory **Monoid**. To represent models as grounded realizations, we need a ZFC-theory **MonoidMod**. This theory arises as the pushout of **Monoid** along **FOLSem** over **FOL**. In MMT, this pushout can be expressed easily:

$$\text{MonoidMod} \xrightarrow{\text{ZFC}} \left\{ \text{mon} : \text{Monoid} \xrightarrow{\text{FOLSem}} \{ \} \right\}$$

Thus, **MonoidMod** declares the same local symbols as **Monoid** but translated along **FOLSem**.

Then we can represent models  $M$ , i.e., monoids, as grounded realizations  $\lceil M \rceil$  of **MonoidMod**. Indeed, a monoid  $M$  provides one value for every declaration of **Monoid**, just like an MMT-morphism.

So far, we have declared the universe in **FOL**, which means that we actually need a family of views **FOLSem**( $U$ ), each of which interprets the universe as the set  $U$ . If we declare the universe in **Monoid** (rather than in **FOL**), this example becomes more intuitive. Then **Monoid** and thus also **MonoidMod** have constant declarations for the names **univ**, **comp**, **unit**, **assoc**, and **neut**. Consequently, a monoid  $M = (U, \circ, e)$  is encoded as the view  $\lceil M \rceil$  that contains assignments  $\text{univ} \mapsto \lceil U \rceil$ ,  $\text{comp} \mapsto \lceil \circ \rceil$ ,  $\text{unit} \mapsto \lceil e \rceil$ ,  $\text{assoc} \mapsto P$ , and  $\text{neut} \mapsto Q$ . Here  $\lceil U \rceil$ ,  $\lceil \circ \rceil$ , and  $\lceil e \rceil$  are the MMT-terms over **ZFC** that represent the objects  $U$ ,  $\circ$ , and  $e$ . Moreover,  $P$  and  $Q$  are the terms representing the necessary proofs that show that  $M$  is indeed a monoid.

As we will see in Sect. 5, MMT guarantees that  $\lceil s \rceil$  and  $\lceil M \rceil$  preserve typing. Thus, the properties of implementing a specification and modeling a theory are captured naturally by the properties of MMT theory morphisms.

**Definition 16 (Functors).** Given two  $D$ -theories  $S$  and  $T$ , a **functor** from  $S$  to  $T$  is a morphism from  $T$  to  $S$  that is the identity on  $D$ . Given such a functor

$f$  and a grounded realization  $r$  of  $S$ , the **functor application** is defined as the grounded realization  $fr$ .

It is often convenient to give such a functor as a triple  $(B, i, o)$  as in the diagram on the right. Here the body of the theory  $B$  consists of a structure declaration  $i : S \stackrel{id_D}{\equiv} \{\}$  followed by arbitrary constant declarations all of which have a definiens. The intuition is that  $B$  imports its input theory  $S$  and then implements the intended output theory  $T$ ; the view  $o$  determines how  $T$  is implemented by  $B$ .

Let  $i^{-1}$  denote the view from  $B$  to  $S$ , which inverts  $i$ , i.e., it maps every constant induced by the structure  $i$  to the corresponding constant of  $S$ . Because all local constant declarations of  $B$  have a definiens,  $i^{-1}$  is total. Then we obtain the intended functor as the composition  $f = o i^{-1}$ . Given a grounded realization  $r$  of  $S$ , functor application is simply composition.

Note that we are flexible whether the intelligence of the functor is given in  $B$  or in  $o$ .  $B$  may contain defined constants for all declarations of  $T$  already so that  $o$  is just an inclusion. The opposite extreme arises if  $B$  contains no declaration besides  $i$  and the assignments in  $o$  give the body of the functor.

Sometimes it is not desirable to use the view  $i^{-1}$  because applying  $i^{-1}$  to a  $B$ -term involves expanding all the definitions of  $B$ . In that case, we can use structure assignments to represent functor application. Consider a  $D$ -theory  $C$ , which has access to a realization  $r$  of  $S$ , i.e.,  $r$  is a morphism from  $S$  to  $C$ . We wish to apply the functor given by  $(B, i, o)$  to  $r$  in order to obtain a realization of  $T$ , i.e., a morphism from  $T$  to  $C$ . We can do that by using the following structure declaration in  $C$

$$\text{apply} : B \stackrel{id_D}{\equiv} \{i \mapsto r\}$$

Now the composed morphism  $o \text{apply}$  is the result of applying  $(B, i, o)$  to  $r$ .

*Example 17 (SML (continued from Ex. 14))* An SML functor

```
functor f(struct i : S) : T = struct Σ end
```

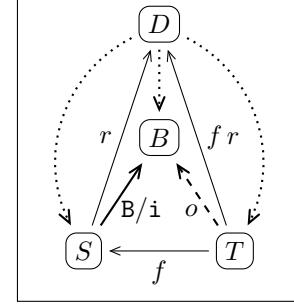
can be represented directly as a triple  $(B, i, o)$  where  $B$  is the theory

$$B \stackrel{id_{\text{SMLib}}}{\equiv} \left\{ i : \vdash S \vdash \stackrel{id_{\text{SMLib}}}{\equiv} \{\}, \vdash \Sigma \vdash \right\}$$

and the view  $o$  from  $\vdash T \vdash$  to  $B$  is an inclusion.

Functors with multiple arguments can be represented by first declaring an auxiliary theory that collects all the arguments of the functor.

*Example 18 (Logic (continued from Ex. 15))*



Consider the functor that maps a monoid  $M = (U, \circ, e)$  to its group of units (whose universe is the set  $\{u \in U \mid \exists v \in U. u \circ v = v \circ u = e\}$ ). We represent it as a triple  $(\text{?UnitGroup}, \text{mon}, o)$  as in the diagram on the right. We assume that all involved modules are declared in the same document so that we can use relative identifiers (see Sect. 8.3) and declare

$$\begin{aligned} \text{UnitGroup} &\stackrel{\text{?ZFC}}{\equiv} \left\{ i : \text{?MonoidMod} \stackrel{\text{?FOLSem}}{\equiv} \{\} \right\} \\ o &: \text{?GroupMod} \rightarrow \text{?UnitGroup} \stackrel{\text{?FOLSem}}{\equiv} \{\sigma\} \end{aligned}$$

Here  $\sigma$  contains the assignments that realize a group in terms of a set theory and an assumed monoid  $\text{mon}$ . For example,  $\sigma$  contains an assignment

$$\text{univ} \mapsto @(\text{C}, \text{?UnitGroup}?i/\text{univ}, I)$$

where we assume that  $C$  is defined in ZFC such that  $@(C, s, p)$  represents the set  $\{x \in s \mid p(x)\}$ , and we use  $I$  to represent the property of having an inverse element.

We can strengthen the above representations considerably by using an additional meta-theory: A foundational theory for a logical framework that occurs as the meta-theory of  $D$ . For example, we can use LF as the meta-theory of SML and ZFC. Then the constants occurring in SML and ZFC can be typed using the type theory of LF.

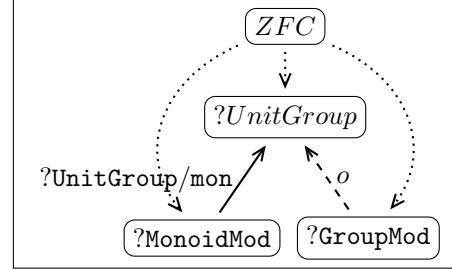
If the semantics of LF is given in terms of typing and equality judgments, then MMT induces a precise semantics of realizations and functors that adequately represents that of, for example, SML and first-order logic. More generally, the type preservation of MMT morphism formalizes the “conforms-to” relation between a specification and an implementation or between a model and a theory.

We follow this approach systematically in [HR11] as indicated in Ex. 5. In [IR11], we show how to formalize other foundations of mathematics. A corresponding representation of the semantics of SML in LF can be found in [LCH07].

#### 4.3. Valid Declarations

In the following we define the valid declarations of a theory graph, which arise by adding all induced symbols and assignments. This corresponds to the flattening semantics of structures that eliminates structures and transforms MMT theory graphs into flat ones.

The judgments for valid declarations are given in Fig. 9. All of them are parametrized by a theory graph  $\gamma$ . The first four judgments are functional in the sense that they take identifiers as input (red) and return declarations (blue) as output. The mutually recursive definitions of all judgments are given below.



| Judgment                             | Intuition: in theory graph $\gamma \dots$                     |
|--------------------------------------|---|
| $\gamma > T = \{\vartheta\}$         | $T$ is a theory in $\gamma$ with body $\vartheta$ .           |
| $\gamma \gg l : S \rightarrow T = B$ | $l$ is a link from $S$ to $T$ with definiens $B$ .            |
| $\gamma >_T c : \tau = \delta$       | $c : \tau = \delta$ is an induced constant of $T$ .           |
| $\gamma \gg_l c \mapsto \delta$      | $c \mapsto \delta$ is an induced constant assignment of $l$ . |
| $M \hookrightarrow T$                | $M$ is the meta-theory of $T$ .                               |
| $\mu \hookrightarrow l$              | $\mu$ is the meta-morphism of $l$ .                           |

Figure 9: Judgments for Valid Declarations

*Valid Modules.* Firstly, the judgments  $\gamma > T = \{\vartheta\}$  and  $\gamma \gg l : S \rightarrow T = B$  define the structure of the MMT theory graph, i.e., the valid module level identifiers. Here  $B$  is of the form  $\{\sigma\}$  or  $\mu$  according to whether  $l$  is defined by a link body or a morphism. Moreover, we write  $M \hookrightarrow T$  and  $\mu \hookrightarrow l$  to give the meta-theory and meta-morphism of a theory  $T$  or a link  $l$ . These judgments are somewhat trivial because they hold iff a meta-theory or meta-morphism is provided explicitly in the syntax of the theory graph.

The first five rules in Fig. 10 are straightforward: They simply cover the declaration of a theory, and the two possible ways each to declare a view or a structure. We use square brackets to denote the optional meta-theories or meta-morphisms, and we give the cases for  $M \hookrightarrow T$  and  $\mu \hookrightarrow l$  as second conclusions of a rule.

The only non-trivial rule is *ind\_str*, which covers the case of induced structures:  $T/s/r$  identifies the structure induced when a structure declaration  $s$  instantiates  $S$  and  $S$  itself has a structure  $r$ . The induced structure is defined to be equal to the composition of the two structures, which formalizes the intended semantics of induced structures.

*Valid Symbols.* For every theory or link of  $\gamma$ , we define the symbol level identifiers valid in it. If  $\gamma > T = \{\_\}$ , we write  $\gamma >_T c : \tau = \delta$  if  $c : \tau = \delta$  is a valid constant declaration of  $T$ . To avoid case distinctions, we write  $\perp$  for  $\tau$  or  $\delta$  if they are omitted. If  $\gamma \gg l : \_\rightarrow \_\_ = \_\_$ , we write  $\gamma \gg_l c \mapsto \delta$  if  $c \mapsto \delta$  is a valid assignment of  $T$ .

The induced constants of a theory are defined by the rules in Fig. 11. The rule *con* simply handles explicit constant declarations. The remaining rules handle induced constants that arise by translating a declaration  $c : \tau = \delta$  along a structure  $T/s$ . In all cases, the type of the induced constant is determined by translating  $\tau$  along  $T/s$ . To avoid case distinctions, we assume  $\perp^{T/s} = \perp$ , i.e., untyped constants induce untyped constants.

But three cases are distinguished to determine the definiens of the induced constant. Firstly, rule *ind\_con\_def* applies if the constant  $c$  already has a definiens  $\delta \neq \perp$ . Then the induced constant has the translation of  $\delta$  along  $T/s$  as its definiens. Otherwise, there are two further cases depending on the assignment provided by the structure  $T/s$  (see the respective rules in Fig. 12). If  $T/s$  provides the default assignment  $T?_s/c$  the induced constant has no definiens

|  |
|--|
| $\frac{T \stackrel{[M]}{\equiv} \{\vartheta\} \text{ in } \gamma}{\gamma > T = \{\vartheta\} \quad [M \hookrightarrow T]} thy$   |
| $\frac{l : S \rightarrow T = \mu \text{ in } \gamma}{\gamma \gg l : S \rightarrow T = \mu} viewdef \qquad \frac{l : S \rightarrow T \stackrel{[\mu]}{\equiv} \{\sigma\} \text{ in } \gamma}{\gamma \gg l : S \rightarrow T = \{\sigma\} \quad [\mu \hookrightarrow l]} view$ |
| $\frac{\gamma > T = \{\vartheta\} \quad s : S = \mu \text{ in } \vartheta}{\gamma \gg T/s : S \rightarrow T = \mu} strdef$   |
| $\frac{\gamma > T = \{\vartheta\} \quad s : S \stackrel{[\mu]}{\equiv} \{\sigma\} \text{ in } \vartheta}{\gamma \gg T/s : S \rightarrow T = \{\sigma\} \quad [\mu \hookrightarrow T/s]} str$   |
| $\frac{\gamma > T = \{\vartheta\} \quad s : S = \_ \text{ in } \vartheta \quad \gamma \gg S/r : R \rightarrow S = \_}{\gamma \gg T/s/r : R \rightarrow T = S/r \ T/s} ind\_str$  |

Figure 10: Valid Modules

(rule *ind-con-dflt*). If  $T/s$  provides an explicit definiens  $\delta$ , it becomes the definiens of the induced constant (rule *ind-con-ass*).

|  |
|--|
| $\frac{\gamma > T = \{\vartheta\} \quad c : \tau = \delta \text{ in } \vartheta}{\gamma >_T c : \tau = \delta} con$  |
| $\frac{\gamma \gg T/s : S \rightarrow T = \_ \quad \gamma >_S c : \tau = \delta \quad \delta \neq \perp}{\gamma >_T s/c : \tau^{T/s} = \delta^{T/s}} ind\_con\_def$          |
| $\frac{\gamma \gg T/s : S \rightarrow T = \_ \quad \gamma >_S c : \tau = \perp \quad \gamma \gg_{T/s} c \mapsto T?s/c}{\gamma >_T s/c : \tau^{T/s} = \perp} ind\_con\_dflt$  |
| $\frac{\gamma \gg T/s : S \rightarrow T = \_ \quad \gamma >_S c : \tau = \perp \quad \gamma \gg_{T/s} c \mapsto \delta}{\gamma >_T s/c : \tau^{T/s} = \delta} ind\_con\_ass$ |

Figure 11: Valid Constants

The induced assignments of a link  $l$  are defined by the rules in Fig. 12. The rule  $\text{def\_link\_ass}$  defines the assignments of a link that is defined as  $\mu$ : Every undefined constant is translated along  $\mu$ .

For links that are defined by a list of assignments, four cases must be distinguished. Firstly, the rule  $\text{ass}$  applies if there is an explicit assignment  $c \mapsto \omega$  in  $l$ . Secondly, rule  $\text{ind\_ass}$  creates induced assignments to  $s/c$ , which arise if there is an assignment of a morphism  $\mu$  to the structure  $r$  in a link  $l$ . Since  $r/c$  identifies the constant  $c$  imported along  $r$ , the induced assignment arises by translating  $c$  along  $\mu$ .

Finally, it is possible that neither rule  $\text{ass}$  nor rule  $\text{ind\_ass}$  applies to  $c$  – namely if the body of  $l$  contains neither an explicit nor an induced assignment for  $c$ . We abbreviate that by “ $c$  not covered by  $\sigma$ ”. In that case the rules  $\text{dflt\_ass\_str}$  and  $\text{dflt\_ass\_view}$  define default assignments depending on whether  $l$  is a structure or a view. If  $l$  is a structure,  $c$  is mapped to the induced constant  $T?r/c$  in rule  $\text{dflt\_ass\_str}$ . If  $l$  is a view,  $c$  is filtered via rule  $\text{dflt\_ass\_view}$ .

|  |   |  |
|--|---|--|
|  | $\frac{\gamma \gg l : S \rightarrow T = \mu \quad \gamma >_S c : \_ = \perp}{\gamma \gg_l c \mapsto (S?c)^\mu} \text{def\_link\_ass}$   |  |
|  | $\frac{\gamma \gg l : S \rightarrow T = \{\sigma\} \quad c \mapsto \omega \text{ in } \sigma}{\gamma \gg_l c \mapsto \omega} \text{ass}$  |  |
|  | $\frac{\gamma \gg l : S \rightarrow T = \{\sigma\} \quad \begin{array}{c} \gamma \gg S/r : R \rightarrow S = \_ \\ \gamma >_S r/c : \_ = \perp \end{array} \quad r \mapsto \mu \text{ in } \sigma}{\gamma \gg_l r/c \mapsto (R?c)^\mu} \text{ind\_ass}$ |  |
|  | $\frac{\gamma \gg l : S \rightarrow T = \{\sigma\} \quad \gamma >_S c : \_ = \perp \quad \begin{array}{c} l \text{ structure} \\ c \text{ not covered by } \sigma \end{array}}{\gamma \gg_l c \mapsto T?r/c} \text{dflt\_ass\_str}$                     |  |
|  | $\frac{\gamma \gg l : S \rightarrow T = \{\sigma\} \quad \gamma >_S c : \_ = \perp \quad \begin{array}{c} l \text{ view} \\ c \text{ not covered by } \sigma \end{array}}{\gamma \gg_l c \mapsto \top} \text{dflt\_ass\_view}$                          |  |

Figure 12: Valid Assignments

*Clash-Free*ness. It is easy to prove that if  $\gamma >_S c : \_ = \perp$  and  $\gamma \gg l : S \rightarrow T = \_$ , then always  $\gamma \gg_l c \mapsto \delta$  for some  $\delta$ , but  $\delta$  is not necessarily unique. More generally, the elaboration judgments do not necessarily define functions from qualified identifiers to induced declarations. For example, a theory graph

might declare the same module name twice or a theory might declare the same symbol name twice. To exclude theory graphs with such name clashes, we use the following definition:

**Definition 19.** A theory graph  $\gamma$  is called *clash-free* if all of the following hold:

- $\gamma$  contains no two module declarations for the names  $i$  and  $j$  such that  $i = j$  or such that  $j$  is of the form  $i/j'$  and the body of  $i$  contains a declaration for the name  $j'$ .
- There is no module in  $\gamma$  whose body contains two declarations for the names  $i$  and  $j$  such that  $i = j$  or  $j$  is of the form  $i/j'$ .

This definition is a bit complicated because it covers theory graphs and theories that explicitly declare qualified identifiers such as in a constant declaration  $s/c : \tau = \delta$ . In most languages, such declarations are forbidden. But such declarations are introduced when flattening the theory graph, and we want the flat theory graph to be well-formed as well. It is natural to solve this problem by assuming that the flattening algorithm can always generate fresh names for the induced constants. However, such a non-canonical choice of identifiers prevents interoperability.

Therefore, MMT permits declarations that introduce qualified identifiers. This is in fact quite natural because deep assignments in links introduce assignments to qualified identifiers already. The definition of clash-freeness handles both theories and links uniformly: Theories may not explicitly declare both a structure  $s$  and a constant  $s/c$ , and links may not provide both an assignment for a structure  $s$  and a deep assignment for an induced constant  $s/c$ .

More precisely, we have:

**Lemma 20.** *If a theory graph  $\gamma$  is clash-free, then the judgments of Fig. 9 are well-defined functions where the red parameters are input and the blue ones output.*

*Proof.* This follows by a simple induction over the derivations of the elaboration judgments.  $\square$

*Example 21* (Continued from Ex. 1) In our running example, we have the theory  $\gamma > e?\text{CGroup} = \{\dots\}$  and the structure  $\gamma \gg e?\text{CGroup}/\text{mon} : e?\text{Monoid} \rightarrow e?\text{CGroup} = \{\}$ . This structure has the induced assignment  $\gamma \gg_{e?\text{CGroup}/\text{mon}} \text{comp} \mapsto e?\text{CGroup}?\text{mon}/\text{comp}$  according to rule  $dflt\_ass\_str$ . And we have the induced constant

$$\gamma >_{e?\text{CGroup}} \text{mon}/\text{comp} : @((m?\text{LF}? \rightarrow, f?\text{FOL}? \iota, f?\text{FOL}? \iota, f?\text{FOL}? \iota)^{e?\text{CGroup}/\text{mon}} = \perp$$

according to rule  $ind\_con\_dflt$ .

#### 4.4. Normal Terms

Because MMT is foundation-independent, the equality relation on terms is transparent to MMT. However, some concepts of MMT influence the equality between terms. In particular, the result of a morphism application  $\omega^\mu$  can be computed by homomorphically replacing all constants in  $\omega$  with their assignments under  $\mu$ . In the sequel, we define this equality relation to the extent that it is imposed by MMT.

We define a **normal form**  $\bar{\omega}$  for MMT terms  $\omega$ . Normalization eliminates all morphism applications, expands all definitions, and enforces the strictness of filtering. The latter means that a term with a filtered subterm is also filtered. Technically,  $\bar{\omega}$  is relative to a fixed theory graph, but we will suppress that in the notation.

$\bar{\omega}$  is defined by structural induction using sub-inductions for the case of morphism application. The definition is given in Fig. 13. There, we also define  $\bar{T}$ , the straightforward extension of normalization to contexts; as before, we assume  $\perp^l = \perp$  to avoid case distinctions.

Among the cases in Fig. 13, the case  $(D?c)^l$  is the most interesting. First of all, we assume  $\gamma \gg l : S \rightarrow T = \perp$  and  $\gamma >_D c : \perp = \delta$ . Note that we permit the case  $D \neq S$ : Below we will see that in well-formed theory graphs  $D$  must be  $S$  or a possibly indirect meta-theory of  $S$ . The definition distinguishes three subcases. If  $D?c$  has a definiens  $\delta \neq \perp$ , it is expanded before applying  $l$  (first subcase) – firstly because  $l$  should not have to give assignments for defined constants, and secondly because  $l$  might filter the name  $c$ . Otherwise, if  $D \neq S$ , then  $l$  must have a meta-morphism  $\mu \hookrightarrow l$ , which is applied to  $D?c$  (second case). Finally, if  $D = S$ , then  $l$  must provide an assignment  $\gamma \gg_l c \mapsto \delta'$  (third subcase).

We use a functional notation  $\bar{\omega}$  for the normal form. But technically, if the underlying theory graph is arbitrary, the normal form does not always exist uniquely, e.g., if the theory graph is not clash-free. We will show in Sect. 5 that  $\bar{\omega}$  exists uniquely if the underlying theory graph  $\gamma$  is well-formed, which justifies our notation.

*Example 22* (Continued from Ex. 1)

Consider the type of the constant  $e?\text{CGroup?mon/comp}$  from Ex. 21. After two normalization steps, we obtain

$$\begin{aligned} & @((m?\text{LF}\? \rightarrow, f?\text{FOL}\? \iota, f?\text{FOL}\? \iota, f?\text{FOL}\? \iota)^{e?\text{CGroup?mon}} = \\ & @(\overline{m?\text{LF}\? \rightarrow^{e?\text{CGroup?mon}}}, \overline{f?\text{FOL}\? \iota^{e?\text{CGroup?mon}}}, \overline{f?\text{FOL}\? \iota^{e?\text{CGroup?mon}}}, \overline{f?\text{FOL}\? \iota^{e?\text{CGroup?mon}}}) \end{aligned}$$

And using  $\text{id}_{f?\text{FOL}} \hookrightarrow e?\text{CGroup?mon}$   $\gamma >_{f?\text{FOL}} \iota : \perp = \perp$ , we obtain further

$$\overline{f?\text{FOL}\? \iota^{e?\text{CGroup?mon}}} = \overline{f?\text{FOL}\? \iota^{\text{id}_{f?\text{FOL}}}} = \overline{f?\text{FOL}\? \iota} = f?\text{FOL}\? \iota$$

so that the result of normalization is  $@(m?\text{LF}\? \rightarrow, f?\text{FOL}\? \iota, f?\text{FOL}\? \iota, f?\text{FOL}\? \iota)$  as expected.

$$\begin{array}{l}
\overline{\top} \quad \top \\
\overline{x} \quad := \quad x \\
\overline{T?c} \quad := \quad \begin{cases} \overline{\delta} & \text{if } \gamma >_T c : \_ = \delta \text{ and } \delta \neq \perp \\ T?c & \text{otherwise} \end{cases} \\
\overline{@(\omega_1, \dots, \omega_n)} \quad := \quad \begin{cases} @(\overline{\omega_1}, \dots, \overline{\omega_n}) & \text{if } \overline{\omega_i} \neq \top \text{ for all } i \\ \top & \text{otherwise} \end{cases} \\
\overline{\beta(\omega_0; \Upsilon; \omega_1)} \quad := \quad \begin{cases} \beta(\overline{\omega}; \overline{\Upsilon}; \overline{\omega_1}) & \text{if } \overline{\omega_i} \neq \top \text{ for all } i, \overline{\Upsilon} \neq \top \\ \top & \text{otherwise} \end{cases} \\
\overline{\omega^{id_T}} \quad := \quad \overline{\omega} \\
\overline{\omega^\mu \omega^\nu} \quad := \quad \overline{(\omega^\mu)^\nu} \\
\\
\overline{\top^l} \quad := \quad \top \\
\overline{x^l} \quad := \quad x \\
\overline{@(\omega_1, \dots, \omega_n)^l} \quad := \quad \overline{@(\omega_1^l, \dots, \omega_n^l)} \\
\overline{\beta(\omega_0; \Upsilon; \omega_1)^l} \quad := \quad \overline{\beta(\omega_0^l; \Upsilon^l; \omega_1^l)} \\
\overline{(\omega^\mu)^l} \quad := \quad \overline{\omega^{\mu^l}} \\
\overline{(D?c)^l} \quad := \quad \begin{cases} \overline{\delta^l} & \text{if } \delta \neq \perp \\ \overline{(D?c)^\mu} & \text{if } \delta = \perp, D \neq S, \mu \hookrightarrow l \\ \overline{\delta'} & \text{if } \delta = \perp, D = S, \gamma \gg_l c \mapsto \delta' \end{cases} \\
\\
\overline{\cdot} \quad := \quad \cdot \\
\overline{\Upsilon, x : \tau = \delta} \quad := \quad \begin{cases} \overline{\Upsilon}, x : \overline{\tau} = \overline{\delta} & \text{if } \overline{\Upsilon} \neq \top, \overline{\tau} \neq \top \text{ and } \overline{\delta} \neq \top \\ \top & \text{otherwise} \end{cases}
\end{array}$$

Figure 13: Normalization

## 5. Well-formed Expressions

In this section we define the well-formed MMT expressions (also called valid expressions). Only those are meaningful. First we define a set of judgments in Sect. 5.1, and we give a set of inference rules for them in Sect. 5.3–5.5. Because MMT is generic, both the judgments and the rules are parametric in a foundation, which we define in Sect. 5.2.

### 5.1. Judgments

The judgments for MMT are given in Fig. 14. They are relative to a fixed foundation, which we omit from the notation.

| Judgment  | Intuition   |
|---|---|
| $\triangleright \gamma$                                   | $\gamma$ is a well-formed theory graph.   |
| $\gamma; \Upsilon \triangleright_T \omega$                | $\omega$ is structurally well-formed over $\gamma$ , $T$ , and $\Upsilon$ .       |
| $\gamma; \Upsilon \triangleright_T \omega : \omega'$      | $\omega$ is well-typed with type $\omega'$ over $\gamma$ , $T$ , and $\Upsilon$ . |
| $\gamma; \Upsilon \triangleright_T \omega \equiv \omega'$ | $\omega$ and $\omega'$ are equal over $\gamma$ , $T$ , and $\Upsilon$ .           |
| $\gamma \triangleright \mu : S \rightarrow T$             | $\mu$ is a well-typed morphism from $S$ to $T$ .                                  |
| $\gamma \triangleright \mu \equiv \mu' : S \rightarrow T$ | $\mu$ and $\mu'$ are equal as morphisms from $S$ to $T$ .                         |

Figure 14: Typing Judgments

For the **structural levels**, the inference system uses a single judgment  $\triangleright \gamma$  for well-formed theory graphs. The inference rules will define how well-formed theory graphs can be extended incrementally. There are three kinds of extensions of a theory graph  $\gamma$ :

- add a module at the end of  $\gamma$  – see the rules in Fig. 15,
- add a symbol at the end of the last module of  $\gamma$  (which must be a theory)  
– see the rules in Fig. 16,
- add an assignment to the last link of  $\gamma$  (which may be a view if  $\gamma$  ends in that view, or a structure if  $\gamma$  ends in a theory which ends in that structure)  
– see the rules in Fig. 17.

When theories or links are added, their body is empty initially and populated incrementally by adding symbols and assignments, respectively. This has the effect that there is exactly one inference rule for every theory, view, symbol, or assignment, i.e., for every URI-bearing knowledge item.

For the **object level**, we use judgments for terms and for morphisms.  $\gamma; \Upsilon \triangleright_T \omega : \omega'$  and  $\gamma; \Upsilon \triangleright_T \omega \equiv \omega'$  express **typing and equality of terms** in context  $\Upsilon$  and theory  $T$ . These judgments are not defined generically by MMT; instead, they are defined by the foundation (see Sect. 5.2). MMT only provides the judgment  $\gamma; \Upsilon \triangleright_T \omega$ , for structurally well-formed terms; this is the strongest necessary condition for the well-formedness of  $\omega$  that does not depend on the foundation. In all three judgments, we omit  $\Upsilon$  when it is empty.

As before, we will occasionally write  $\perp$  when the optional type or definition of a constant or variable is not present. For that case, it is convenient to extend the equality and typing judgment to  $\perp$ . We write  $\gamma; \Upsilon \triangleright_T \omega : \perp$  to express that  $\omega$  is a well-formed untyped value, and  $\gamma; \Upsilon \triangleright_T \perp : \omega$  to express that  $\omega$  is a well-formed type, i.e., a term that may occur on the right hand side of  $::$ . Moreover, we assume that  $\gamma \triangleright_T \perp \equiv \perp$ .

Contrary to the judgments for terms, all judgments for **typing and equality of morphisms** are defined foundation-independently by MMT.  $\gamma \triangleright \mu : S \rightarrow T$

expresses that  $\mu$  is a well-formed morphism from  $S$  to  $T$ . Similarly,  $\gamma \triangleright \mu \equiv \mu' : S \rightarrow T$  expresses equality. This notation emphasizes the category theoretic intuition of morphisms with domain and codomain. If, instead, we prefer the type theoretic intuition of realizations as typed objects, we can use the notation  $\gamma \triangleright_T \mu : S$  (speak:  $\mu$  is a well-typed realization of  $S$  over  $T$ ).

### 5.2. Foundations

Intuitively, foundations attach a semantics to the constants occurring in the foundational theories. For the purposes of MMT, this is achieved as follows:

**Definition 23.** A **foundation** is a definition of the judgments  $\gamma; \Upsilon \triangleright_T \omega : \omega'$  and  $\gamma; \Upsilon \triangleright_T \omega \equiv \omega'$ . In order to avoid case distinctions, we require foundations to define these judgments also for the cases where  $\omega$  or  $\omega'$  are  $\perp$ .

In theoretical accounts, foundations can be given, for example, as an inference system or a decision procedure, or via a denotational semantics. In the MMT implementation, foundations are realized as oracles that are provided by plugins.

In fact, inspecting the rules of MMT will show that MMT only needs the special case of these judgments where  $\Upsilon$  is the empty context. But it is useful to require the general case to permit future extensions of MMT; moreover, for most foundations, the use of an arbitrary context makes the definitions easier.

While the details of the foundation are transparent to MMT, it is useful to impose a regularity condition on foundations that captures some intuitions of typing and equality. First we need an auxiliary definition for the declaration-wise equality of contexts:

**Definition 24.** For two contexts  $\Upsilon^j = (x : \tau_1^j = \delta_1^j, \dots, x : \tau_n^j = \delta_n^j)$  for  $j = 1, 2$ , we write  $\gamma; \Upsilon^0 \triangleright_T \Upsilon^1 \equiv \Upsilon^2$  iff for all  $i = 1, \dots, n$

$$\gamma; \Upsilon^0, \Upsilon_{i-1}^1 \triangleright_T \tau_i^1 \equiv \tau_i^2 \quad \text{and} \quad \gamma; \Upsilon^0, \Upsilon_{i-1}^1 \triangleright_T \delta_i^1 \equiv \delta_i^2$$

where  $\Upsilon_i^1 := x : \tau_1^1 = \delta_1^1, \dots, x : \tau_i^1 = \delta_i^1$ . Recall that we assume  $\gamma \triangleright_T \perp \equiv \perp$  to avoid case distinctions.

**Definition 25** (Regular Foundation). A foundation is called **regular** if it satisfies the following conditions where  $\gamma$ ,  $T$ , and all terms are arbitrary:

1. The equality judgment respects normalization:

$$\gamma \triangleright_T \omega \equiv \bar{\omega}$$

2. The equality relation induced by  $\gamma; \Upsilon \triangleright_T \omega \equiv \omega'$  is an equivalence relation for every  $\Upsilon$  and satisfies the following congruence laws (where  $i$  runs over

the respective applicable indices):

$$\begin{aligned}
& \gamma; \Upsilon \triangleright_T \omega_i \equiv \omega'_i \text{ implies } \gamma; \Upsilon \triangleright_T @(\omega_0, \dots, \omega_n) \equiv @(\omega'_0, \dots, \omega'_n) \\
& \gamma; \Upsilon_0 \triangleright_T \omega_i \equiv \omega'_i \text{ and } \gamma; \Upsilon_0 \triangleright_T \Upsilon \equiv \Upsilon' \text{ implies} \\
& \quad \gamma; \Upsilon_0 \triangleright_T \beta(\omega_0; \Upsilon; \omega_1) \equiv \beta(\omega'_0; \Upsilon'; \omega'_1) \\
& \gamma; \Upsilon \triangleright_T \omega_i \equiv \omega'_i \text{ and } \gamma; \Upsilon \triangleright_T \omega_1 : \omega_2 \text{ implies } \gamma; \Upsilon \triangleright_T \omega'_1 : \omega'_2 \\
& \quad \gamma \triangleright_T \Upsilon \equiv \Upsilon' \text{ and } \gamma; \Upsilon \triangleright_T \omega_1 \equiv \omega_2 \text{ implies } \gamma; \Upsilon' \triangleright_T \omega_1 \equiv \omega_2 \\
& \quad \gamma \triangleright_T \Upsilon \equiv \Upsilon' \text{ and } \gamma; \Upsilon \triangleright_T \omega_1 : \omega_2 \text{ implies } \gamma; \Upsilon' \triangleright_T \omega_1 : \omega_2
\end{aligned}$$

Note that we do not impose a congruence law for morphism application at this point.

3. Foundations preserve typing and equality along flat morphisms. To state this precisely, assume flat theories  $S$  and  $T$ . Moreover assume a mapping  $f$  of constant identifiers to terms such that: Whenever  $D = S$  or  $D$  is a possibly indirect meta-theory of  $S$  and  $c : \tau = \delta$  is declared in  $D$ , then  $\gamma \triangleright_T f(D?c) : f(\tau)$  and  $\gamma \triangleright_T f(D?c) \equiv f(\delta)$ .

Then we require that for two flat terms  $\gamma \triangleright_S \omega_i$

$$\begin{aligned}
& \gamma \triangleright_S \omega_1 : \omega_2 \text{ implies } \gamma \triangleright_T f(\omega_1) : f(\omega_2) \\
& \gamma \triangleright_S \omega_1 \equiv \omega_2 \text{ implies } \gamma \triangleright_T f(\omega_1) \equiv f(\omega_2)
\end{aligned}$$

where  $f(\omega)$  arises by replacing every constant  $D?c$  in  $\omega$  with  $f(D?c)$ .

Regular foundations are uniquely determined by their action on flat terms so that the module system is transparent to the foundation:

**Lemma 26.** *For every regular foundation and arbitrary  $\gamma, T, \omega, \omega'$ :*

$$\begin{aligned}
& \gamma \triangleright_T \omega \equiv \omega' \text{ iff } \gamma \triangleright_T \bar{\omega} \equiv \bar{\omega'} \\
& \gamma \triangleright_T \omega : \omega' \text{ iff } \gamma \triangleright_T \bar{\omega} : \bar{\omega'}
\end{aligned}$$

*Proof.* The first equivalence follows easily using property (1), symmetry, and transitivity. The second equivalence follows easily using property (1), symmetry, and the last of the congruence properties.  $\square$

Note that the typing and equality judgments are only assumed for the foundational theories. For all other theories, the typing and equality judgments are inherited from the respective meta-theory. For example, a foundation for SML must specify the typing and normalization relations of SML expressions. And a foundation for ZFC must specify the well-formedness and provability of propositions, both of which we consider as special cases of typing.

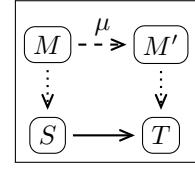
It is no coincidence that exactly these two judgments form the interface between MMT and the foundation: They are closely connected to the syntax of MMT constant declarations, which may carry types and definitions. If types

can be declared for the constants of a language, then the typing relation should be extended to all complex expressions. This is necessary, for example, to check that theory morphisms preserve types. Similarly, if the constants may carry definitions, an equality relation for complex expressions becomes necessary. Vice versa, MMT constant declarations provide the foundation with the base cases for the definitions of typing and equality.

### 5.3. Inference Rules for the Structural Levels

When defining well-formed theory graphs, we assume for simplicity that all theory graphs are clash-free. It is straightforward to extend all inference rules with additional newness hypotheses for identifiers such that eventually  $\triangleright \gamma$  implies that  $\gamma$  is clash-free. But we omit this here to simplify the notation.

The rules in Fig. 15 define theory graphs as lists of modules. The rule *Start* starts with an empty theory graph, and the rules *Thy* and *View* add modules with empty bodies (that will be filled incrementally). Rule *View* unifies the cases whether  $S$  has a meta-theory or not by using square brackets for optional parts; whether  $T$  has a meta-theory, is irrelevant.



Finally *ViewDef* adds a view defined by a morphism.

In rule *View*, one might intuitively expect the assumption  $[M \hookrightarrow S \quad M' \hookrightarrow T \quad \gamma \triangleright \mu : M \rightarrow M']$  which is the situation depicted in the diagram on the right. That case is subsumed by *View* as we will see in the rules for morphisms below.

|  |  |   |                  |
|--|--|---|------------------|
| $\rule{0pt}{1.2em} \quad \text{Start}$ | $\frac{\triangleright \gamma \quad [\gamma > M = \{-\}]}{\triangleright \gamma, T \stackrel{[M]}{\equiv} \{\cdot\}}$ | $\text{Thy}$  |                  |
|  |  | $\frac{\triangleright \gamma \quad \gamma \triangleright \mu : S \rightarrow T}{\triangleright \gamma, m : S \rightarrow T = \mu}$  | $\text{ViewDef}$ |
|  |  | $\frac{\triangleright \gamma \quad \gamma > S = \{-\} \quad \gamma > T = \{-\} \quad [M \hookrightarrow S \quad \gamma \triangleright \mu : M \rightarrow T]}{\triangleright \gamma, m : S \rightarrow T \stackrel{[\mu]}{\equiv} \{\cdot\}}$ | $\text{View}$    |

Figure 15: Adding Modules

The rules in Fig. 16 add symbols to theories. There are three cases corresponding to the three kinds of symbols: constants, structures defined by a morphisms, and structures defined by a list of assignments. The rule *Con* says that constant declarations  $c : \tau = \delta$  can be added if  $\delta$  has type  $\tau$ . Recall that this includes the cases where  $\tau = \perp$  or  $\delta = \perp$ .

Note also that  $\gamma \triangleright_T \delta$  and  $\gamma \triangleright_T \tau$  are necessary even though we require  $\gamma \triangleright_T \delta : \tau$ : Indeed in most type systems, the latter would entail the former two,

but in MMT the typing judgment is given by the foundation as an oracle, so we cannot be sure.

The rules  $Str$  and  $StrDef$  are completely analogous to the rules  $View$  and  $ViewDef$ . Again square brackets are used in  $Str$  to unify the two cases where  $S$  has a meta-theory or not. In all three rules it is irrelevant whether  $T$  has a meta-theory or not; we indicate that by giving this optional meta-theory in gray.

$$\begin{array}{c}
 \boxed{\begin{array}{c}
 \frac{\triangleright \gamma, T \stackrel{M}{=} \{\vartheta\} \quad \gamma' \triangleright_T \delta \quad \gamma' \triangleright_T \tau \quad \gamma' \triangleright_T \delta : \tau}{\triangleright \gamma, T \stackrel{M}{=} \{\vartheta, c : \tau = \delta\}} Con \\
 \\ 
 \frac{\triangleright \gamma, T \stackrel{M}{=} \{\vartheta\} \quad \gamma' \triangleright \mu : S \rightarrow T}{\triangleright \gamma, T \stackrel{M}{=} \{\vartheta, s : S = \mu\}} StrDef \\
 \\ 
 \frac{\triangleright \gamma, T \stackrel{M}{=} \{\vartheta\} \quad \gamma > S = \{\cdot\} \quad [M' \hookrightarrow S \quad \gamma' \triangleright \mu : M' \rightarrow T]}{\triangleright \gamma, T \stackrel{M}{=} \left\{ \vartheta, s : S \stackrel{[\mu]}{=} \{\cdot\} \right\}} Str
 \end{array}}
 \end{array}$$

Figure 16: Adding Symbols ( $\gamma'$  abbreviates  $TG$ ,  $T \stackrel{M}{=} \{\vartheta\}$ )

The addition of assignments to a link  $l$  is more complicated because assignments can be added to views or structures. MMT treats both cases in the same way, which we want to stress by unifying the rules. Therefore, let  $\gamma \gg^{\text{last}} l : S \rightarrow T$  denote that  $l$  is a link occurring at the end of  $\gamma$ , i.e., either

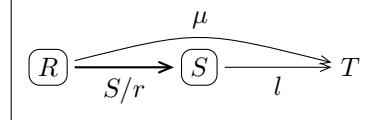
- $l$  refers to a view and  $\gamma = \dots, l : S \rightarrow T \stackrel{[\mu]}{=} \{\sigma\}$  or
- $l = T/s$  refers to a structure and  $\gamma = \dots, T \stackrel{[M]}{=} \left\{ \dots, s : S \stackrel{[\mu]}{=} \{\sigma\} \right\}$ ,

and in that case let  $\gamma + Ass$  be the theory graph arising from  $\gamma$  by replacing  $\sigma$  with  $\sigma, Ass$ .

Then the rules in Fig. 17 add assignments to a link.  $ConAss$  adds an assignment  $c \mapsto \delta$  for an undefined constant  $c$  of  $S$ . Such assignments are well-typed if  $\delta$  is typed by the translation of the type of  $c$  along  $l$ . Again we assume  $\perp^l := \perp$  to avoid case distinctions.

Rule  $ConAss$  includes the case of  $\delta = \top$ , i.e., undefined constants can be filtered by mapping them to  $\top$ . For defined constants, filtering is the only possible assignment; this is covered by Rule  $ConFlt$ .

The rule  $StrAss$  is similar to  $ConAss$  except that adding assignments for structures is a bit more complicated. The first three hypotheses correspond to the rule



|   |  |
|---|--|
| $\triangleright \gamma \quad \gamma \gg^{\text{last}} l : S \rightarrow T \quad \gamma >_S c : \tau = \perp \quad \gamma \triangleright_T \delta \quad \gamma \triangleright_T \delta : \tau^l$   | $\frac{}{\triangleright \gamma + c \mapsto \delta} ConAss$ |
| $\triangleright \gamma \quad \gamma \gg^{\text{last}} l : S \rightarrow T \quad \gamma >_S c : \_ = \delta \quad \delta \neq \perp$   | $\frac{}{\triangleright \gamma + c \mapsto \top} ConFlt$   |
| $\triangleright \gamma \quad \gamma \gg^{\text{last}} l : S \rightarrow T \quad \gamma \gg S/r : R \rightarrow S = \_ \quad \gamma \triangleright \mu : R \rightarrow T$<br>$[M \hookrightarrow R \quad \mu' \hookrightarrow S/r \quad \gamma \triangleright \mu \equiv \mu' l : M \rightarrow T]$<br>$\gamma \triangleright_T \delta^l \equiv (R?c)^\mu \text{ whenever } \gamma >_S r/c : \_ = \delta, \delta \neq \perp$ | $\frac{}{\triangleright \gamma + r \mapsto \mu} StrAss$    |

Figure 17: Adding Assignments

*StrAss.* The guiding intuition for the remaining hypotheses is that an assignment  $r \mapsto \mu$  for a structure  $r$  in  $S$  should make the diagram on the right commute. From this intuition, we can immediately derive the typing requirements that  $\mu$  must be a well-typed morphism from  $R$  to  $T$ .

However, this is not sufficient yet to make the diagram commute. In general, the link  $l$  already contains some assignments and possibly a meta-morphism so that the semantics of the composition  $S/r/l$  is already partially determined. Therefore,  $\mu$  must agree with  $S/r/l$  whenever the latter is already determined.

This is easy for a possible meta-morphism of  $\gamma \triangleright \mu' : M \rightarrow S$  of  $S/r$ . The composition of  $\mu'$  and  $l$  must agree with the restriction of  $\mu$  to  $M$ . Additionally, for all constants  $r/c$  of  $S$  that have a definiens  $\delta$ , the translation of  $\delta$  along  $l$  must be equal to the translation of  $R?c$  along  $\mu$ .

The rule *StrAss* is in fact inefficient because it requires to flatten  $S$ , i.e., to compute all induced constants  $r/c$  of  $S$ . But it is important for scalability to avoid this whenever possible. Therefore, we give some admissible rules of inference in Sect. 5.6 that use module-level reasoning to avoid flattening.

#### 5.4. Inference Rules for Morphisms

Fig. 18 gives the typing rules for morphisms. The rule  $\mathcal{M}_{\text{link}}$  handles links.  $\mathcal{M}_{\text{ident}}$  and  $\mathcal{M}_{\text{comp}}$  give identity and composition of morphisms. Meta-theories behave like inclusions with regard to composition of morphisms: The rules  $\mathcal{M}_{\text{covar}}$  and  $\mathcal{M}_{\text{contravar}}$  give the usual co- and contravariance rules.

Finally, we define the equality of morphisms in rule  $\mathcal{M}_\equiv$ . We use an extensional equality that identifies two morphisms if they map the same argument to equal terms. This is equivalent to the special case where the morphisms agree for all undefined constants. If the domain has a meta-theory  $M$ , the

meta-morphisms must be equal as well. This is checked recursively by requiring  $\gamma \triangleright \mu \equiv \mu' : M \rightarrow T$ .

|  |  |
|--|--|
| $\frac{\gamma \gg l : S \rightarrow T = \_}{\gamma \triangleright l : S \rightarrow T} \mathcal{M}_{link}$   |  |
| $\frac{\gamma > T = \{ \_ \}}{\gamma \triangleright id_T : T \rightarrow T} \mathcal{M}_{ident}$   | $\frac{\gamma \triangleright \mu : R \rightarrow S \quad \gamma \triangleright \mu' : S \rightarrow T}{\gamma \triangleright \mu \mu' : R \rightarrow T} \mathcal{M}_{comp}$ |
| $\frac{M \hookrightarrow S \quad \gamma \triangleright \mu : S \rightarrow T}{\gamma \triangleright \mu : M \rightarrow T} \mathcal{M}_{contravar}$  | $\frac{\gamma \triangleright \mu : S \rightarrow M \quad M \hookrightarrow T}{\gamma \triangleright \mu : S \rightarrow T} \mathcal{M}_{covar}$                              |
| $\frac{\begin{array}{c} \gamma \triangleright \mu : S \rightarrow T \quad \gamma \triangleright \mu' : S \rightarrow T \\ \gamma \triangleright_T S?c^\mu \equiv S?c^{\mu'} \text{ whenever } \gamma >_S c : \_ = \perp \\ [M \hookrightarrow S \quad \gamma \triangleright \mu \equiv \mu' : M \rightarrow T] \end{array}}{\gamma \triangleright \mu \equiv \mu' : S \rightarrow T} \mathcal{M}_\equiv$ |  |

Figure 18: Morphisms

### 5.5. Inference Rules for Terms

As noted above, MMT relegates the judgments

$$\gamma; \Upsilon \triangleright_T \omega \equiv \omega' \quad \text{and} \quad \gamma; \Upsilon \triangleright_T \omega : \omega'$$

for typing and equality of terms to the foundation. MMT only defines the judgment  $\gamma; \Upsilon \triangleright_T \omega$  for **structurally well-formed** terms. Structural well-formedness guarantees in particular that only constants and variables are used that are in scope.

This judgment is axiomatized by the rules in Fig. 19. First we define an auxiliary judgment  $\gamma \triangleright_T \Upsilon$  for well-formed contexts using the rules  $\mathcal{T}$  and  $\mathcal{T}_\Upsilon$ . These are such that every variable may occur in the types and definitions of subsequent variables. The rules  $\mathcal{T}_x$ ,  $\mathcal{T}_c$ ,  $\mathcal{T}_\top$ ,  $\mathcal{T}_@$ , and  $\mathcal{T}_\perp$  are straightforward.  $\mathcal{T}_\perp$  is such that a bound variable may occur in the type or definition of subsequent variables in the same binder.

Finally,  $\mathcal{T}_\mu$  and  $\mathcal{T}_\hookrightarrow$  formalize the cases relevant for the MMT module system.  $\mathcal{T}_\mu$  moves closed terms along morphisms, and  $\mathcal{T}_\hookrightarrow$  moves terms along the meta-theory relation. Note that  $\omega^\mu$  is well-formed independent of whether  $\omega$  is filtered by  $\mu$ . This is important because the decision whether  $\omega$  is filtered is expensive if the theory graph has not been flattened yet.

|  |   |
|--|---|
| $\frac{\gamma > T = \{-\}}{\gamma \triangleright_T \cdot} \mathcal{T}$   | $\frac{\gamma \triangleright_T \Upsilon \quad [\gamma; \Upsilon \triangleright_T \tau] \quad [\gamma; \Upsilon \triangleright_T \delta]}{\gamma \triangleright_T \Upsilon, x[: \tau][= \delta]} \mathcal{T}_\Upsilon$ |
| $\frac{\gamma \triangleright_T \Upsilon \quad x : \_ = \_ \text{ in } \Upsilon}{\gamma; \Upsilon \triangleright_T x} \mathcal{T}_x$  | $\frac{\gamma \triangleright_T \Upsilon \quad \gamma >_T c : \_ = \_}{\gamma; \Upsilon \triangleright_T T?c} \mathcal{T}_c$   |
|  | $\frac{\gamma \triangleright_T \Upsilon}{\gamma; \Upsilon \triangleright_T \top} \mathcal{T}_\top$  |
| $\frac{\gamma; \Upsilon \triangleright_T \omega_i \text{ for all } i = 1, \dots, n}{\gamma; \Upsilon \triangleright_T @(\omega_1, \dots, \omega_n)} \mathcal{T}_@$                             | $\frac{\gamma; \Upsilon \triangleright_T \omega \quad \gamma \triangleright_T \Upsilon, \Upsilon'}{\gamma; \Upsilon \triangleright_T \beta(\omega; \Upsilon'; \omega')} \mathcal{T}_\emptyset$                        |
| $\frac{\gamma \triangleright_T \Upsilon \quad \gamma \triangleright_S \omega \quad \gamma \triangleright \mu : S \rightarrow T}{\gamma; \Upsilon \triangleright_T \omega^\mu} \mathcal{T}_\mu$ | $\frac{\gamma; \Upsilon \triangleright_M \omega \quad M \hookrightarrow T}{\gamma; \Upsilon \triangleright_T \omega} \mathcal{T}_{\hookrightarrow}$   |

Figure 19: Structurally Well-formed Terms

It is easy to prove a subexpression property for structural well-formedness: If  $\gamma; \Upsilon \triangleright_T \omega$  then all subexpressions of  $\omega$  are well-formed in the respective context.

### 5.6. Module-Level Reasoning

The extensional definition of the equality of morphisms is very inefficient because it requires the full elaboration of the domain. We encountered a similar problem in rule *StrAss*. To remedy this, we introduce the following admissible rule of inference, which refines rule  $\mathcal{M}_\equiv$  to avoid elaboration:

$$\frac{\begin{array}{c} \gamma \triangleright \mu : S \rightarrow T \quad \gamma \triangleright \mu' : S \rightarrow T \\ \gamma \triangleright_T S?c^\mu \equiv S?c^{\mu'} \text{ whenever } c : \_ \text{ in } S \\ \gamma \triangleright S/s \mu \equiv S/s \mu' : R \rightarrow T \text{ whenever } s : R \doteq \{-\} \text{ in } S \\ [M \hookrightarrow S \quad \gamma \triangleright \mu \equiv \mu' : M \rightarrow T] \end{array}}{\gamma \triangleright \mu \equiv \mu' : S \rightarrow T} \mathcal{M}'_\equiv$$

Both  $\mathcal{M}_\equiv$  and  $\mathcal{M}'_\equiv$  require  $\gamma \triangleright_T S?c^\mu \equiv S?c^{\mu'}$  for the constants of  $S$ . But in  $\mathcal{M}_\equiv$ , this is required for all constants, including the ones induced by structures.  $\mathcal{M}'_\equiv$ , on the other hand, only requires it for the local constants of  $S$ , which can be verified without elaboration. For the induced constants of  $S$ ,

rule  $\mathcal{M}'_{\equiv}$  recursively checks equality of morphisms for every structure declared in  $S$ . A variant of *StrAss* that does not require elaboration can be obtained in a similar way.

By unraveling the recursion, it is easy to see that  $\mathcal{M}'_{\equiv}$  eventually checks the same prerequisites as  $\mathcal{M}_{\equiv}$ . Therefore,  $\mathcal{M}'_{\equiv}$  by itself does not yield an efficiency gain. However, we can often avoid the recursive calls in  $\mathcal{M}'_{\equiv}$  by using other, more efficient admissible rules to establish the equality of two morphisms.

These additional rules are axioms that are obtained from the invariants of MMT. Firstly, we have one equality axiom for every defined view or structure. And secondly, Thm. 31 establishes the soundness of one equational axiom for every structure assignment. Recall that all nodes and edges in the theory graph have URIs, and morphisms are paths in the theory graph, i.e., lists of URIs. Therefore, representing them and reasoning about the equality of morphisms using these equational axioms is efficient in most cases.

We call this module-level reasoning because it forgets all details about the bodies of theories and links and only uses the theory graph. Naturally module-level reasoning about equality of morphisms is sound but not complete. Moreover, the equational theory of paths in the theory is not necessarily decidable. However, in our experience, module-level reasoning succeeds in the majority of cases occurring in practice.

## 6. Formal Properties

Now that we have established the grammar and well-formedness conditions for MMT, we can analyze the properties of well-formed theory graphs: In Sect. 6.1 we establish that normalization is well-defined and in Sect. 6.2 that assignment to structures can be used to establish commutativity conditions in theory graphs. In Sect. 6.3 we introduce the concept of structural well-formedness, as a computationally motivated compromise between MMT-well-formedness and grammatical well-formedness. Finally, in Sect. 6.4 we examine the operation of flattening (i.e. copying out the modular aspects of MMT) as a semantics-giving operation of theory graphs and show that in MMT it can be made incremental, which is important for computational tractability and scalability.

### 6.1. Theory Graphs

To finish the formal definition of MMT, we must take care of one proof obligation that we have deferred so far: the well-definedness of normalization.

**Lemma 27.** *Assume a regular foundation and (i)  $\triangleright \gamma$  for a clash-free  $\gamma$  and (ii)  $\gamma \triangleright_T \omega$ . Then  $\bar{\omega}$  is well-defined and does not contain any morphism applications.*

*Proof.* Inspecting the definition of  $\bar{\omega}$ , we see there is exactly one case for every possible term  $\omega$ . Technically, this observation uses (i) to deduce that  $\gamma$  is clash-free so that all lookups occurring during the normalization are well-defined. It also uses (ii) to conclude that whenever the case  $\overline{D?c^l}$  occurs,  $D$  is either the domain of  $l$  or a possibly indirect meta-theory of it.

Furthermore, a straightforward induction shows that if the normal form is well-defined, it does not contain morphism applications.

Therefore, the only thing that must be proved is the well-foundedness of the recursive definition. Essentially, this follows because every case decreases one of the following: the size of  $\gamma$ , the size of  $\omega$ , or the size of the subterms of  $\omega$  to which a morphism is applied. Only some cases warrant closer attention:

- $\overline{\omega^{\mu\mu'}}$  and  $\overline{(\omega^\mu)^l}$ . These cases do not decrease the size of the terms involved. But it is easy to see that they recurse between themselves only finitely many times, namely until the term

$$\overline{\frac{l_n}{\omega^{l_1}}} \dots$$

is reached where  $l_1, \dots, l_n$  is the list of links comprising  $\mu\mu'$  (modulo associativity and identity morphism).

- $\overline{T?c}$ . This case may increase the size of the involved terms when a constant is replaced with its definiens. But due to the well-formedness of  $\gamma$  and the regularity of the foundation, the definiens must be structurally well-formed over a theory graph smaller than  $\gamma$ . (In particular, there are no cyclic dependencies between definitions in well-formed theory graphs.)
- $\overline{S?c^l}$ . Similar to the previous case, this case may increase the size of the involved terms when  $S?c^l$  is replaced with the assignment  $l$  provides for  $S?c$ . The same argument applies.

□

## 6.2. Properties of Morphisms

With this bureaucracy out of the way, we can prove some intended properties of morphisms. First we show that morphisms behave as expected. In fact, the presence of filtering makes some of these theorems quite subtle. Therefore, we use the following definition:

**Definition 28.** A morphism  $\gamma \triangleright \mu : S \rightarrow T$  is **total** if  $\overline{S?c^\mu} \neq \top$  whenever  $\gamma >_S c : \_ = \_$  and if its metamorphism (if there is one) is total as well. A theory graph is total if all its links are total morphisms.

Note that a morphism that filters only defined constants is still total because the normalization expands definitions. A morphism is not total if it filters undefined constants. Partial (i.e., non-total) morphisms often behave badly because they do not preserve truth: Assume a view from  $S$  to  $T$  that does not provide an assignment for an axiom  $a$ , maybe because that axiom is not provable in  $T$  at all. Then clearly we cannot expect all theorems of  $S$  to be translated to theorems of  $T$ . However, this property is also what makes partial morphisms interesting in practice: For example, a partial morphism can be used to represent

a translation from a higher-order axiomatization of the real numbers to a first-order one: Such a translation would only translate the first-order-expressible parts, which is still useful in practice.

First, we prove the following intuitively obvious, but technically difficult lemma.

**Lemma 29.** *If  $\gamma \triangleright_S \omega$  and  $\gamma \triangleright \mu : S \rightarrow T$ , then  $\overline{\omega^\mu} = \overline{\omega}^\mu$ .*

*Proof.* This is proved by a straightforward but technical induction on the structure of  $\omega^\mu$ . A notable subtlety is that the primary induction is on  $\gamma$  and  $\mu$  using the statement for arbitrary  $\omega$  as the induction hypothesis. Then the case where  $\mu$  is a link uses a sub-induction on  $\omega$ . We give some example cases where **Def** refers to the definition of normalization and **IH** refers to the induction hypothesis:

- case for a composed morphism in the induction on  $\mu$ :

$$\overline{\omega^{\mu\mu'}} \stackrel{\text{Def}}{=} \overline{(\omega^\mu)^\mu'} \stackrel{\text{IH}\mu'}{=} \overline{\omega^{\mu\mu'}} \stackrel{\text{IH}\mu}{=} \overline{\overline{\omega^\mu}^\mu'} \stackrel{\text{IH}\mu'}{=} (\overline{\omega^\mu})^{\mu'} \stackrel{\text{Def}}{=} \overline{\omega^{\mu\mu'}}$$

- case for  $\omega^l$  for a single link  $l$ , proved by a sub-induction on  $\omega$ :

- case for application:

$$\begin{aligned} \overline{@(\omega_1, \dots, \omega_n)}^l &\stackrel{\text{Def}}{=} \overline{@(\omega_1^l, \dots, \omega_n^l)} \stackrel{\text{IH}}{=} \overline{@(\overline{\omega_1}^l, \dots, \overline{\omega_n}^l)} \stackrel{\text{Def}}{=} \\ \overline{@(\overline{\omega_1}, \dots, \overline{\omega_n})}^l &\stackrel{\text{Def}}{=} \overline{\overline{@(\omega_1, \dots, \omega_n)}}^l \end{aligned}$$

- case for a constant  $D?c$ : If  $\gamma >_D c : \_ = \perp$ , the statement is trivial because  $\overline{D?c} \stackrel{\text{Def}}{=} D?c$ . If  $\gamma >_D c : \_ = \delta \neq \perp$ , then

$$\overline{D?c}^l \stackrel{\text{Def}}{=} \overline{\delta}^l \stackrel{\text{IH}}{=} \overline{\overline{\delta}}^l \stackrel{\text{Def}}{=} \overline{\overline{D?c}}^l$$

□

Then we have the main technical results about theory graphs and morphisms.

**Theorem 30** (Morphisms). *Assume a fixed regular foundation. Then*

1. *For fixed  $\gamma$ , the binary relation on morphisms induced by  $\gamma \triangleright \mu \equiv \mu' : S \rightarrow T$  is an equivalence relation.*
  2. *If  $\gamma \triangleright \mu_1 \equiv \mu'_1 : R \rightarrow S$  and  $\gamma \triangleright \mu_2 \equiv \mu'_2 : S \rightarrow T$  and  $\mu_2$  and  $\mu'_2$  are total, then*
- $$\gamma \triangleright \mu_1 \mu_2 \equiv \mu'_1 \mu'_2 : R \rightarrow T,$$
3. *When composition is well-formed, it is associative and  $\text{id}_T$  is a neutral element.*

4. The identity and the composition of total morphisms are total. In particular, every well-formed total theory graph induces a category of theories and – modulo equality – morphisms.
5. If  $\gamma \triangleright_R \omega$  and  $\gamma \triangleright \mu \mu' : R \rightarrow T$ , then  $\gamma \triangleright_T \omega^{\mu \mu'} \equiv (\omega^\mu)^{\mu'}$ .
6. If  $\gamma \triangleright_S \omega$  and  $\gamma \triangleright \mu \equiv \mu' : S \rightarrow T$ , then

$$\gamma \triangleright_T \omega^\mu \equiv \omega^{\mu'}.$$

7. If  $\gamma \triangleright \mu : S \rightarrow T$ ,  $\gamma \triangleright_S \omega$ ,  $\gamma \triangleright_S \omega'$ , then

- if  $\bar{\omega} = \bar{\omega}'$ , then  $\bar{\omega}^\mu = \bar{\omega}'^\mu$ , and
- if  $\gamma \triangleright_S \omega \equiv \omega'$  and  $\mu$  is total, then  $\gamma \triangleright_T \omega^\mu \equiv \omega'^\mu$ .

*Proof.* 1. Reflexivity, symmetry, and transitivity follow immediately from the corresponding properties for terms using rule  $\mathcal{M}_\equiv$ .

2. This is proved by induction on the number of meta-theories of  $R$ . If there is none, the result follows using rule  $\mathcal{M}_\equiv$  and applying (5) and twice (7). If  $M \hookrightarrow R$ , the same argument applies with  $M$  instead of  $R$ .
3. Because of rule  $\mathcal{M}_\equiv$ , the equality of two morphisms is equivalent to a set of judgments of the form  $\gamma \triangleright_D C^\mu \equiv C^{\mu'}$  for all constants  $c$  of  $S$  or one of its meta-theories. Because the foundation is regular, every such judgment is equivalent to  $\gamma \triangleright_T \bar{C}^\mu \equiv \bar{C}^{\mu'}$ . Then the conclusion follows from the definition of normalization.
4. The totality properties are easy to prove. A category is obtained by taking the theories  $\gamma > T = \{\_\}$  as the objects, and the quotient

$$\{\mu \mid \gamma \triangleright \mu : S \rightarrow T\} / \{(\mu, \mu') \mid \gamma \triangleright \mu \equiv \mu' : S \rightarrow T\}$$

as the set of morphisms from  $S$  to  $T$ . Identity and composition are induced by  $id_T$  and  $\mu \mu'$ . (See [Lan98] for the notion of a *category*.)

5. Because the foundation is regular, the conclusion is equivalent to  $\gamma \triangleright_T \omega^{\mu \mu'} \equiv \bar{(\omega^\mu)}^{\mu'}$ . And this follows directly from the definition of normalization.
6. Using regularity, it is sufficient to show  $\bar{\omega}^\mu = \bar{\omega}^{\mu'}$ . Using Lem. 29, this reduces to the case where  $\omega$  is flat. For flat  $\omega$ , the definition of normalization shows that  $\omega^\mu$  arises from  $\omega$  by replacing all constants  $C$  with  $C^\mu$ .  $\gamma \triangleright \mu \equiv \mu' : S \rightarrow T$  yields  $\gamma \triangleright_T C^\mu \equiv C^{\mu'}$ . Because  $\omega$  is flat, the result follows from property (2) in Def. 25.
7. The first statement follows immediately from Lem. 29. Also using Lem. 29, the conclusion of the second statement reduces to  $\gamma \triangleright_T \bar{\omega}^\mu \equiv \bar{\omega}'^\mu$ , i.e., it is sufficient to consider the case where  $\omega$  and  $\omega'$  are flat. And that

case follows using property (3) in Def. 25 and the type-preservation of well-formed total morphisms guaranteed by rule *ConAss*.

□

The restriction that  $\mu$  must be total in part (7) of Thm. 30 is necessary. To see why, assume  $\omega = @(\pi_1, @(\text{pair}, a, b))$ . A foundation might define  $\gamma \triangleright_S \omega \equiv a$ . Now if  $\mu$  filters  $b$ , then  $\bar{\omega}^\mu = \top$  but not necessarily  $\bar{a}^\mu = \top$ . An even trickier example arises when  $S$  contains an axiom  $a : @(\text{true}, @(\text{equal}, \omega, \omega'))$  and the foundation uses  $a$  to derive  $\gamma \triangleright_S \omega \equiv \omega'$ . If  $\mu$  filters  $a$ , then it is possible that  $\gamma \triangleright_T \omega^\mu \equiv \omega'^\mu$  does not hold even when  $\mu$  filters neither  $\omega$  nor  $\omega'$ .

The following theorem establishes a central property of MMT theory graphs that plays a crucial role in adequacy proofs. In the diagram on the right,  $S$  is a theory with a structure instantiating  $R$ , and  $l$  is a link from  $S$  to  $T$  that assigns  $\mu$  to  $r$ . The theorem states that the triangle commutes. This means that assignments to structures can be used to represent commutativity conditions on diagrams.

**Theorem 31.** *Assume  $\triangleright \gamma$  relative to a fixed regular foundation. If  $\gamma \triangleright \mu : R \rightarrow T$  and  $\gamma \gg l : S \rightarrow T = \{\sigma\}$  such that  $\sigma$  contains the assignment  $r \mapsto \mu$ , then*

$$\gamma \triangleright S/r \quad l \equiv \mu : R \rightarrow T.$$

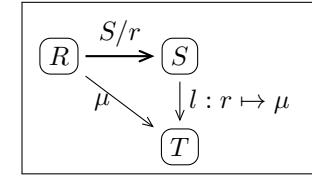
*Proof.* By rule  $\mathcal{M}_\equiv$ , we have to show  $\gamma \triangleright_T R?c^{S/r \mid l} \equiv R?c^\mu$  for all  $\gamma >_R c : \_ = \perp$ . Using the regularity, it is enough to show equality after normalization. A first normalization step reduces the left hand side to  $\overline{S?r/c^l}$ . Now there are two cases differing by whether  $r/c$  has a definiens in  $S$  or not.

- If  $\gamma >_S r/c : \_ = \perp$ , then  $\gamma \gg_l r/c \mapsto R?c^\mu$ , and the left hand side normalizes to  $\overline{R?c^\mu}$ .
- If  $\gamma >_S r/c : \_ = \delta$  for some  $\delta \neq \perp$ , a further normalization step reduces the left hand side to  $\overline{\delta^l}$ . And rule *StrAss* guarantees that in this case  $\gamma \triangleright_T \delta^l \equiv R?c^\mu$ .

Furthermore, we have to show that the morphisms agree on the meta-theory of  $R$  if there is one. This is explicitly required in rule *StrAss*. □

### 6.3. Structural Well-Formedness

The formal definition of structural well-formedness of theory graphs and terms was the original motivation of MMT. Essentially, it means that all references to modules, symbols, or variables exist and are in scope, and that all morphisms are well-typed. But it does not guarantee that terms are well-typed. This yields an intermediate well-formedness level between context-free and semantic validation.



Context-free validation checks a theory graph against a context-free grammar. This is the state of the art in XML-based languages, where the grammar is usually given as XML schema. It is simple and widely implemented, but it is very weak and accepts many meaningless expressions. For example, documents containing references to non-existent knowledge item pass validation. For many knowledge management applications, this is too weak.

Semantic validation on the other hand accepts only meaningful expressions. It checks a theory graph using a type system or an interpretation function. This is normal for formal languages such as logics and type theories. But semantic validation depends on the foundation. Therefore, it is complex, and often only one implementation is available for a specific formal language, which cannot easily be reused by other applications.

Structural well-formedness is a trade-off between these extremes. It is foundation-independent and therefore easy to implement. And the added strength of full validation is not necessary as a precondition for many web scale algorithms such as browsing or versioning.

Technically, structural well-formedness can be defined using a special foundation:

**Definition 32.** The *structural foundation* is the foundation where  $\gamma \triangleright_T \omega : \omega'$  and  $\gamma \triangleright_T \omega \equiv \omega'$  always hold. A theory graph  $\gamma$  is **structurally well-formed** if it is clash-free and  $\triangleright \gamma$  holds relative to the structural foundation.

Clearly, it is not a reasonable mathematical foundation, but it is useful because it is maximal or most permissive among all foundations. It is also easy to implement and can be used as a default foundation when the actual foundation is not known or an implementation for it not available.

Structural well-formedness is foundation-independent in the following sense:

**Theorem 33.** If a theory graph is well-formed relative to any foundation, then it is structurally well-formed. If  $\gamma$  is structurally well-formed, then  $\gamma \triangleright_T \omega$  is independent of the foundation.

*Proof.* The first statement holds because the use of the structural foundation simply amounts to removing the typing and equality hypotheses in the rules *Con* and *ConAss*. The second statement holds because the rules for the judgment  $\gamma \triangleright_T \omega$  do not refer to any other judgment.  $\square$

Corresponding to the notions of structural and semantic validation, we can define structural and semantic equivalence of theory graphs:

**Definition 34.** Relative to a fixed foundation, two well-formed theory graphs  $\gamma$  and  $\gamma'$  are called **structurally equivalent** if the following holds:

- $\gamma > T = \{-\}$  iff  $\gamma' > T = \{-\}$ , and in that case  $T$  has meta-theory  $M$  in  $\gamma$  iff it does so in  $\gamma'$ ,
- $\gamma \gg l : S \rightarrow T = \_$  iff  $\gamma' \gg l : S \rightarrow T = \_$ ,

- whenever  $\gamma > T = \{\_\}$ , where  $\gamma >_T c : \_ = \_$  iff  $\gamma' >_T c : \_ = \_$ .

The intuition behind structural equivalence is that structurally equivalent theory graphs declare the same names: they have the same theories, the same constants, and the same links. It leaves open whether a constant of name  $s/c$  is declared or whether a constant  $c$  is imported via a structure  $s$ . It also leaves open whether a link is a structure or a view.

The value of structural equivalence is that it imposes no requirements on the foundation. Furthermore, structural equivalence is sufficiently strong an invariant for many applications such as indexing or cross-referencing. This is formalized in the following next theorem.

**Theorem 35.** *Assume two structurally equivalent theory graphs  $\gamma$  and  $\gamma'$ . Then for all theories  $S, T$  of  $\gamma$ :*

- $\gamma \triangleright_T \omega \quad \text{iff} \quad \gamma' \triangleright_T \omega,$
- $\gamma \triangleright \mu : S \rightarrow T \quad \text{iff} \quad \gamma' \triangleright \mu : S \rightarrow T.$

*Proof.* This follows by a straightforward induction on the derivations of well-formed terms and morphisms.  $\square$

In structurally equivalent theory graphs, the same constant might have different types. Semantic equivalence refines this:

**Definition 36.** Two structurally equivalent theory graphs  $\gamma$  and  $\gamma'$  are called **semantically equivalent** if the following holds:

- If  $\gamma > T = \{\_\}$ ,  $\gamma >_T c : \tau = \delta$ , and  $\gamma' >_T c : \tau' = \delta'$ , then  $\bar{\delta}^\gamma = \bar{\delta'}^{\gamma'}$  and  $\bar{\tau}^\gamma = \bar{\tau'}^{\gamma'}$ .
- For all  $\gamma \gg l : S \rightarrow T = \_$ , if  $\gamma \gg_l c \mapsto \delta$  and  $\gamma' \gg_l c \mapsto \delta'$ , then  $\bar{\delta}^\gamma = \bar{\delta'}^{\gamma'}$ .

Intuitively, if two theory graphs are semantically equivalent, then they have the same constant declarations and the same assignments. Another way to put it, is that the theory graphs are indiscernable in the following sense:

**Theorem 37.** *Assume two semantically equivalent theory graphs  $\gamma$  and  $\gamma'$  and a regular foundation. Then for all module declarations  $Mod$ :*

$$\triangleright \gamma, Mod \quad \text{iff} \quad \triangleright \gamma', Mod.$$

*Proof.* First of all, due to the structural equivalence,  $\gamma, Mod$  is clash-free iff  $\gamma', Mod$  is. Now assume a well-formedness derivation  $D$  for  $\triangleright \gamma, Mod$ . Let  $D'$  arise from  $D$  by replacing every occurrence of  $\gamma$  with  $\gamma'$ , and replacing the subtree of  $D$  deriving  $\triangleright \gamma$  with some derivation of  $\triangleright \gamma'$ . We claim that every subtree of  $D'$  is a well-formedness derivation for its respective root. Then in particular,  $D'$  is a well-formedness derivation for  $\triangleright \gamma', Mod$ . This is shown by induction on  $Mod$ . All induction steps are simple because in most rules

the theory graph only occurs as a fixed parameter. Those rules that “look into” the theory graph do so via the judgments given in Sect. 4.3, and the semantic equivalence of  $\gamma$  and  $\gamma'$  guarantees that these judgments agree up to normalization, and normalization is respected by a regular foundation.  $\square$

This provides systems working with MMT theory graphs with an invariant for foundation-independent and semantically indiscernible transformations. Systems maintaining theory graphs can apply such transformations to increase the efficiency of storage or lookup in a way that is transparent to other applications. Moreover, it provides an easily implementable criterion to analyze the management relevance of a change.

Of course, Def. 36 is just a sufficient criterion for semantic indiscernability. If a foundation adds equalities between terms, then theory graphs that are distinguished by Def. 36 become equivalent with respect to that foundation. But the strength of Def. 36 and Thm. 37 is that they are foundation-independent. Therefore, it can be implemented easily and generically.

The most important examples of semantical equivalence are reordering and flattening (see Sect. 6.4).

**Theorem 38.** *If  $\gamma$  and  $\gamma'$  are well-formed theory graphs that differ only in the order of modules, symbols, or assignments, then they are semantically equivalent.*

*Proof.* Clear since the elaboration judgments are insensitive to reorderings.  $\square$

However, note that not all reorderings preserve the well-formedness of theory graphs as defined here – there is a partial order on declarations that the linearization in the theory graph must respect, for example, constants must be declared before they are used. But Thm. 38 permits to generalize the definition of well-formed theory graphs as follows: A theory graph can be considered well-formed if there is some reordering for which it is well-formed. Using this relaxed definition is extremely valuable in practice because it permits applications to forget the order and thus to store theory graphs more efficiently. It is also relevant for distributed developments where keeping track of the order is often not feasible.

#### 6.4. Flattening

The representation of theory graphs introduced in the last section is geared towards expressing mathematical knowledge in its most general form and with the least redundancy: constants can be shared by inheritance (i.e., via imports), and terms can be moved between theories via morphisms. This style of writing mathematics has been cultivated by the Bourbaki group [Bou68, Bou74] and lends itself well to a systematic development of theories.

However, it also has drawbacks: Items of mathematical knowledge are often not where or in the form in which we expect them, as they have been generalized to a different context. For example, a constant  $c$  need not be explicitly

represented in a theory  $T$ , if it is induced as the image of a constant  $c'$  under some import into  $T$ .

In this section, we show that for every theory graph there is an equivalent flat one. This involves adding all induced knowledge items to every theory thus making all theories self-contained (but hugely redundant between theories). For a given MMT theory graph  $\gamma$ , we can view the flattening of  $\gamma$  as its semantics because flattening eliminates the specific MMT-representation infrastructure of structures and morphisms.

**Theorem 39.** *Given a fixed regular foundation, every well-formed theory graph is semantically equivalent to a flat one.*

*Proof.* Given a  $\triangleright \gamma$  the flat theory graph  $\gamma'$  is obtained as follows.

1. Theories

- For every  $\gamma > T = \{\cdot\}$ , there is a theory  $T$  in  $\gamma'$ . It has the same meta-theory (if any) in  $\gamma'$  as in  $\gamma$ .
- For every  $\gamma >_T c : \tau = \delta$ , the theory  $T$  of  $\gamma'$  contains a constant declaration  $c : \bar{\tau} = \bar{\delta}$ .

2. Links with definiens: For every  $\gamma \gg l : S \rightarrow T = \mu$ ,  $\gamma'$  contains a view  $l : S \rightarrow T = \mu$ .

3. Links with assignments:

- For every  $\gamma \gg l : S \rightarrow T = \{\cdot\}$ ,  $\gamma'$  contains a view from  $S$  to  $T$ . It has the same meta-morphism (if any) in  $\gamma'$  as in  $\gamma$ .
- For every  $\gamma \gg_l c \mapsto \delta$ , the view  $l$  of  $\gamma'$  contains a constant assignment  $c \mapsto \bar{\delta}$ .

It is easy to see that these declarations can be arranged in some way that makes  $\gamma'$  structurally well-formed. Furthermore, it is clear from the construction of  $\gamma'$  that  $\gamma'$  is flat and that  $\gamma$  and  $\gamma'$  are semantically equivalent. The only property that is not obvious is that  $\gamma'$  is well-formed. For that, we must show in particular that all assignments in all views in  $\gamma'$  satisfy the typing assumption of rule *ConAss*. This follows from the construction of  $\gamma$  and property (1) of regular foundations (which is the only property of regular foundations needed for this proof).  $\square$

*Example 40* (Continued from Ex. 1) The flattening of the theory graph of our running example contains the module declarations in Fig 20, where we omit all types for simplicity

Two features of MMT are not eliminated in the flattening: meta-theories and filtering.

Regarding meta-theories, the definitions and results in this section could be easily extended to elaborate meta-theories as well. For example, a meta-theory

```

 $e?\text{Monoid} \stackrel{f?\text{FOL}}{=} \{\text{comp, unit}\}$ 
 $e?\text{CGroup} \stackrel{f?\text{FOL}}{=} \{\text{mon/comp, mon/unit, inv}\}$ 
 $e?\text{CGroup}/\text{mon} : e?\text{Monoid} \rightarrow e?\text{CGroup} \stackrel{id_{f?\text{FOL}}}{=} \{$ 
     $\text{comp} \mapsto e?\text{CGroup}?\text{mon}/\text{comp},$ 
     $\text{unit} \mapsto e?\text{CGroup}?\text{mon}/\text{unit}\}$ 
 $e?\text{Ring} \stackrel{f?\text{FOL}}{=} \{$ 
     $\text{add/mon/comp, add/mon/unit, add/inv, mult/comp, mult/unit}\}$ 
 $e?\text{Ring}/\text{add} : e?\text{CGroup} \rightarrow e?\text{Ring} \stackrel{id_{f?\text{FOL}}}{=} \{$ 
     $\text{mon/comp} \mapsto e?\text{Ring}?\text{add/mon/comp},$ 
     $\text{mon/unit} \mapsto e?\text{Ring}?\text{add/mon/unit},$ 
     $\text{inv} \mapsto e?\text{Ring}?\text{add/inv}\}$ 
 $e?\text{Ring}/\text{mult} : e?\text{Monoid} \rightarrow e?\text{Ring} \stackrel{id_{f?\text{FOL}}}{=} \{$ 
     $\text{comp} \mapsto e?\text{Ring}?\text{mult/comp},$ 
     $\text{unit} \mapsto e?\text{Ring}?\text{mult/unit}\}$ 
 $e?\text{Ring}/\text{add/mon} : e?\text{Monoid} \rightarrow e?\text{Ring} = e?\text{CGroup}/\text{mon } e?\text{Ring}/\text{add}$ 

```

Figure 20: Module Declarations for the running example

$M$  can be reduced to a structure that instantiates  $M$  and has some reserved name. In fact, that is what we did in an earlier version of MMT [Rab08a]. However, this is not desirable because both humans and machines can use meta-theories to relate MMT theories to their semantics. In particular, constants of the meta-theory are often treated differently than the others; for example, their semantics might be hard-coded in an implementation.

Regarding filtering, the situation is more complicated. Imagine a constant declaration  $c : \tau = \top$  in the flat theory graph. This is particularly intuitive if we think of  $c$  as a theorem stating  $\tau$ . Then  $c : \tau = \top$  means that the theorem holds but its proof is filtered because it relies on a filtered assumption.

It is now a foundational question how to handle this case. One possibility is to delete the declaration of  $c$ . This is especially appealing from a type/proof theoretical perspective where constant declarations are what defines the existence of objects and their meaning. This community might argue that if the proof is filtered, then the theorem is useless because it can never be applied or verified. Consequently, it can just as well be removed. Another possibility is to replace  $c$  with the declaration  $c : \tau$ , i.e., to turn it into an axiom. This is appealing from a set/model theoretical perspective where constant declarations merely introduce names for objects that exist in the models. This community might argue that it is irrelevant whether the proof is filtered or not as long as we know that there is one.

In order to stay neutral to this foundational issues, we do not elaborate filtering. Instead, we leave all filtered declarations in the flattened signature and leave it to the foundation to decide whether they are used or not.

The most important practical aspect of the flattening in MMT is not its existence but that it can be applied incrementally. This is significantly more difficult. Consider a theory graph

$$\gamma_0, T \stackrel{M}{=} \{\vartheta_0, s : S = \{\sigma\}, \vartheta_1\}, \gamma_1.$$

We would like to flatten only the structure  $T/s$ . Then the structure can be replaced with a translated copy of the body of  $S$ .

For example  $c : \tau = \perp$  is translated to  $s/c : \tau' = \perp$ , where  $\tau'$  is the translation of  $\tau$ . In particular, in  $\tau'$  all names referring to constant of  $S$  must be prefixed with  $s$ . If  $s$  has an assignment  $c \mapsto \delta'$ , then the declaration is translated to  $s/c : \tau' = \delta'$ .

We obtain incrementality if structure declarations in  $S$  are not flattened recursively. This is possible in MMT. For example, for a structure  $r : R$  in the body of  $S$ , a structure  $s/r : R = S/r T/s$  can be added to  $T$  rather than adding all induced constants  $T?S/r/c$ . Individual assignments to structures can be flattened similarly.

## 7. Specific Foundations

To define a specific foundation, we need to define the judgments  $\gamma; \Upsilon \triangleright_T \omega \equiv \omega'$  and  $\gamma; \Upsilon \triangleright_T \omega : \omega'$ .

For a fixed theory graph, let  $<$  be the transitive closure of the relation “ $X$  has meta-theory  $Y$ ”. Then the foundational theories are the  $<$ -maximal ones; and for every other theory  $T$ , there is a unique foundational theory  $M$  with  $T < M$ , which we call **the foundational theory of  $T$** . A specific foundation is typically coupled with a certain foundational theory  $M$  and only defines  $\gamma \triangleright_T \omega \equiv \omega'$  and  $\gamma \triangleright_T \omega : \omega'$  for theories  $T < M$ . For example, a foundation for set theory could be coupled with the foundational theory ZFC.

Foundational theories and foundations can be given for a wide variety of formal languages. As examples, we give them for two very different languages: OPENMATH and LF.

### 7.1. OpenMath

OPENMATH [BCC<sup>+</sup>04] is used for the communication of set theory-based mathematical objects over the internet. OPENMATH content dictionaries correspond to MMT theories, so that MMT yields a module system for OPENMATH content dictionaries. Pure OPENMATH is an untyped language, in which  $\alpha$ -conversion of bound variables is the only non-trivial equality relation. Clearly, this foundation is very easy to implement.

The foundational theory for OPENMATH is empty because OPENMATH does not use any predefined constant names. Thus the standard content dictionaries can be introduced as MMT theories with that meta-theory. We can define a foundation for OPENMATH as follows.

Firstly,  $\gamma \triangleright_T \omega : \omega'$  holds iff one of the following holds:

1.  $\gamma \triangleright_T \omega$  and  $\omega' = \perp$ ,
2.  $\omega = \perp$  and  $\omega' = \perp$ .

To understand why this characterizes OPENMATH, consider how it affects the rules *Con* and *ConAss*. According to rule *Con*, a constant declaration  $c : \tau = \delta$  is only well-formed if  $\gamma \triangleright_T \delta : \tau$ . Thus, each of the above cases leads to one kind of constant declaration: The first case is used to define a constant to be equal to some term. The second case is used to declare undefined constants. All constants are untyped.

According to rule *ConAss*, an assignment  $c \mapsto \delta$  must satisfy that  $\delta$  is typed by the translation of the type of  $c$ . Since all constants are untyped, this is vacuous.

Secondly,  $\gamma \triangleright_T \omega \equiv \omega'$  is the smallest relation on structurally well-formed terms that

- is reflexive,
- is closed under substitution of equals,
- is closed under  $\alpha$ -renaming,
- respects normalization, i.e.,  $\gamma \triangleright_T \omega \equiv \bar{\omega}$ .

**Theorem 41.** *The foundation for OPENMATH is regular.*

*Proof.* All properties can be verified directly.  $\square$

Foundations for other untyped languages such as set theories can be defined similarly. The main difference is that significantly more complicated definitions of the (undecidable) equality judgment must be employed.

## 7.2. The Edinburgh Logical Framework (LF)

LF [HHP93] is a logical framework based on dependent type theory. Being a logical framework, it represents both logics and theories as LF signatures. MMT subsumes this approach by also representing LF as a (foundational) MMT theory. As for OPENMATH, MMT yields a module system for LF.

The foundational theory for LF is given by:

$$\text{LF} = \{\text{type}, \text{kind}, \text{lambda}, \text{Pi}\}.$$

`type` is the kind of all types. `kind` is the universe of kinds; it does not occur in concrete syntax for LF, but is needed as the MMT type of all well-formed LF kinds. `Pi` is the dependent type constructor, and `lambda` its introductory form. The application of MMT can be used as the eliminatory form of `Pi`.

If  $T = \text{LF}$ , only the typing judgment  $\gamma \triangleright_{\text{LF}} \perp : \perp$  holds. This is needed to make the untyped constants in the theory LF well-formed (see rule *ConAss*).  $\gamma \triangleright_{\text{LF}} \omega \equiv \omega'$  holds iff  $\bar{\omega} = \bar{\omega}'$ .

Otherwise, typing and equality are defined according to the LF type theory:

- For constants,  $\gamma \triangleright_T D?c : \tau$  holds if  $T < \text{LF}$  and  $\gamma >_D c : \tau = \_$ .
- For other terms,  $\gamma \triangleright_T \omega : \omega'$  holds if  $\bar{\omega}$  is a well-formed LF-term of type  $\bar{\omega}'$  or a well-formed LF-type family of kind  $\bar{\omega}'$ . The details are as in [HHP93] except that the rule for constants is not needed.
- $\gamma \triangleright_T \perp : \omega$  holds if  $\bar{\omega}$  is a well-formed LF-type or a well-formed LF-kind. This permits declarations of typed or kinded constants.

Similarly, the judgment  $\gamma \triangleright_T \omega \equiv \omega'$  is defined by the rules given in the properties (1) and (2) of Def. 25 and the equality rules for LF given in [HHP93].

**Theorem 42.** *The foundation for LF is regular.*

*Proof.* The properties (1) and (2) are built into the definition. Property (3) follows from the results in [HST94] after observing that every type-preserving mapping from  $S$  to  $T$  yields an LF signature morphisms from  $\bar{S}$  to  $\bar{T}$ . Here  $\bar{S}$  denotes the union of the bodies of all theories  $D$  with  $T \leq D < \text{LF}$ .  $\square$

Regular foundations for any pure type system and for other type theories can be given in the same way.

## 8. Web-Scalability

Because MMT documents are transparent to the semantics, they have been deliberately ignored so far. But documents play a central role for web-scalability because they permit the packaging and distribution of theory graphs. We will discuss them in Sect. 8.1. The basis for web-scalability is web standards-compliance, and we introduce the XML-based concrete, external syntax for MMT theory graphs in Sect. 8.2 and a URI-based concrete syntax for identifiers in Sect. 8.3. We will also define relative URI references that are indispensable for scalability. Finally, we describe in Sect. 8.4 the decomposition of MMT documents into sequences of atomic declarations, their incremental validation, and a basic query language for atomic document fragments.

### 8.1. Documents and Libraries

Recall that our syntax uses two-partite module identifiers  $g?I$ .  $g$  is a URI that identifies a package, called **document** in MMT. We use the syntax

$$\text{Doc} ::= g = \{\gamma\}$$

to declare a document  $g$  containing the theory graph  $\gamma$ . A document is called **primary** if all modules declared within  $\gamma$  have module identifiers of the form  $g?I$ . Non-primary documents arise when documents are aggregated dynamically using fragments from different documents, i.e., as the result of a search query; we call those **virtual** documents in [KRZ10].

Within MMT documents, we define two relaxations of the MMT syntax that are important for scalability and that can be easily elaborated into the official

syntax: relative identifiers and remote references. **Relative identifiers** and their resolution into the official identifiers are defined in Sect. 8.3. We speak of **remote references** if a document refers to a module that is declared in some other document. Technically, according to the rules of MMT, such a non-self-contained theory graph would be invalid. Therefore, we make documents with remote references self-contained by adding all referenced remote modules in some valid order at the beginning. This is always possible if there is no cyclic dependency between documents.

The semantics of remote references is well-defined because MMT identifiers are URIs and thus globally unique. However, they are not necessarily URLs and thus do not necessarily indicate physical locations from which the remote module could be retrieved. Therefore, we make use of a **catalog** that translates MMT URIs into URLs, which give the physical locations. This way applications are free to retrieve content from a variety of backends, such as file systems, databases, or local working copies, in a way that is transparent to the MMT semantics.

We call a collection of documents together with a catalog an **MMT library**. A library's document collection can be anything from a self-contained document to (the MMT-relevant subset of) the whole internet. The central component is the catalog that defines the meaning of identifiers in terms of physical locations. Adding a document to a library may include the upload of a physical document, but may also simply consist in adding some catalog entries.

Well-formedness of libraries is checked incrementally by checking individual documents when they are added. A document  $g = \{\gamma\}$  is **well-formed** relative to a library  $L$  if the following hold:

- If a module identifier declared in  $\gamma$  already exists in  $L$ , then the two modules must be identical.
- $\gamma_L$  is a well-formed theory graph where  $\gamma_L$  arises by prepending all remotely referenced modules according to their resolution in  $L$ .

It is easy to prove that if we only ever add well-formed documents to an initially empty library, all modules in the library can be arranged into a single well-formed theory graph. This can be realized, for example, by implementing  $L$  as a database that rejects the commit of ill-formed content (see Sect. 9). Thus, libraries provide a safe and scalable way of building large theory graphs.

### 8.2. XML-based Concrete Syntax

For the XML syntax, we build on the OMDOC format [Koh06], which already integrates some of the primitive notions of MMT including the MATHML 3 syntax for terms (interpreted as OPENMATH objects). In fact, the MMT data model and the XML syntax presented here will form the kernel of the upcoming version of the OMDOC format.

The XML grammar mostly follows the abstract grammar of MMT. Theory graphs are **omdoc** elements with **theory** and **view** elements as children. The

|   |   |
|---|---|
| $E(T?c)$  | $E^{\text{triple}}(T?c)$  |
| $E(x)$  | $<\text{m:ci}>x</\text{m:ci}>$  |
| $E(\top)$   | $\text{mmt}(\text{filtered})$   |
| $E(\omega^\mu)$   | $\begin{aligned} &<\text{m:apply}> \\ &\quad \text{mmt}(\text{morphism-application}) \\ &\quad E(\omega) \\ &\quad E(\mu) \\ &</\text{m:apply}> \end{aligned}$  |
| $E(@(\omega_1, \dots, \omega_n))$   | $<\text{m:apply}>E(\omega_1) \dots E(\omega_n)</\text{m:apply}>$  |
| $E(\beta(\omega_1; \Upsilon; \omega_2))$  | $\begin{aligned} &<\text{m:bind}> \\ &\quad E(\omega_1) \\ &\quad E(\Upsilon) \\ &\quad E(\omega_2) \\ &</\text{m:bind}> \end{aligned}$   |
| $E(\cdot)$  | (empty sequence)  |
| $E(\Upsilon, x[: \tau][= \delta])$  | $\begin{aligned} &E(\Upsilon) \\ &<\text{m:bvar}> \\ &\quad <\text{m:semantics}> \\ &\quad \quad <\text{m:ci name}=\text{"x"}> \\ &\quad \quad [<\text{m:annotation-xml base}=\text{"}\langle\!\langle \text{MMTURI} \rangle\!\rangle\text{"} \text{cd}=\text{"mmt" name}=\text{"type"}> \\ &\quad \quad \quad E(\tau) \\ &\quad \quad \quad </\text{m:annotation-xml}>] \\ &\quad \quad [<\text{m:annotation-xml base}=\text{"}\langle\!\langle \text{MMTURI} \rangle\!\rangle\text{"} \text{cd}=\text{"mmt" name}=\text{"value"}> \\ &\quad \quad \quad E(\delta) \\ &\quad \quad \quad </\text{m:annotation-xml}>] \\ &\quad </\text{m:semantics}> \\ &</\text{m:bvar}> \end{aligned}$ |
| $\langle\!\langle \text{MMTURI} \rangle\!\rangle = \text{http://cds.omdoc.org/omdoc/mmt.omdoc}$ |   |

Figure 21: XML Encoding of Terms

children of **theory** elements are **constant** and **structure** elements. And the children of **view** and **structure** elements are **conass** and **strass** elements. Both terms and morphisms are represented as strict content MATHML expressions.

The definition of the XML encoding  $E(\cdot)$  is given in Fig. 21, 22, and 23. The encoding of identifiers is given in Sect. 8.3. In the definition of the encoding, we assume that the following namespace bindings are in effect:

```
xmlns="http://www.omdoc.org/ns/omdoc"
xmlns:m="http://www.w3.org/1998/Math/MathML"
```

Moreover, we assume a special OPENMATH content dictionary with **cdbase** <http://cds.omdoc.org/omdoc/mmt.omdoc> and name **mmt** declaring the following symbols:

| $E(l)$                 | $E^{\text{triple}}(l)$   |
|------------------------|--|
| $E(id_T)$              | $\langle \text{m:apply} \rangle$<br>$mmt(\text{identity})$<br>$E^{\text{triple}}(T)$<br>$\langle / \text{m:apply} \rangle$       |
| $E(\mu_1 \dots \mu_n)$ | $\langle \text{m:apply} \rangle$<br>$mmt(\text{composition})$<br>$E(\mu_1) \dots E(\mu_n)$<br>$\langle / \text{m:apply} \rangle$ |

Figure 22: XML Encoding of Morphisms

| Name                              | Intuition    | Role        |
|-----------------------------------|--------------|-------------|
| <code>type</code>                 | :            | attribution |
| <code>value</code>                | =            | attribution |
| <code>filtered</code>             | T            | constant    |
| <code>identity</code>             | $id_T$       | application |
| <code>composition</code>          | $\mu\mu$     | application |
| <code>morphism-application</code> | $\omega^\mu$ | application |

These symbols are used to encode the MMT primitives discussed in Sect. 4. We write  $mmt(n)$  for the element

```
<m:csymbol base="http://cds.omdoc.org/omdoc/mmt.omdoc" module="mmt" name="n"/>
```

The encoding of the structural levels in Fig. 23 is straightforward.

The encoding of terms in Fig. 21 is similar to the encoding of OPENMATH objects in strict content MathML [ABC<sup>+</sup>10]. It differs in some minor respects:

- We use a `base` attribute to give the document URI (interpreted as the “content dictionary base” in MATHML 3) of `csymbols` and annotations. This is necessary because MATHML 3 does not provide a way to ascribe different CD bases to individual symbols except when format-specific mechanisms are defined by the format in which MATHML is embedded. For MMT, this mechanism is given by the `base` attribute and Def. 44.
- The `csymbol` element is used to refer to both symbols and modules. This is only necessary when encoding modular MMT theory graphs.
- Symbol and module names permit a larger set of characters. In particular, the forward slash character that we use for constructing theory paths is not allowed in names, which are restricted to NCNames (see [W3C98]). We do not normalize these away here and assume an omitted encoding step that eliminates the offending characters.
- We do not enclose OPENMATH objects in `math` elements. This is redundant due to the use of XML namespaces.

Alternatively, we could use the XML encoding of OPENMATH objects defined by the OPENMATH 2 standard [BCC<sup>+</sup>04].

|            |   |  |
|------------|---|--|
| Document   | $E(g = \{Mod_1, \dots, Mod_n\})$                        | <pre>&lt;omdoc base="g"&gt;   E(Mod_1) ... E(Mod_n) &lt;/omdoc&gt;</pre>   |
| Theory     | $E(T \stackrel{[M]}{=} \{S_1, \dots, S_n\})$            | <pre>&lt;theory name="T"   [meta="E^URI(M)"]&gt;   E(S_1) ... E(S_n) &lt;/theory&gt;</pre>   |
| View       | $E(l : S \rightarrow T \stackrel{[\mu]}{=} \{\sigma\})$ | <pre>&lt;view name="l" from="E^URI(S)"   to="E^URI(T)"&gt;   [&lt;include&gt;E(\mu)&lt;/include&gt;]   E(\sigma) &lt;/view&gt;</pre>           |
|            | $E(l : S \rightarrow T = \mu)$                          | <pre>&lt;view name="l" from="E^URI(S)"   to="E^URI(T)"&gt;   &lt;definition&gt;E(\mu)&lt;/definition&gt; &lt;/view&gt;</pre>                   |
| Constant   | $E(c : \tau)[= \delta]$                                 | <pre>&lt;constant name="c"&gt;   [&lt;type&gt;E(\tau)&lt;/type&gt;]   [&lt;definition&gt;E(\delta)&lt;/definition&gt;] &lt;/constant&gt;</pre> |
| Structure  | $E(s : S \stackrel{[\mu]}{=} \{\sigma\})$               | <pre>&lt;structure name="s"   from="E^URI(S)"&gt;   [&lt;include&gt;E(\mu)&lt;/include&gt;]   E(\sigma) &lt;/structure&gt;</pre>               |
|            | $E(s : S = \mu)$  | <pre>&lt;structure name="s"   from="E^URI(S)"&gt;   &lt;definition&gt;E(\mu)&lt;/definition&gt; &lt;/structure&gt;</pre>                       |
| Assignment | $E(Ass_1, \dots, Ass_n)$                                | $E(Ass_1) \dots E(Ass_n)$  |
|            | $E(c \mapsto \omega)$                                   | <pre>&lt;conass name="E(c)"&gt;   E(\omega) &lt;/conass&gt;</pre>  |
|            | $E(s \mapsto \mu)$                                      | <pre>&lt;strass name="E(s)"&gt;   E(\mu) &lt;/strass&gt;</pre>   |

Figure 23: XML Encoding of Structural Levels

|        |                             |   |
|--------|-----------------------------|---|
| triple | $E^{\text{triple}}(g?I?I')$ | $\langle \text{m:csymbol base}=\text{"}g\text{" cd}=\text{"}I\text{"} \rangle I' \langle \text{/csymbol} \rangle$ |
|        | $E^{\text{triple}}(g?I)$    | $\langle \text{m:csymbol base}=\text{"}g\text{" cd}=\text{"}I\text{"} \rangle$                                    |
| URI    | $E^{\text{URI}}(g?I?I')$    | $g?I?I'$  |
|        | $E^{\text{URI}}(g?I)$       | $g?I$   |

Figure 24: XML Encoding of Identifiers

### 8.3. URI-based Addressing

As defined in the MMT grammar, an **absolute identifier** of an MMT knowledge item is a document URI  $G$ , a module identifier  $G?M$ , or a symbol identifier  $G?M?S$ . It is convenient to unify these three cases by assuming  $M = \varepsilon$  and/or  $S = \varepsilon$  if the respective component is not present. Then absolute references are always triples  $(G, M, S)$ .

Similarly, a **relative identifier** is a triple  $(g, m, s)$ .  $g$  is a relative document reference, i.e., a URI reference as defined in RFC 3986 [BLFM05] but without query or fragment. Note that this includes the case  $g = \varepsilon$ .  $m$  and  $s$  are usually of the form  $I$ , i.e., slash-separated (possibly empty) sequences of non-empty names. For completeness, we mention that MMT also permits  $m$  and  $s$  to be relative: If  $g = \varepsilon$ ,  $m$  may be of the form  $/I$ , which is a module reference that is interpreted relative to the current module; and if  $g = m = \varepsilon$ ,  $s$  may also be of the form  $/I$ , which is a symbol reference that is interpreted relative to the current symbol.

Since absolute and relative identifiers are both triples, they can be encoded in the same way. There are two different ways to encode MMT identifiers, which are given in Fig. 24. When identifiers occur as XML elements, we use  $E^{\text{triple}}(-)$  to obtain the triple of document, module, and symbol name. If they occur as attribute values, we use  $E^{\text{URI}}(-)$  to obtain a string.

This triple-based addressing model takes up an idea (called “reference by context”) from OMDOC 1.1 that was dropped in OMDOC 1.2 because its semantics could not be rigorously defined without the MMT concepts. In particular triples  $(g, T, c)$  correspond to the  $(\text{cbase}, \text{cd}, \text{name})$  triples of the OPENMATH standard [BCC<sup>+</sup>04].

When names occur in attribute values, we encode identifiers as URI strings using  $?$  as a separating character. In this encoding, concrete and abstract syntax are identical. Absolute and relative identifiers are encoded as URIs and URI references, respectively. We adopt the convention that trailing but not leading  $?$  characters can be dropped. For example, we encode

- $(g, m, \varepsilon)$  as  $g?m$ ,
- $(\varepsilon, m, s)$  as  $?m?s$ ,
- $(\varepsilon, \varepsilon, s)$  as  $??s$ ,

This encoding can be parsed back uniquely into triples.

**Definition 43** (Relative URI Resolution). The **resolution** of relative identifier  $R = (g, m, s)$  is defined relative to an absolute identifier  $B = (G, M, S)$ , which serves as the base of the resolution. The result is the following absolute identifier:

$$resolve(B, R) := \begin{cases} (G + g, m, s) & \text{if } g \neq \varepsilon \\ (G, M + m, s) & \text{if } g = \varepsilon, m \neq \varepsilon \\ (G, M, S + s) & \text{if } g = m = \varepsilon, s \neq \varepsilon \\ (G, M, S) & \text{if } g = m = s = \varepsilon \end{cases}$$

where  $G + g$  denotes the resolution of the URI reference  $g$  relative to the URI  $G$  as defined in RFC 3986 [BLFM05]. Furthermore,  $M + m$  resolves  $m$  relative to  $M$ : If  $m = /m'$ , then  $M + m$  arises by appending  $m'$  to  $M$ ; otherwise  $M + m = m$ .  $S + s$  is defined accordingly.

The above definition yields the ill-formed result  $(G, \varepsilon, s)$  when resolving a symbol level reference  $R = (\varepsilon, \varepsilon, s)$  against a document level base  $B = (G, \varepsilon, \varepsilon)$ . We forbid that pathological case, which would correspond to a symbol being declared outside a theory.

To resolve relative identifiers within a document, we need the following:

**Definition 44** (Base URI). Let  $\gamma$  a theory graph, then we define a **base URI** for MMT expressions occurring in  $\gamma$ :

1. The base of a module declaration or a remote reference to a module is the URI of the containing document.
2. The base of symbol declaration is the URI of the containing theory.
3. The base of an assignment to a symbol is the URI of the codomain theory.
4. If  $\mu$  is a morphism with domain  $S$ , then the bases of  $\omega$  in  $\omega^\mu$ , and  $\mu'$  in  $\mu' \mu$ , are  $S$ .
5. In all other cases, the base of an expression is the base of the parent node in the syntax tree.

Furthermore, in the XML encoding of documents, authors may override the base reference by using an attribute `base`. This attribute may be present on any XML element occurring in the encoding, and all relative MMT URIs are interpreted relative to the closest enclosing `base` attribute. Thus, `base` is similar to the `xml:base` attribute except that its value is an MMT URI and that relative identifiers are resolved according to Def. 43. Note that the value of `base` may itself be relative – this implies that there is no semantic difference between an empty and an omitted `base` attribute.

URIs are the main data structure needed for cross-application scalability, and our experience shows that they must be implemented by almost every peripheral system, even those that do not implement MMT itself. Already at this point, we

had to implement them in SML [RS09], Javascript [GLR09], XQuery [ZKR10], Haskell (for Hets, [MML07]), and Bean Shell (for a jEdit plugin) – in addition to the Scala-based reference API presented in Sect. 9.1.

This was only possible because MMT-URIs constitute a well-balanced trade-off between mathematical rigor, feasibility, and URI-compatibility: In particular, due to the use of the two separators / and ? (rather than only one), they can be parsed locally, i.e., without access to or understanding of the surrounding MMT document.

#### 8.4. An API for Knowledge Management

We use the concepts introduced above to specify an API for MMT documents. It is designed around MMT library operations that add or retrieve atomic URI-identified knowledge items. It is easy to implement and can be quickly integrated with different front and back ends ranging from HTTP servers to interactive editors.

*Adding Knowledge Items.* MMT fragments are added during the validation algorithm. In general, we distinguish three levels of **validation** with varying strictness strictness. Plain XML validation is quick but cannot guarantee MMT-well-formedness. The latter is guaranteed by structural validation, which implements the inference system given in this paper. Structural validation uses a default foundation, in which typing and equality of terms is always true. Foundation-relative validation refines structural validation by additionally checking typing and equality constraints by using a plugin for specific foundations.

Structural validation of MMT theory graphs can be implemented by decomposing the theory graph into a sequence of atomic declarations that are validated and added incrementally. Except for structures, every symbol or assignment is an atomic declaration. Declarations of documents, theories, views, and structures, are atomic if the body is empty. For example, a view is decomposed into the declaration of an empty view and one declaration for each assignment.

The MMT inference system is designed such that structural validation is possible. In particular, later atomic declarations can never invalidate earlier ones.

*Retrieving Knowledge Items.* To retrieve knowledge items from a library, we use **atomic queries** given in Fig. 25. These take an MMT URI and return the MMT declaration identified by the elaboration judgments.

Atomic queries permit not only the retrieval of all declarations of the original documents in the library, but also of all induced declarations. Note that there are two ways to combine a structure URI  $g?T/s$  and a constant  $c$ : The query  $g?T/s?c$  retrieves an assignment provided by  $s$  for  $c$  (defaulting to  $c \rightarrow g?T?s/c$  if there is none); and the query  $g?T?s/c$  retrieves the induced constant declaration of  $T$ .

*Example 45* (Continued from Ex. 21) Examples for atomic queries were already indicated in Ex. 21.

| URI   | Definedness condition  | Result   |
|-------|--|--|
| $g$   | $g = \{\gamma\}$ in $L$  | $g = \{\gamma\}$                                     |
| $T$   | $\gamma_L > T = \{\vartheta\}, [M \hookrightarrow T]$                    | $T \stackrel{[M]}{=} \{\vartheta\}$                  |
| $l$   | $\gamma_L \gg l : S \rightarrow T = \{\sigma\}, [\mu \hookrightarrow l]$ | $l : S \rightarrow T \stackrel{[\mu]}{=} \{\sigma\}$ |
| $l$   | $\gamma_L \gg l : S \rightarrow T = \mu$                                 | $l : S \rightarrow T = \mu$                          |
| $T?c$ | $\gamma_L >_T c : \tau = \delta$   | $c : \tau = \delta$                                  |
| $l?c$ | $\gamma_L \gg_l c \mapsto \delta$  | $c \mapsto \delta$                                   |

Figure 25: Atomic Queries

| atomic query                             | returns:  | comment   |
|--|---|---|
| $e?\text{CGroup}/\text{mon}/\text{comp}$ | $\text{mon}/\text{comp} \mapsto e?\text{CGroup}?\text{mon}/\text{comp}$ | the default assignment for lack of an explicit assignment |
| $e?\text{CGroup}?\text{mon}/\text{comp}$ | $\text{mon}/\text{comp} : \tau^{e?\text{CGroup}/\text{mon}} = \perp$    | where $\tau$ is as in Ex. 21                              |

Atomic queries are relatively easy to implement and provide a sufficient interface for higher knowledge management layers to implement many additional services. For example, we can use them to implement **local validation**. Given a library  $L$  and an implementation of atomic queries, we can validate documents and document fragments relative to  $L$  without having to read all of  $L$ . Instead, the respective atomic query is sent to  $L$  whenever a reference to an unknown knowledge item is encountered.

Moreover, if we are interested in structural validation only, it is sufficient to know only the type of a query result (i.e., theory, view, constant, structure, assignment to constant, assignment to structure). This information can be precomputed and cached by the library.

Atomic queries also yield an easy implementation of **flattening** because they already return all declarations of induced constants and assignments that occur in the flattened theory graph. Therefore, to implement flattening, we only have to know the URIs of the induced declarations. Again this information can be cached by the library, and applications can aggregate the flattened theory graph without having to implement MMT.

*Foundations as Plugins.* An implementation of MMT should provide a plugin interface for foundations. Every plugin must identify a theory  $M$ , and implement functions that decide or (attempt to prove) instances of the typing and equality judgments for any theory  $T < M$ . Foundations should be regular, and due to Lem. 26, they only need to consider flat instances of these judgments. Moreover, due to Thm. 39, they can assume that  $T$  is flat. Thus, existing implementations of formal systems for  $M$  can easily be reused to obtain plugins for the corresponding foundational theory.

## 9. Implementations

The design of the MMT language has been driven in a tight feedback loop between theoretical analysis of knowledge structures and practical implementation efforts. In particular, we have evaluated the space of possible module systems along the classifications developed in Sect. 2 in terms of expressivity, computational tractability, and scalability in a variety of case studies. The most relevant one for MMT is the logic atlas in the LATIN project [KMR09, CHK<sup>+</sup>11b], where we are attempting a modular development of logics and inference systems currently used in mathematical/logic-based software systems with a focus on concept sharing and trans-logic interoperability. These efforts led to three implementations, which we will present here. All of them are open source, and can be obtained from the authors.

### 9.1. The MMT Reference Implementation

The MMT implementation [Rab08b] provides a Scala-based [OSV07] (and thus fully Java-compatible) open-source implementation for the API from Sect. 8.4. The core of the implementation acts as a library with atomic add and retrieve methods. XML documents are decomposed, validated, and added incrementally, and retrieval of document fragments is implemented via atomic queries.

The **validation** algorithm provides a plugin interface for foundations. Every plugin must identify a foundational theory  $M$ , and implement functions that decide or (attempt to prove) instances of the typing and equality judgments for any theory  $T < M$ .

Foundations should be regular, and due to Lem. 26, they only need to consider flat instances of these judgments. Moreover, due to Thm. 39, they can assume that  $T$  is flat. Thus, existing algorithms and implementations for the flat case can be reused, and the MMT reference implementation adds a module system to them. Currently, one such plugin exists for the foundation for LF from Sect. 7.

As a by-product of validation, a **relational representation** of the validated document is generated, which corresponds to an ABox in the MMT ontology. The individuals of this ontology are the valid MMT URIs, and the relations between them include for example “Constant  $c_1$  occurs in the type of constant  $c_2$ .” or “View  $v$  has domain  $S$ .”. This information is cached, and the implementation includes a simple relational query language. The combination of atomic queries and relational queries is a simple but powerful interface to MMT libraries.

The library component can be combined with various back and front ends. The back ends implement the **catalog** that translates MMT URIs into physical locations. The current implementation includes back ends that retrieve documents from remote MMT libraries via HTTP or from local working copies of repositories via file system access. This catalog is fully transparent to the library component.

The front ends provide users and systems access to the library. The current implementation includes a **shell** and a **web server** front end. The shell is scriptable and can be used to explicitly retrieve, validate, and query MMT

documents. The web server is implemented using the Lift web framework for Scala [Pee07]. The interaction with the web server proceeds like with the shell except that input and output are passed via HTTP. In particular, the web server can easily be run as a local proxy that provides MMT functionality and file system abstraction to local applications. An example instance of the web server is serving the content of the TNTBASE repository of the LATIN project [KMR09].

In fact, the MMT language is significantly larger than presented here. Going beyond the scope of this paper, it also provides an OMDOC-style **notation language** with a simple declarative syntax to define renderings of MMT content in arbitrary human- or machine-oriented formats; see [KMR08] for an overview of a precursor of the MMT notation system. Notations can be grouped into styles, which are themselves subject to the MMT module system. The web server mentioned above can serve documents as XHTML with presentation MATHML and integrates the JOBAD technology for **interactive browsing** we presented in [GLR09]. Another use of notations is as a fast way of translating MMT into system's concrete input syntax so that MMT can serve as an interchange language.

### *9.2. TNTbase – a Scalable MMT-Compliant Database*

The TNTBASE system [ZK09] is an open-source versioned XML **database** developed at Jacobs University. It was obtained by integrating Berkeley DB XML [Ora10] into the Subversion Server [Apa00], is intended as a basis for collaborative editing and sharing XML-based documents, and integrates versioning and access of document fragments. We have extended TNTBASE with an MMT **plugin** that makes it MMT-aware [KRZ10, ZKR10].

The most important aspect of this plugin is validation-upon-commit. Using the `tntbase:validate` property, folders and files can be configured to require validation. Thus, users can choose between no, XML-based, structural, or foundational validation of MMT files. Since the commit of ill-formed files can be rejected, TNTBASE can guarantee that it only contains well-formed documents. Thus, other systems can use MMT-enriched TNTBASE for the long term storage of their system libraries and can trust in the correctness of documents retrieved from the database.

Moreover, the plugin computes the relational representation of a committed well-formed document, which TNTBASE stores as an XML file along with every MMT file. TNTBASE exposes the relational representation of MMT documents via an XQuery interface, and we have implemented a variety of custom queries in an XQuery module that is integrated into TNTBASE. TNTBASE indexes the files containing the relational representation so that such queries scale very well. For example, our XQuery module includes a function that computes the transitive closure of the structural dependency relation between MMT modules and dynamically generates a self-contained MMT document that includes all dependencies of a given module. Even for small libraries and even if TNTBASE runs on a remote server, this query outperforms the straightforward implementation based on local files.

Moreover, using the virtual documents of TNTBASE, such generated documents are editable; see [ZK10] for details. TNTBASE keeps track of how a document was aggregated and propagates the necessary patches when a changed version of the virtual document is committed.

### 9.3. Twelf – an MMT-Compliant Logical Framework

The MMT implementation from Sect. 9.1 starts with a generic MMT implementation and adds a plugin for a specific formal language  $F$ . Alternatively, an invasive implementation is possible, which starts with an implementation of  $F$  and adds the MMT module system to it. Such implementations are restricted to theory graphs with a single foundational theory for  $F$ , but can reuse special features for  $F$  such as user interfaces and type inference. We have implemented this for LF as well [RS09] using the Twelf implementation [PS99] of LF.

The effect of adding MMT to Twelf is that Twelf becomes a tool for authoring theory graphs with LF as the single foundational theory. A major advantage of this approach is that authors can benefit from the advanced Twelf features, in particular infix parsing, type reconstruction, and implicit arguments. This implementation was used successfully to generate large case studies of MMT theory graphs in [DHS09], [HR11], and [IR11].

Twelf also supports several advanced language features that are part of MMT but were not mentioned in this paper. In particular, this includes nested theories and unnamed imports between theories and links. Furthermore, fixity and precedence declarations of Twelf are preserved as MMT notations that are used when rendering the MMT theory graph.

Twelf can produce MMT documents in XML syntax from its input that are guaranteed to be well-formed. In [CHK<sup>+</sup>11c], we showed how logics written in Twelf can be exported in MMT concrete syntax and imported into and used in the Hets system [MML07].

## 10. Related Work

In this section, we survey the state of the art in module systems for formal languages using the terminology developed in Sect. 2 and relate MMT to them. Fig. 26 gives an overview of the discussed systems.

*Mathematical Language.* Even though mathematical knowledge can vary greatly in its presentation as well as its level of formality and rigor, there is a level of deep semantic structure that is common to all forms of mathematics. This large-scale structure of mathematical knowledge is much less apparent than that of formulas and is usually implicit in informal representations. Experienced mathematicians are nonetheless aware of it, and use it for navigating in and communicating mathematical knowledge.

Much of this structure can be found in networks of theories such as those in a monograph “Introduction to Group Theory” or a chapter in a textbook. The relations among such theories are described in the text, sometimes supported

|                                    | OpenMath (CD) | OMDoc (theory) | OBJ (theory) | ASL (specification) | dev. graphs (node) | CASL (specification) | IMPS (theory) | PVS (theory) | Isabelle (locale) | Nuprl | Coq (module type) | Agda (module) | SML (signature) | Java/Scala (class) | MMT (theory) | Twelf (signature) | Internalized |
|------------------------------------|---------------|----------------|--------------|---------------------|--------------------|----------------------|---------------|--------------|-------------------|-------|-------------------|---------------|-----------------|--------------------|--------------|-------------------|--------------|
| formality <sup>1</sup>             | p             | p              | c            | c                   | c                  | c                    | c             | c            | c                 | c     | c                 | c             | c               | c                  | c            | c                 |              |
| packages <sup>2</sup>              | l             | l              | s            |                     | p                  | s                    | p             | s            |                   | p     | p                 | s             | l               | l                  | l            |                   |              |
| package imports <sup>3</sup>       | o             | o              |              |                     | c                  |                      | c             | c            |                   | o     | c                 |               | o               | o                  | o            |                   |              |
| named inheritance <sup>4</sup>     |               | s              |              |                     |                    |                      |               | s            | i                 | i     | i                 | i             | i               | i                  | i            | i                 |              |
| instantiation <sup>5</sup>         |               | iit            |              |                     |                    |                      |               | fep          | fep               | iit   | fep               | iit           | fep             | fep                | iit          |                   |              |
| renaming                           |               |                |              |                     |                    |                      |               |              |                   |       | +                 |               |                 |                    |              | +                 |              |
| hiding <sup>6</sup>                |               |                |              |                     |                    |                      |               |              |                   | s     | s                 | s             | s               | f                  |              |                   |              |
| unnamed inheritance <sup>4</sup>   | i             | s              | a            | a                   | i                  | a                    | i             | s            | i                 |       | i                 | s             |                 | i                  |              |                   |              |
| diamond semantics <sup>7</sup>     | i             | i              |              |                     | i                  |                      | a             | a            | e                 |       | d                 | i             |                 | i                  |              |                   |              |
| name clash resolution <sup>8</sup> | q             | i              |              |                     | i                  |                      | q             | e            | e                 |       | s                 | i             |                 | q                  |              |                   |              |
| instantiation <sup>5</sup>         | fep           | fet            | fet          | iep                 |                    | iit                  | fep           |              | fep               |       | fep               | iit           |                 |                    |              |                   |              |
| renaming                           |               |                |              |                     | +                  |                      |               |              |                   |       |                   |               |                 |                    | +            |                   |              |
| hiding <sup>6</sup>                | f             | c              | c            | s                   |                    | s                    |               |              | s                 |       | s                 | s             |                 |                    |              |                   |              |
| realizations as objects            |               |                |              |                     |                    |                      |               |              |                   | +     |                   | +             | +               | +                  | +            | +                 |              |
| grounded realizations              |               |                |              |                     |                    |                      |               |              | +                 | +     |                   | +             | +               | +                  | +            | +                 |              |
| views/functors                     | +             | +              | +            | +                   | +                  | +                    | +             | +            | +                 | +     |                   | +             | +               | +                  | +            | +                 |              |
| higher-order                       |               |                |              |                     |                    |                      |               |              |                   |       |                   |               | +               |                    | +            |                   |              |
| translations <sup>9</sup>          |               |                | y            | y                   |                    |                      |               |              | e                 |       | e                 | e             | ye              | ye                 | e            |                   |              |
| semantics <sup>10</sup>            |               | m              | m            | m                   | m                  | e                    | m             | e            |                   | e     | e                 | e             | e               | e                  | e            |                   |              |
| internalized <sup>11</sup>         |               |                |              |                     |                    |                      |               |              | *                 | *     | *                 |               |                 | +                  |              |                   |              |
| logic-independent                  |               | +              | +            | +                   | +                  |                      |               | +            |                   |       |                   |               |                 | +                  | +            |                   |              |
| foundation-independent             | +             | +              |              |                     |                    |                      |               |              |                   |       |                   |               |                 | +                  |              |                   |              |
| URIs as identifiers                | +             | +              |              |                     |                    |                      |               |              |                   |       |                   |               | +               | +                  | +            |                   |              |
| XML syntax                         | +             | +              |              |                     |                    |                      |               |              |                   |       |                   |               | +               |                    |              |                   |              |

<sup>1</sup>p = presentation, c = content

<sup>2</sup>p = physical, l = logical, s = single package

<sup>3</sup>o = open, c = closed

<sup>4</sup>i = interspersed, s = separated, a = axiom-inheritance

<sup>5</sup>i/f = interfaced/free, e/i = explicit/implicit, t/p = total/partial

<sup>6</sup>s = simple, c = complex, f = filtering

<sup>7</sup>i = identify, d = distinguish, a = identify iff instantiations agree, e = error

<sup>8</sup>j = overload/identify, q = qualified names, s = shadowing, e = error

<sup>9</sup>explicit syntax for the translation along views/functors: y = syntactic, e = semantic

<sup>10</sup>m = model theory, e = elaboration

<sup>11</sup>+ = internalized, \* = additionally an internalized module system as in last column

Figure 26: Features of Module Systems

by mathematical statements called “representation theorems”. We can observe that mathematical texts can only be understood with respect to a particular mathematical context given by a theory which the reader can usually infer from the document, e.g., from the title or the specialization of the author. The intuitive notion of meta-theory is well-established in mathematics, but again

it is mainly used informally. Formal definitions are found in the area of logic where a logic is used as the meta-language of a logical theory.

Mathematical theories have been studied by mathematicians and logicians in the search of a rigorous foundation for mathematical practice. They have usually been formalized as collections of symbol declarations and axioms. Mathematical reasoning often involves several related mathematical theories, and it is desirable to exploit these relationships by moving theorems between theories. The first systematic, large-scale applications of this technique in mathematics are found in the works by Bourbaki [Bou68, Bou74], which tried to prove every theorem in the theory with the smallest possible set of axioms.

This technique was formalized in [FGT92], which introduced the *little theories* approach. Theories are studied as formal objects. And structural relationships between them are represented as theory morphisms, which serve as conduits for passing information (e.g., definitions and theorems) between theories (see [Far00]).

*Web Scale Languages.* The challenge in putting mathematics on the World Wide Web is to capture both notation and meaning in a way that documents can utilize the human-oriented notational forms of mathematics and provide machine-supported interactions at the same time. The W3C recommendation for mathematics on the web is the MATHML language [ABC<sup>+</sup>10]. It provides two sublanguages: **presentation** MATHML permits the specification of notations for mathematical formulas, and **content** MATHML is geared towards specifying the meaning in a machine-processable way. The latter is structurally equivalent to OPENMATH. In particular, both formats represent the structure of mathematical formulas as OPENMATH objects, i.e. tree-like expressions built up from constants, variables, and primitive data types via function applications and bindings.

MMT constants correspond to symbols in MATHML and OPENMATH and MMT theories to **content dictionaries** (CDs). CDs are machine-readable and web-accessible documents that provide a very simple way to declare mathematical objects for the communication over the WWW and attach meaning to them. Meaning can be expressed in the form of axioms or types given as OPENMATH objects representing logical formulas or in the form informal mathematical text.

OPENMATH provides a certain communication safety over traditional mathematics: It can no longer be the case that the author writes  $\mathbb{N}$  for the set of natural numbers with 0, and the reader understands the set of natural number without 0, as the two notions of “natural numbers” — even though presented identically — are represented by different symbols (probably from different CDs). Thus, the service offered by the OPENMATH/MATHML approach is one of disambiguation as a base for further machine support.

In MMT terms, the productions for constants, variables, application, and binding correspond closely to OPENMATH. MMT adds morphism application and the special term  $\top$ , and we omit the primitive data types. We use typed and defined variables in analogy to MMT constant declarations and do not use the attributions of OPENMATH.

OpenMath CDs enable formula disambiguation and web scale communication, but the lack of machine-understandable intra-CD knowledge structure and inter-CD relations preclude higher-level machine support. Therefore, OMDoc [Koh06] represents mathematical knowledge at the levels of objects, statements, theories, and documents: OPENMATH and content MATHML are subsumed to represent objects. Statements are symbols, axioms, definitions, theorems, proofs and occur as declarations within theories. Moreover, theories may declare unnamed, interspersed, free instantiations, and structured theory morphisms can be declared as in development graphs. Documents provide a basic content-oriented infrastructure for communication and archival.

Syntactically, OMDoc and OPENMATH are distinguished from purely formal representation languages by the fact that all formal mathematical elements of the language can be augmented or replaced by natural language text fragments. Semantically, they differ because they do not supplement the formal syntax with a formal semantics.

Some implementations of purely formal representation languages have made use of XML, OPENMATH/MATHML, or OMDOC as primary or secondary representation formats. For example, Mizar [TB85] uses XML as the primary internal format, and Matita [ACTZ06] uses content and presentation MATHML; Coq [CH88, BC04] provides an OMDoc export, and Isabelle [Pau94] a partial XML export. Web-scale languages can in principle serve as standardized interchange formats between such systems. Some examples of interoperability mediated by OMDOC and OPENMATH are [CHK<sup>+</sup>11c, CO01, HR09]. But applications have so far been limited due to the lack of an interchange format with a standardized semantics.

MMT provides such a semantics. It keeps OMDoc's leveled representation but restricts attention to a subset for which a formal semantics can be developed. Syntactically, the main addition of MMT is the use of named imports and of theory morphisms as objects.

OPENMATH and OMDOC use URIs [BLFM05] to identify symbol by triples of symbol name, CD id, and CD base. The CD base is a URI acting as a namespace identifier, which corresponds to the triples in MMT identifiers. But the formation of OPENMATH URIs is only straightforward via the one-CD-one-file restriction imposed by OPENMATH, which is too restrictive in general. MMT is designed such that all knowledge items have canonical URIs. Moreover, the formation of symbol URIs in OPENMATH and OMDOC uses the *fragment* components of URIs. Therefore, fragment access does not scale well because clients have to download a complete document and then execute the fragment access locally. MMT avoids this by using the *query* component of the URI.

*Algebraic Specification Languages.* In algebraic specification, theories are used to specify the behavior of programs and software components, and realizations (theory morphisms in MMT) are used to enable reuse of components (structures in MMT) and to formalize refinements of specifications (views in MMT).

In this setting, implementations can be regarded as refinements into executable specifications, which we have called *grounded realizations*. This ap-

proach naturally leads to a regime of specification and implementation co-development, where initial, declarative specifications are refined to take operational issues into account. Implementations are adapted to changing specifications, and verification conditions and their proofs have to be adapted as programming errors are found and fixed. This has been studied extensively, and a number of systems have been developed. We will discuss OBJ [GWM<sup>+</sup>93], ASL [SW83, ST88], CASL [CoF04, MML07], and development graphs [AHMS99, MAH06] as representative examples.

**OBJ** refers to a family of languages based on variants of sorted first-order logic. It was originally developed in the 1970s based on the Clear programming language and pioneered many ideas of modular specifications, in particular the use of initial model semantics [GTW78]. The most important variant is OBJ3; Maude [CELM96] is a closely related system based on rewriting logic. OBJ is a single-package system. Theories and views are similar to MMT. OBJ permits unnamed imports without instantiation and with identify-semantics, and named imports with interfaced, implicit, and total instantiations. All imports are separated. Named imports can be instantiated with views, but more complex realizations cannot be formed.

**ASL** is a generic module system over an arbitrary institution [GB92] with a model theoretical semantics. Similar to institutions, the focus is on abstract modeling rather than concrete syntax. Modules are called “specifications” and are formed using the operations of union (which corresponds to concatenation of theory bodies in MMT), imports, and complex hiding (which was introduced by ASL). Imports between specifications are unnamed and do not use instantiations but only axiom-inheritance and renaming. Unnamed views are used to express refinement theorems. We gave a representation of ASL in MMT in [CHK<sup>+</sup>11a], which uses an extension of MMT to accommodate hiding.

The **development graph** language is an extension of ASL specifically designed for the management of change. The central data structure are theory graphs of theories and two kinds of links, which correspond to the ones in MMT. (Global) “definitional links” are unnamed imports like in ASL and provide axiom-inheritance; (global) “theorem links” are partial views where the missing instantiations are treated as proof obligations that are to be discharged by theorem proving systems. ASL style hiding is supported by *hiding links*. The Maya system [AHMS02] implements development graphs for first-order logic. Like the MMT implementation and contrary to most other systems discussed here, Maya does not flatten the specification while reading it in. Thus, the modular information, in particular the theory graph, is available in the internal data structures. This is much more robust against changes in the underlying modules and provides a good basis for theorem reuse and management of change.

The development graph calculus uses local links. From the MMT perspective, a local link is a link which filters all but the local constants of its domain. A global theorem theorem link can be decomposed into a set of commuting local theorem links. By finding these local theorem links individually and reusing them where possible, development graphs can avoid redundancy and move the-

orems between theories. From the MMT perspective, a decomposed global theorem link is simply a set of total views without deep assignments, i.e., views where all structures are mapped to morphisms. Thus, MMT provides not only a representation format for development graphs and decomposed theorem links, but also for intermediate development graphs in which theorem links have been partially decomposed or where local theorem links are postulated but have not been found yet.

Our rules for the module-level reasoning about morphisms are very similar to such decompositions: A judgment about all (possibly imported) constants in  $S$  is decomposed into separate judgments about the local constants and the structures declared of  $S$ .

The common algebraic specification language (**CASL**) was initiated in 1994 in an attempt to unify and standardize existing specification languages. As such, it was strongly influenced by other languages such as OBJ and ASL. The CASL logics are centered around partial subsorted first-order logic, and specific logics are obtained by specializing (e.g., total functions, no subsorting) or extending (e.g., modal logic or higher-order logic). CASL uses closed physical packages based on files and called “libraries”. The modules are called “specifications”, the imports are unnamed and interspersed, permit renaming, and use the identify-semantics. The overload/identify-semantics is used to handle import name clashes. Instantiations are interfaced, explicit, and total, and map constants to constants. In parametric specifications, special separated imports are used that can be instantiated with views. CASL offers simple hiding.

In HetCASL [Mos05] and the Hets system [MML07], CASL is extended to heterogeneous specifications using different logics and logic morphism the same specification). Imports and views may go across logics if logic morphisms are attached. This is a very similar to the use of meta-theories and metamorphisms in MMT. Contrary to MMT, the logics and logic morphisms are implemented in the underlying programming language and not declared within the formal language itself. Hets implements the development graph calculus for heterogeneous specifications.

*Type Theories.* Type theories and related formal languages utilize strong logical systems to express both mathematical statements and proofs as mathematical objects. Some systems like AutoMath [dB70], Isabelle [Pau94], or Twelf [PS99] even allow the specification of the logical language itself, in which the reasoning takes place. Semi-automated theorem proving systems have been used to formalize substantial parts of mathematics and mechanically verify many theorems in the respective areas.

These systems usually come with a module system that manages and structures the body of knowledge formalized in the system and a library containing a large set of modules. We will consider the module systems of IMPS [FGT93], PVS [ORS92, OS97], Isabelle [Pau94], Coq [CH88, BC04], Agda [Nor05], and Nuprl [CAB<sup>+</sup>86]. We have already discussed the module system of Twelf, which was designed based on MMT, in Sect. 9.3.

**IMPS** was the first theorem proving system that systematically exploited the “little theories approach” of separating theories into small modules and moving theorems along theory morphisms. It was initiated in 1990 and is built around a custom variant of higher-order logic. It is a single-package system, the imports are unnamed and separated without instantiations; there is no renaming. Modules can be related via views, which map symbols to symbols.

**PVS** is an interactive theorem prover for a variant of classical higher-order logic with a rich undecidable type system. The PVS packages are called “libraries” and are physical packages based on directories. Unnamed, interspersed imports have interfaced, total, and implicit instantiations, which map symbols to terms. Unnamed imports of the same module are identified if the instantiations agree. There is no renaming, and the import name clash situation is handled using the overload/identify semantics. Simple hiding is supported by export declarations that determine which names become available upon import.

**Isabelle** is an interactive theorem prover based on simple type theory [Chu40] with a structured high-level proof language. Its packages are called “theories” and are identified physically based on files, packaging is closed. Isabelle provides two generic module systems.

Originally, only axiomatic type classes were used as modules. They permit only inheritance via unnamed, separated imports without instantiations. Type class ascriptions to type variables and overloading resolution are used to access the symbols of a type class. Later locales were introduced as modules in [KWP99] and gradually extended. In the current release, locales offer unnamed, separated imports with free instantiations; renaming is possible. Type classes are recovered as a special case.

Realizations are treated differently depending on whether they are grounded or not and whether the domain is a type class or a locale: Theory morphism between locales are called “sublocale” and “subclass declarations”, and grounded realizations are called “interpretation” for locales and “instantiation” for type classes.

Isabelle assigns the semantics of a modular theory by elaboration. Locales are internalized by locale predicates that abstract over all symbols and assumptions of the locale; every theorem proved in the locale is relativized by the locale predicate and exported to the toplevel. Thus, instantiation is reduced to  $\beta$ -reduction.

**Nuprl** is an interactive theorem prover based on a rich undecidable type theory. It does not provide an explicit module system. However, its type theory is so expressive that it can in principle be used to define an internalized module system as shown in [CH00]. Then modules, grounded realizations, and higher-order functors can be defined using Nuprl types, terms, and function terms, respectively. Named and unnamed imports are defined using intersection and dependent sum types. But Nuprl does not provide specific module system-like syntax for these notions.

**Coq** is an interactive theorem prover based on the calculus of constructions [CH88]. Physical open packages are called “libraries” and correspond to directories and files.

The Coq module system is modeled after the SML module system (see below). SML signatures, structures, and functors correspond to Coq module types, modules without parameters, and modules with parameters, respectively. Contrary to SML, no shadowing is used, and errors are signaled instead. In addition, Coq can be used with an internalized higher-order module system using record types. As for Nuprl, this yields modules, grounded realizations, and higher-order functors. Both module systems are used independently. The standard library mainly uses the former. The latter is used systematically in [GM08].

**Agda** is a functional programming language based on Martin-Löf’s dependent type theory [ML74]. It uses dependent record types to internalize certain theories. In addition, the notion of “modules” combines aspects of what we call packages and modules. These modules are physical closed packages based on files and are used mainly for namespace management. Named interspersed imports between modules are possible using nested module declarations where the inner one is defined in terms of a parametric module. These imports carry interfaced, implicit, and total instantiations that map symbols to term. Named imports may not occur as parameters so that this does not yield a notion of functors.

*Programming Languages.* Programming languages differ from the languages mentioned above in that they focus on aspects of execution including input/output and state. But if we ignore those aspects, we find the same module system patterns as in the other languages. We discuss the functional language SML [MTHM97] and the object-oriented language Java [GJJ96] as examples.

**SML** uses a single-package system that permits the modular design of specifications (called “signatures”) and realizations (called “functors”, and if grounded “structures”).

The specification level module system has signatures as modules. Imports are interspersed and can be named (called “structure declarations”) or unnamed (called “inclusions”). Both kinds of imports carry free, explicit, and partial instantiations that map symbols to symbols or structures to realizations. If unnamed imports lead to a diamond situation or a name clash, the later declarations always shadow the previous ones. Views are restricted to inclusion morphisms between signatures (called “structural subtyping”); these views are implicit and inferred by implementations.

Realizations can themselves be given modularly. A functor is a realization of a signature that is parametric in symbols or structure declarations. Imports between realizations are possible by declaring a structure and defining it to be equal to the result of a functor application. Consequently, these imports are named and interspersed, and the instantiations are interfaced, explicit, and total, map symbols to symbols and structures to realizations. Structures are typed structurally by signatures, which permits simple hiding.

From an MMT perspective, SML signatures, structures, and functors can be unified conceptually. Signatures correspond to MMT theories in which no constant has a definition; structures to MMT theories in which all constants have definitions; and functors to MMT theories where only a few declarations at the beginning (the interface of the functor) have no definition. Both the structural subtyping relation between signatures and the typing relation between structures and signatures correspond to an inclusion view between the respective MMT theories.

**Java** uses open packages with optional imports. Package names are the authority components of URIs [BLFM05]. Packages are provided in `jar` archive files, and implementations provide a catalog to locate packages that is based on the `classpath`. Java packages are very close to MMT documents. Similar to MMT, Java identifiers are logical and formed from the three hierarchical components package URI, class name, and field name. However, Java uses “.” as a separator character both between and within these components and resolves ambiguities dynamically; MMT uses “?” and “/” so that MMT URIs can be understood statically.

Java modules are called “classes”. There are two kinds of imports. Firstly, unnamed, separated imports without renaming are called “class inheritance”; a class may only inherit from one other class though. Secondly, named, interspersed imports are called “object instantiation”, and the resulting structures “objects”. Instantiations are interfaced, implicit, and total, but a class may provide multiple interfaces (called “constructors”), which map symbols to expressions or objects to objects. As constructors may execute code, the expressions passed to the constructor do not have to correspond to symbols or objects declared in the class. Views are restricted to inclusion morphism out of special modules (called “interfaces”). Simple hiding is realized via private declarations.

Java internalizes its module system, and functors are subsumed by the concept of methods.

Scala [OSV07] is a higher-order extension of Java that retains all features listed above for Java. Moreover, Scala permits multiple unnamed imports into the same class by using “traits”.

## 11. Conclusion and Future Work

Formal knowledge is at the core of mathematics, logic, and computer science, and we are seeing a trend towards employing computational systems like (semi-)automated theorem provers, model checkers, computer algebra systems, constraint solvers, or concept classifiers to deal with it. It is a characteristic feature of these systems that they either have mathematical knowledge implicitly encoded in their critical algorithms or (increasingly) manipulate explicit representations of this knowledge, often in the form of logical formulas. Unfortunately, these systems have differing domains of applications, foundational assumptions, and input languages, which makes them non-interoperable and

difficult to compare and relate in practice. Moreover, the quantity of mathematical knowledge is growing faster than our ability to formalize and organize it, aggravating the problem that mathematical software systems cannot easily share knowledge representations.

In this work, we contributed to the solution of this problem by providing a scalable representation language for mathematical knowledge. We have focused on the modular organization of formal, explicitly represented mathematical knowledge. We have developed a classification of modular knowledge representation languages and evaluated the space of possible module systems in terms of expressivity, computational tractability, and scalability. We have distilled our findings into one particularly well-behaved system – MMT – discussed its properties, and described a set of loosely coupled implementations.

### *11.1. The MMT Language*

MMT is a foundationally unconstrained module system that serves as a web-scalable interface layer between computational systems working with formally represented knowledge.

MMT integrates successful features of existing paradigms

- reuse along theory morphisms from the “little theories” approach,
- the theory graph abstraction from algebraic specification languages,
- categories of theories and logics from model theoretical logical frameworks,
- the logics-as-theories representation from proof theoretical logical frameworks,
- declarations of constants and named realizations from type theory,
- the Curry-Howard correspondence from type/proof theory,
- URIs as logical namespace identifiers from OPENMATH/OMDOC and Java,
- standardized XML-based concrete syntax from web-oriented representation languages,

and makes them available in a single, coherent representational system for the first time.

The combination of these features is reduced to a small set of carefully chosen, orthogonal primitives in order to obtain a simple and extensible language design. In fact, some of the primitives combine so many intuitions that it was rather difficult to name them.

MMT contributes three new features:

**Canonical identifiers** By making morphisms named objects, MMT can provide globally unique, web-scalable identifiers for all knowledge items. Even in the presence of modularity and reuse, all induced knowledge items become addressable via URIs. Moreover, identifiers are invariant under MMT operations such as flattening.

**Meta-theories** The logical foundations of domain representations of mathematical knowledge can be represented as modules themselves and can be structured and interlinked via meta-morphisms. Thus, the different foundations of systems can be related and the systems made interoperable. The explicit representation of epistemic foundations also benefits systems whose mathematical knowledge is only implicitly embedded into the algorithms: The explicit representation can serve as a documentation of the system interface as well as a basis for verification or testing attempts.

**Foundation-independence** The design, implementation, and maintenance of large scale logical knowledge management services will realistically only pay off if the same framework can be reused for different foundations of mathematics. Therefore, MMT does not commit to a particular foundation and provides an interface layer between the logical-mathematical core of a mathematical foundation and knowledge management services. Thus, the latter can respect the semantics of the former without knowing or implementing the foundation.

MMT is **web-scalable** in the sense that it supports the distribution of resources (theories, proofs, etc.) over the internet thus permitting their collaborative development and application. We can encapsulate MMT-based or MMT-aware systems as web-services and use MMT as a universal interface language. At the same time MMT is **fully formal** in the sense that its semantics is specified rigorously in a self-contained formal system, namely using the type-theoretical style of judgments and inference rules. Such a level of formality is rare among module systems, SML being one of the few examples.

We contend that the dream of formalizing large parts of mathematics to make them machine-understandable can only be reached based on a system with both these features. However, in practice, they are often in conflict, and their combination makes MMT unique. In particular, it is easy to write large scale implementations in MMT, and it is easy to verify and trust them.

### 11.2. Beyond MMT

We have designed MMT as the simplest possible language that combines foundation-independence, modularity, web-scalability, and formality. Future work can now build on MMT and add individual orthogonal language features – in each case preserving these four qualities. In particular, for each feature, we have to define grammar and inference rules, the induced knowledge items and their URIs, and their behavior under theory morphisms. In fact, we have already developed some of these features but excluded them in this paper to focus on a minimal core language.

In the following we list some language features that we will carefully add to MMT in the future:

**Unnamed Imports** In addition to the described named imports with distinguish-semantics, MMT is designed to provide also unnamed imports with identify-semantics. They are already part of the MMT API, and the main reason to

omit them here was to simplify the presentation of the formal semantics of MMT.

**Cyclic Imports** Inspecting the flattening theorem reveals that cyclic imports are not as harmful as one might think: Cyclic imports can be elaborated easily if we permit theories with infinitely many constant declarations. In particular, cyclic imports will permit elegant representations of languages with an infinite hierarchy of universes or with an infinite hierarchy of reflection.

**Nested Theories** Nested theories will provide a scalable mechanism for representing hierachic scopes and visibility. Many language features naturally suggest such a nesting of scopes such as mutual recursion, local functions, record types, or proofs with local definitions.

Intuitively, if  $S$  is a subtheory of  $T$ , the declarations of  $T$  occurring before  $S$  are implicitly imported into  $S$  via an unnamed import, and the declarations of  $T$  succeeding  $S$  can refer to  $S$ , e.g., by importing it. The main difficulty here is to add nested theories in a way that preserves the order-invariance of declarations.

**(Co-)Inductive Data Types** Using some of the above features, it is possible to give foundation-independent definitions of inductive and coinductive data types. An inductive data type over  $T$  is declared as a theory  $I$  with a distinguished type  $t$  over  $I$ : The values of the induced type are defined using the closed terms  $\omega$  such that  $\gamma \triangleright_I \omega : t$ . Functions from this type to some type  $u$  over  $T$  can be defined by induction, which amounts to giving a theory morphism from  $I$  to  $T$  that maps  $t$  to  $u$ .

A coinductive data type over  $T$  is declared as a theory  $C$  with a distinguished partial morphism  $m$  from  $C$  to  $T$ . The values of the induced type are defined using the valid morphisms  $\gamma \triangleright \mu : C \rightarrow T$  that agree with  $m$ . Thus, definitions by coinduction are reduced to theory morphisms. In particular,  $C$  specializes to a record type if it does not contain cyclic imports.

Coinductive types can be used to reflect the MMT-concept of realizations into individual foundations. For example, consider Ex. 15 with  $C = \text{Monoid}$ ,  $T = \text{ZFC}$ , and  $m = \text{FOLSem}$ . Then the values of the coinductive type over  $T$  given by  $C$  and  $m$  are the models of  $C$ .

**Theory Expressions** Some module systems, e.g., CASL or Isabelle, provide complex theory expressions. For example,  $S \cup T$  can denote the union of the theories  $S$  and  $T$ . Other examples are the translation of a theory along a morphism, the extension of a theory with some declarations, or the pushout of certain morphisms. Similarly, we can add further productions for morphism expression, e.g., for the mediating morphism out of a pushout.

The main difficulty here is that these complex theories and consequently their declarations do not have canonical identifiers. Indeed, most systems handle theory expressions by decomposing them internally and generating fresh internal names for the involved subexpressions. Similarly, all of these constructions can be expressed in MMT already by introducing auxiliary theories as

we showed in [CHK<sup>+</sup>11a]. But certain theory expressions – most importantly unions and pushouts along unnamed imports – can be added to MMT in a way that preserves canonical identifiers without using generated names.

**Conservative Extensions** A common practice is to give a theory  $S$  with undefined constants – the primitive concepts – and then another theory  $T$  that imports  $S$  and adds with defined constants – the derived concepts. This is particularly important when the declarations of  $S$  represent axioms and those of  $T$  theorems. In that case, it is desirable to make this kind of conservativity of  $T$  explicit in order to exploit it later. For example, if  $T$  is conservative over  $S$ , then a theory importing  $S$  should implicitly also gain access to  $T$ .

**Hiding and Filtering** In [KRC11], we showed how a slight extension of the semantics of filtering yields a substantial increase in expressivity. In particular, it becomes possible to safely relax the strictness of filtering. The key idea is that foundations do not only say “yes” when confirming a typing or equality relation but also return a list of dependencies, which MMT maintains and uses to propagate filtering. We use a syntactically similar but semantically different extension of MMT in [CHK<sup>+</sup>11a] to extend MMT with model theoretical hiding. We expect that further research will permit the unification of these two features.

**Sorting** The components of a constant declaration – type and definiens – correspond to the base judgments provided by the foundations – typing and equality. In particular, MMT uses the constant declarations to provide the axioms of the inference systems used in specific foundations. It is natural but not necessary to consider exactly typing and equality. For example, we can extend MMT with constant declarations  $c <: \tau$  that declare  $c$  as a *sort* refining  $\tau$ . Examples are subtypes (refining types), type classes (refining the kind of types), and set theoretical classes (refining the universe of sets). This extension would go together with a subsorting judgment  $\gamma \triangleright_T \omega <: \omega'$  in the foundation.

**Logical Relations** The notions of theory and theory morphisms between theories can be extended with logical relations between theory morphisms. MMT logical relations will be purely syntactical notions that correspond to the well-known semantic ones. A preliminary account was given in [Soj10]. They will permit natural representations of relations between realizations – such as model morphisms – as well as of extensional equality relations.

**Computation** MMT is currently restricted to declarative languages thus excluding the important role of computation, e.g., in computer algebra systems, decision procedures, and programs extracted from proofs. Generating code from appropriate MMT theories is relatively simple. But we also want to permit literal code snippets in the definiens of a constant. This will provide a formal interface between a formal semantics and scalable implementations.

**Aliases** MMT avoids the introduction of new names for symbols; instead, canonical qualified identifiers are formed. But this often leads to long unfriendly identifiers. Aliases for individual identifiers or identifier prefixes are

a simple syntactic device for providing human-friendly names, e.g., by declaring the aliases `+` and `*` for `add/mon/comp` and `mult/comp` in the theory `Ring`. Moreover, such names can be used to make the modular structure of a theory transparent. This is already part of our implementation.

**Declaration Patterns and Functors** A common feature of declarative languages is that the declarations in a theory  $T$  with meta-theory  $M$  must follow one out of several patterns. For example, if  $M$  is first-order logic, then  $T$  should contain only function symbol, predicate symbol, and axiom declarations. We can capture this foundation-independently in MMT by declaring such patterns in  $M$  and then pattern-checking the declarations in  $T$  against them.

Patterns also permit adding a notion of functors to MMT whose input is an arbitrary well-patterned theory  $T$  with meta-theory  $M$ . The output is a theory defined by induction on the list of declarations in  $T$ . This permits concise representations of functors between categories of theories, e.g., the functor that takes a sorted first-order theory and returns its translation to unsorted first-order logic by relativization of quantifiers. This can be extended to functors between categories of diagrams.

**Minimal Foundations** Not all language features can be defined foundation-independently. Consider Mizar-style [TB85] implicit definitions of the form

```
func c means F(c); correctness P;
```

where  $P$  is a proof of  $\exists!x.F(x)$  and  $c$  is defined as that unique value. Such a definition is meaningful iff the foundational theory can express the quantifier  $\exists!$  of unique existence. Moreover, in that case it can be elaborated into the two declarations  $c$  and  $c.def : F(c)$  (which is in fact what Mizar and most other systems are doing).

In the spirit of little foundations, we will add such pragmatic language features to MMT together with the minimal foundations needed to define their semantics. If an individual foundational theory  $M$  imports one of these distinguished minimal foundations, the corresponding pragmatic feature becomes available in theories with meta-theory  $M$ .

Further pragmatic declarations include, for example, function declarations (possible if  $M$  can express  $\lambda$ -abstraction) and constants with multiple types (possible if  $M$  can express intersection types). The above-mentioned features of sorting and (co-)inductive data types as well as the Curry-Howard representation of axioms, theorems, and proof rules can become special cases of pragmatic features as well. We can even generalize the notion of foundations and then recover the type and definiens of a constant as pragmatic features that are possible if  $M$  can express typing and equality.

**Narrative and Informal Representations** One motivation of MMT has been to give a formal semantics to OMDoc 1.2, and the present work does this for the OMDoc fragment concerned with formal theory development. It omits

narrative aspects (e.g., document structuring, notations, examples, citations) as well as informal and semi-formal representations. We will extend MMT towards all of OMDOC, and this effort will culminate in the OMDOC 2 language. As a first step, we have included sectioning and notations in the MMT API. Many other features of OMDOC 1.2 will be recovered as pragmatic features in the above sense.

### 11.3. Applying MMT

The development of MMT and its implementations has been driven by our ongoing and intended applications. Most importantly, we have evaluated MMT on the logic atlas built in the LATIN project as described in Sect. 9. Here, MMT is applied in two ways.

Firstly, MMT provides the ontology used to organize the highly interlinked theories in the logic graph. In particular, the MMT principles of meta-theories and foundation-independence provide a clean separation of concerns between the logical framework (LF in the case of LATIN), the logics, and the domain theories written in these logics.

Secondly, MMT serves as the scalable interface language between the various MMT-aware software systems used in LATIN. Twelf [PS99] is used to write logics, TNTBASE [ZK09] for persistent storage, the MMT API for presentation and indexing, JOBAD [GLR09] for interactive browsing, and Hets [MML07] for institution-based cross-logic proof management, and we are currently adding sTeXIDE [JK10] for semantic authoring support. MMT is crucial to communicate the content and its semantics between both the heterogeneous platforms and the respective developers. In particular, the canonical MMT identifiers have proved pivotal for the integration of software systems.

Building on the LATIN atlas, we are creating an “Open Archive of FlexiForms” (OAFF). It will store flexiformal (i.e., represented at flexible degrees of formality) representations of mathematical knowledge and supply them with MMT-base knowledge management services. OAFF will contain the domain theories and libraries written in the logics that are part of the LATIN atlas. Using MMT, it becomes possible to represent libraries developed in different foundational systems in one uniform formalism. Since MMT can also represent relations between the underlying foundational system, this provides a base for practical reliable system integration. For example, we are currently importing the libraries of TPTP [SS98] and Mizar [TB85] into OAFF. Other systems like Coq [BC04], Isabelle [Pau94], or PVS [ORS92] already have XML or OMDOC 1.2 exports that can be updated to export MMT. Variables:

- [ABC<sup>+</sup>10] R. Ausbrooks, S. Buswell, D. Carlisle, G. Chavchanidze, S. Dalmas, S. Devitt, A. Diaz, S. Dooley, R. Hunter, P. Ion, M. Kohlhase, A. Lazrek, P. Libbrecht, B. Miller, R. Miner, C. Rowley, M. Sargent, B. Smith, N. Soiffer, R. Sutor, and S. Watt. Mathematical Markup Language (MathML) Version 3.0. Technical report, World Wide Web Consortium, 2010. See <http://www.w3.org/TR/MathML3>.

- [ACTZ06] A. Asperti, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. Crafting a Proof Assistant. In T. Altenkirch and C. McBride, editors, *TYPES*, pages 18–32. Springer, 2006.
- [AHMS99] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an Evolutionary Formal Software-Development Using CASL. In D. Bert, C. Choppy, and P. Mosses, editors, *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1999.
- [AHMS02] S. Autexier, D. Hutter, T. Mossakowski, and A. Schairer. The Development Graph Manager Maya (System Description). In H. Kirchner and C. Ringeissen, editors, *Algebraic Methods and Software Technology, 9th International Conference*, pages 495–502. Springer, 2002.
- [Apa00] Apache Software Foundation. Apache Subversion, 2000. see <http://subversion.apache.org/>.
- [BC04] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.
- [BCC<sup>+</sup>04] S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See <http://www.openmath.org/standard/om20>.
- [Ber37] P. Bernays. A System of Axiomatic Set Theory – Part I. *Journal of Symbolic Logic*, 2(1):65–77, 1937.
- [BLFM05] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, Internet Engineering Task Force, 2005.
- [Bou68] N. Bourbaki. *Theory of Sets*. Elements of Mathematics. Springer, 1968.
- [Bou74] N. Bourbaki. *Algebra I*. Elements of Mathematics. Springer, 1974.
- [CAB<sup>+</sup>86] R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.
- [CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4, pages 65–89, 1996.
- [CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988.

- [CH00] R. Constable and J. Hickey. Nuprl’s Class Theory and Its Applications. In F. Bauer and R. Steinbruggen, editors, *Foundations of Secure Computation*, pages 91–115. IOS Press, 2000.
- [CHK<sup>+</sup>11a] M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. A Proof Theoretic Interpretation of Model Theoretic Hiding. In *Workshop on Abstract Development Techniques*, Lecture Notes in Computer Science. Springer, 2011. To appear.
- [CHK<sup>+</sup>11b] M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. Project Abstract: Logic Atlas and Integrator (LATIN). Submitted to CICM Systems & Projects, 2011.
- [CHK<sup>+</sup>11c] M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, F. Rabe, and K. Sojakova. Towards Logical Frameworks in the Heterogeneous Tool Set Hets. In *Workshop on Abstract Development Techniques*, Lecture Notes in Computer Science. Springer, 2011. To appear.
- [Chu40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.
- [CO01] Olga Caprotti and Martijn Oostdijk. On communicating proofs in interactive mathematical documents. In Eugenio Roanes Lozano, editor, *Proceedings of Artificial Intelligence and Symbolic Computation, AISC’2000*, number 1930 in LNAI, pages 53–64. Springer Verlag, 2001.
- [CoF04] CoFI (The Common Framework Initiative). *CASL Reference Manual*, volume 2960 of *LNCS*. Springer, 2004.
- [dB70] N. de Bruijn. The Mathematical Language AUTOMATH. In M. Laudet, editor, *Proceedings of the Symposium on Automated Demonstration*, volume 25 of *Lecture Notes in Mathematics*, pages 29–61. Springer, 1970.
- [DHS09] S. Dumbrava, F. Horozal, and K. Sojakova. A Case Study on Formalizing Algebra in a Module System. In F. Rabe and C. Schürmann, editors, *Workshop on Modules and Libraries for Proof Assistants*, volume 429 of *ACM International Conference Proceeding Series*, pages 11–18, 2009.
- [Far00] W. Farmer. An Infrastructure for Intertheory Reasoning. In D. McAllester, editor, *Conference on Automated Deduction*, pages 115–131. Springer, 2000.
- [FGT92] W. Farmer, J. Guttman, and F. Thayer. Little Theories. In D. Kapur, editor, *Conference on Automated Deduction*, pages 467–581, 1992.

- [FGT93] W. Farmer, J. Guttman, and F. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11(2):213–248, 1993.
- [Fra22] A. Fraenkel. Zu den Grundlagen der Cantor-Zermeloschen Mengenlehre. *Mathematische Annalen*, 86:230–237, 1922. English title: On the Foundation of Cantor-Zermelo Set Theory.
- [GB92] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.
- [GJJ96] J. Gosling, W. Joy, and G. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996.
- [GLR09] J. Gićeva, C. Lange, and F. Rabe. Integrating Web Services into Active Mathematical Documents. In J. Carette and L. Dixon and C. Sacerdoti Coen and S. Watt, editor, *Intelligent Computer Mathematics*, volume 5625 of *Lecture Notes in Computer Science*, pages 279–293. Springer, 2009.
- [GM08] G. Gonthier and A. Mahboubi. A Small Scale Reflection Extension for the Coq system. Technical Report RR-6455, INRIA, 2008.
- [Göd40] K. Gödel. The Consistency of Continuum Hypothesis. *Annals of Mathematics Studies*, 3:33–101, 1940.
- [GTW78] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice Hall, 1978.
- [GWM<sup>+</sup>93] J. Goguen, Timothy Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. In J. Goguen, D. Coleman, and R. Gallimore, editors, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.
- [HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
- [HR09] Peter Horn and Dan Roodmand. OpenMath in SCIEnce: SC-SCP and POPCORN. In Jacques Carette, Lucas Dixon, Claudio Sacerdoti Coen, and Stephen M. Watt, editors, *MKM/Calculemus Proceedings*, number 5625 in LNAI, pages 474–479. Springer Verlag, July 2009.
- [HR11] F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. *Theoretical Computer Science*, 2011. To appear, see [http://kwarc.info/frabe/Research/HR\\_folsound\\_10.pdf](http://kwarc.info/frabe/Research/HR_folsound_10.pdf).

- [HST94] R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.
- [IR11] M. Iancu and F. Rabe. Formalizing Foundations of Mathematics. *Mathematical Structures in Computer Science*, 2011. To appear, see [http://kwarc.info/frabe/Research/IR\\_foundations\\_10.pdf](http://kwarc.info/frabe/Research/IR_foundations_10.pdf).
- [JK10] C. Jucovschi and M. Kohlhase. sTeXIDE: An Integrated Development Environment for sTeX Collections. In S. Autexier, J. Calmet, D. Delahaye, P. Ion, L. Rideau, R. Rioboo, and A. Sexton, editors, *Intelligent Computer Mathematics*, number 6167 in Lecture Notes in Artificial Intelligence. Springer, 2010.
- [KMR08] M. Kohlhase, C. Müller, and F. Rabe. Notations for Living Mathematical Documents. In S. Autexier and J. Campbell and J. Rubio and V. Sorge and M. Suzuki and F. Wiedijk, editor, *Mathematical Knowledge Management*, volume 5144 of *Lecture Notes in Computer Science*, pages 504–519, 2008.
- [KMR09] M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN Project, 2009. See <https://trac.omdoc.org/LATIN/>.
- [Koh06] M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*. Number 4180 in Lecture Notes in Artificial Intelligence. Springer, 2006.
- [KRC11] M. Kohlhase, F. Rabe, and C. Sacerdoti Coen. A Foundational View on Integration Problems. Submitted to CICM, see [http://kwarc.info/frabe/Research/KRS\\_integration\\_10.pdf](http://kwarc.info/frabe/Research/KRS_integration_10.pdf), 2011.
- [KRZ10] M. Kohlhase, F. Rabe, and V. Zholudev. Towards MKM in the Large: Modular Representation and Scalable Software Architecture. In S. Autexier, J. Calmet, D. Delahaye, P. Ion, L. Rideau, R. Rioboo, and A. Sexton, editors, *Intelligent Computer Mathematics*, volume 6167 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2010.
- [KWP99] F. Kammüller, M. Wenzel, and L. Paulson. Locales – a Sectioning Concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics*, pages 149–166. Springer, 1999.
- [Lan98] S. Mac Lane. *Categories for the working mathematician*. Springer, 1998.
- [LCH07] D. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of Standard ML. In M. Hofmann and M. Felleisen, editors, *Symposium on Principles of Programming Languages*, pages 173–184. ACM, 2007.

- [MAH06] T. Mossakowski, S. Autexier, and D. Hutter. Development graphs - Proof management for structured specifications. *J. Log. Algebr. Program.*, 67(1–2):114–145, 2006.
- [ML74] P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the '73 Logic Colloquium*, pages 73–118. North-Holland, 1974.
- [MML07] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522, 2007.
- [Mos05] T. Mossakowski. Heterogeneous Specification and the Heterogeneous Tool Set, 2005. Habilitation thesis, see <http://www.informatik.uni-bremen.de/~till/>.
- [MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*, Revised edition. MIT Press, 1997.
- [Nor05] U. Norell. The Agda WiKi, 2005. <http://wiki.portal.chalmers.se/agda>.
- [Odl95] A. Odlyzko. Tragic loss or good riddance? The impending demise of traditional scholarly journals. *International Journal of Human-Computer Studies*, 42:71–122, 1995.
- [Ora10] Oracle. Oracle berkeley db xml, 2010. see <http://www.oracle.com/us/products/database/berkeley-db/xml/index.html>.
- [ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752. Springer, 1992.
- [OS97] S. Owre and N. Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, SRI International, 1997.
- [OSV07] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. artima, 2007.
- [Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
- [Pea07] D. Pollak and et. al. Lift web framework, 2007. <http://liftweb.net>.
- [PS99] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.

- [Rab08a] F. Rabe. *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen, 2008. Available at <http://kwarc.info/frabe/Research/phdthesis.pdf>.
- [Rab08b] F. Rabe. The MMT System, 2008. See <https://trac.kwarc.info/MMT/>.
- [Rab10] F. Rabe. A Logical Framework Combining Model and Proof Theory. Submitted to Mathematical Structures in Computer Science, see [http://kwarc.info/frabe/Research/rabe\\_combining\\_09.pdf](http://kwarc.info/frabe/Research/rabe_combining_09.pdf), 2010.
- [RS09] F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, pages 40–48. ACM Press, 2009.
- [SML97] Standard ml basis library, 1997. See <http://www.standardml.org/Basis/>.
- [Soj10] K. Sojakova. Mechanically Verifying Logic Translations, 2010. Master’s thesis, Jacobs University Bremen.
- [Sol95] R. Solomon. On Finite Simple Groups and Their Classification. *Notices of the AMS*, pages 231–239, 1995.
- [SS98] G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
- [ST88] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Control*, 76:165–210, 1988.
- [SW83] D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. In M. Karpinski, editor, *Fundamentals of Computation Theory*, pages 413–427. Springer, 1983.
- [TB85] A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985.
- [W3C98] W3C. Extensible Markup Language (XML), 1998. <http://www.w3.org/XML>.
- [W3C99] W3C. XML Path Language, 1999. <http://www.w3.org/TR/xpath/>.
- [W3C07] W3C. XQuery 1.0: An XML Query Language, 2007. <http://www.w3.org/TR/xquery/>.

- [Wir77] N. Wirth. Design and Implementation of Modula. *Software-Practice and Experience*, 7(1):67–84, 1977.
- [ZBM31] Zentralblatt MATH, 1931. <http://www.zentralblatt-math.org>.
- [Zer08] E. Zermelo. Untersuchungen ber die Grundlagen der Mengenlehre I. *Mathematische Annalen*, 65:261–281, 1908. English title: Investigations in the foundations of set theory I.
- [ZK09] V. Zholudev and M. Kohlhase. TNTBase: a Versioned Storage for XML. In *Proceedings of Balisage: The Markup Conference 2009*, volume 3 of *Balisage Series on Markup Technologies*. Mulberry Technologies, Inc., 2009.
- [ZK10] V. Zholudev and M. Kohlhase. Scripting Documents with XQuery: Virtual Documents in TNTBase. In *Proceedings of Balisage: The Markup Conference*, Balisage Series on Markup Technologies. Mulberry Technologies, Inc., 2010.
- [ZKR10] V. Zholudev, M. Kohlhase, and F. Rabe. A [insert XML Format] Database for [insert cool application]. In *Proceedings of XML-Prague*. XMPPraague.cz, 2010.

# Notations for Living Mathematical Documents

Michael Kohlhase and Christine Müller and Florian Rabe

Computer Science, Jacobs University Bremen  
`{m.kohlhase,c.mueller,f.rabe}@jacobs-university.de`

**Abstract.** Notations are central for understanding mathematical discourse. Readers would like to read notations that transport the meaning well and prefer notations that are familiar to them. Therefore, authors optimize the choice of notations with respect to these two criteria, while at the same time trying to remain consistent over the document and their own prior publications. In print media where notations are fixed at publication time, this is an over-constrained problem. In living documents notations can be adapted at reading time, taking reader preferences into account.

We present a representational infrastructure for notations in living mathematical documents. Mathematical notations can be defined declaratively. Author and reader can extensionally define the set of available notation definitions at arbitrary document levels, and they can guide the notation selection function via intensional annotations.

We give an abstract specification of notation definitions and the flexible rendering algorithms and show their coverage on paradigmatic examples. We show how to use this framework to render OPENMATH and Content-MATHML to Presentation-MATHML, but the approach extends to arbitrary content and presentation formats. We discuss prototypical implementations of all aspects of the rendering pipeline.

## 1 Introduction

Over the last three millennia, mathematics has developed a complicated two-dimensional format for communicating formulae (see e.g., [Caj93,Wol00] for details). Structural properties of operators often result in special presentations, e.g., the scope of a radical expression is visualized by the length of its bar. Their mathematical properties give rise to placement (e.g., associative arithmetic operators are written infix), and their relative importance is expressed in terms of binding strength conventions for brackets. Changes in notation have been influential in shaping the way we calculate and think about mathematical concepts, and understanding mathematical notations is an essential part of any mathematics education. All of these make it difficult to determine the functional structure of an expression from its presentation.

Content Markup formats for mathematics such as OPENMATH [BCC<sup>+</sup>04] and content MATHML [ABC<sup>+</sup>03] concentrate on the functional structure of mathematical formulae, thus allowing mathematical software systems to exchange mathematical objects. For communication with humans, these formats rely on a

“presentation process” (usually based on XSLT style sheets) that transforms the content objects into the usual two-dimensional form used in mathematical books and articles. Many such presentation processes have been proposed, and all have their strengths and weaknesses. In this paper, we conceptualize the presentation of mathematical formulae as consisting of two components: the two-dimensional **composition** of visual sub-presentations to larger ones and the **elision** of formula parts that can be deduced from context.

Most current presentation processes concentrate on the relatively well-understood composition aspect and implement only rather simple bracket elision algorithms. But the visual renderings of formulae in mathematical practice are not simple direct compositions of the concepts involved: mathematicians gloss over parts of the formulae, e.g., leaving out arguments, iff they are non-essential, conventionalized or can be deduced from the context. Indeed this is part of what makes mathematics so hard to read for beginners, but also what makes mathematical language so efficient for the initiates. A common example is the use of  $\log(x)$  or even  $\log x$  for  $\log_{10}(x)$  or similarly  $[t]$  for  $[t]_{\mathcal{M}}^{\varphi}$ , if there is only one model  $\mathcal{M}$  in the context and  $\varphi$  is the most salient variable assignment.

Another example are the bracket elision rules in arithmetical expressions:  $ax + y$  is actually  $(ax) + y$ , since multiplication “binds stronger” than addition. Note that we would not consider the “invisible times” operation as another elision, but as an alternative presentation.

In this situation we propose to encode the presentational characteristics of symbols (for composition *and* elision) declaratively in **notation definitions**, which are part of the representational infrastructure and consist of “prototypes” (patterns that are matched against content representation trees) and “renderings” (that are used to construct the corresponding presentational trees). Note that since we have reified the notations, we can now devise flexible management process for notations. For example, we can capture the notation preferences of authors, aggregators and readers and adapt documents to these. We propose an elaborated mechanism to collect notations from various sources and specify notation preferences. This brings the separation of *function* from *form* in mathematical objects and assertions in MKM formats to fruition on the document level. This is especially pronounced in the context of dynamic presentation media (e.g., on the screen), we can now realize “*active documents*”, where we can interact with a document directly, e.g., instantiating a formula with concrete values or graphing a function to explore it or “*living/evolving documents*” which monitor the change of knowledge about a topic and adapt to a user’s notation preferences consistently.

Before we present our system, let us review the state of the art. Naylor, Smirnova, and Watt [[NW01a](#),[SW06b](#),[SW06a](#)] present an approach based on meta stylesheets that utilizes a MATHML-based markup of arbitrary notations in terms of their content and presentation and, based on the manual selection of users, generates user-specific XSLT style sheets [[Kay06](#)] for the adaptation of documents. Naylor and Watt [[NW01a](#)] introduce a one-dimensional context annotation of content expressions to intensionally select an appropriate nota-

tion specification. The authors claim that users also want to delegate the styling decision to some defaulting mechanism and propose the following hierarchy of default notation specification (from high to low): command line control, input documents defaults, meta stylesheets defaults, and content dictionary defaults.

In [MLUM05], Manzoor et al. emphasize the need for maintaining uniform and appropriate notations in collaborative environments, in which various authors contribute mathematical material. They address the problem by providing authors with respective tools for editing notations as well as by developing a framework for a consistent presentation of symbols. In particular, they extend the approach of Naylor and Watt by an explicit language markup of the content expression. Moreover, the authors propose the following prioritization of different notation styles (from high to low): individual style, group, book, author or collection, and system defaults.

In [KLR07] we have revised and improved the presentation specification of OMDoc1.2. [Koh06] by allowing a static well-formedness, i.e., the well-formedness of presentation specifications can be verified when writing the presentations rather than when presenting a document. We also addressed the issue of flexible elision. However, the approach does not facilitate to specify notations, which are not local tree transformations of the semantic markup.

In [KMM07] we initiated the redefinition of *documents* towards a more *dynamic* and *living* view. We explicated the narrative and content layer and extended the *document model* by a third dimension, i.e., the *presentation layer*. We proposed the *extensional* markup of the *notation context* of a document, which facilitates users to explicitly select suitable notations for document fragments. These extensional collection of notations can be inherited, extended, reused, and shared among users. For the system presented in this paper, we have re-engineered and extended the latter two proposals.

In Sect. 2, we introduce abstract syntax for notation definitions, which is used for the internal representation of our notation objects. (We use a straightforward XML encoding as concrete syntax.) In Sect. 3, we describe how a given notation definition is used to translate an OPENMATH object into its presentation. After this local view of notation definitions, the remainder of the paper takes a more global perspective by introducing markup that permits users to control which notation definitions are used to present which document fragment. There are two conflicting ways how to define this set of available notation definitions: extensionally by pointing to a notation container; or intensionally by attaching properties to notation definitions and using them to select between them. These ways are handled in Sect. 4 and 5, respectively.

## 2 Syntax of Notation Definitions

We will now present an abstract version of the presentation starting from the observation that in content markup formalisms for mathematics formulae are represented as “formula trees”. Concretely, we will concentrate on OPENMATH objects, the conceptual data model of OPENMATH representations, since it is

sufficiently general, and work is currently under way to re-engineer content MATHML representations based on this model. Furthermore, we observe that the target of the presentation process is also a tree expression: a layout tree made of layout primitives and glyphs, e.g., a presentation MATHML or L<sup>A</sup>T<sub>E</sub>X expression.

To specify notation definitions, we use the one given by the abstract grammar from Fig. 1. Here  $|$ ,  $[-]$ ,  $-^*$ , and  $-^+$  denote alternative, bracketing, and non-empty and possibly empty repetition, respectively. The non-terminal symbol  $\omega$  is used for patterns  $\varphi$  that do not contain jokers. Throughout this paper, we will use the non-terminal symbols of the grammar as meta-variables for objects of the respective syntactic class.

| Notation declarations              | $ntn$           | $::=$ | $\varphi^+ \vdash [(\lambda : \rho)^p]^+$   |
|------------------------------------|-----------------|-------|---|
| Patterns                           | $\varphi$       | $::=$ |   |
| Symbols                            |                 |       | $\sigma(n, n, n)$   |
| Variables                          |                 | $ $   | $v(n)$  |
| Applications                       |                 | $ $   | $@(\varphi, [\varphi])^+$   |
| Binders                            |                 | $ $   | $\beta(\varphi, \Upsilon, \varphi)$   |
| Attributions                       |                 | $ $   | $\alpha(\varphi, \sigma(n, n, n) \mapsto \varphi)$                                |
| Symbol/Variable/Object/List jokers |                 | $ $   | $\underline{s} \mid \underline{v} \mid \underline{o} \mid \underline{l}(\varphi)$ |
| Variable contexts                  | $\Upsilon$      | $::=$ | $\varphi^+$   |
| Match contexts                     | $M$             | $::=$ | $[q \mapsto X]^*$   |
| Matches                            | $X$             | $::=$ | $\omega^*   S^*   (X)$  |
| Empty match contexts               | $\mu$           | $::=$ | $[q \mapsto H]^*$   |
| Holes                              | $H$             | $::=$ | $\_   ``   (H)$   |
| Context annotation                 | $\lambda$       | $::=$ | $(S = S)^*$   |
| Renderings                         | $\rho$          | $::=$ |   |
| XML elements                       |                 |       | $\langle S \rangle \rho^* \langle / \rangle$                                      |
| XML attributes                     |                 | $ $   | $S = `` \rho^* ``$  |
| Texts                              |                 | $ $   | $S$   |
| Symbol or variable names           |                 | $ $   | $\underline{q}$   |
| Matched objects                    |                 | $ $   | $\underline{q}^p$   |
| Matched lists                      |                 | $ $   | $\text{for}(q, I, \rho^*) \{ \rho^* \}$   |
| Precedences                        | $p$             | $::=$ | $-\infty   I   \infty$  |
| Names                              | $n, s, v, l, o$ | $::=$ | $C^+$   |
| Integers                           | $I$             | $::=$ | integer   |
| Qualified joker names              | $q$             | $::=$ | $l/q   s   v   o   l$   |
| Strings                            | $S$             | $::=$ | $C^*$   |
| Characters                         | $C$             | $::=$ | character except /  |

**Fig. 1.** The Grammar for Notation Definitions

*Intuitions* The intuitive meaning of a notation definition  $ntn = \varphi_1, \dots, \varphi_r \vdash (\lambda_1 : \rho_1)^{p_1}, \dots, (\lambda_s : \rho_s)^{p_s}$  is the following: If an object matches one of the patterns  $\varphi_i$ , it is rendered by one of the renderings  $\rho_i$ . Which rendering is chosen, depends on the active rendering context, which is matched against the context annotations  $\lambda_i$  (see Sect. 5). Each context annotation is a key-value list des-

ignating the intended rendering context. The integer values  $p_i$  give the output precedences of the renderings.

The patterns  $\varphi_i$  are formed from a formal grammar for a subset of OPENMATH objects extended with named jokers. The jokers  $\underline{o}$  and  $\underline{l}(\varphi)$  correspond to  $\backslash(\cdot)$  and  $\backslash(\varphi)^+$  in Posix regular expression syntax ([POS88]) – except that our patterns are matched against the list of children of an OPENMATH object instead of against a list of characters. We need two special jokers  $\underline{s}$  and  $\underline{v}$ , which only match OPENMATH symbols and variables, respectively. The renderings  $\rho_i$  are formed by a formal syntax for simplified XML extended with means to refer to the jokers used in the patterns. When referring to object jokers, input precedences are given that are used, together with the output precedences, to determine the placement of brackets.

Match contexts are used to store the result of matching a pattern against an object. Due to list jokers, jokers may be nested; therefore, we use qualified joker names in the match contexts (which are transparent to the user). Empty match contexts are used to store the structure of a match context induced by a pattern: They contain holes that are filled by matching the pattern against an object.

*Example* We will use a multiple integral as an example that shows all aspects of our approach in action.

$$\int_{a_1}^{b_1} \dots \int_{a_n}^{b_n} \sin x_1 + x_2 \, dx_n \dots dx_1.$$

Let *int*, *iv*, *lam*, *plus*, and *sin* abbreviate symbols for integration, closed real intervals, lambda abstraction, addition, and sine. We intend *int*, *lam*, and *plus* to be flexary symbols, i.e., symbols that take an arbitrary finite number of arguments. Furthermore, we assume symbols *color* and *red* from a content dictionary for style attributions. We want to render into LATEX the OPENMATH object

```
@(int, @((iv, a1, b1), ..., @((iv, an, bn),
    β(lam, v(x1), ..., v(xn), α(@plus, @((sin, v(x1)), v(x2)), color ↪ red)))
```

as  $\backslash\text{int}\_{\{a_1\}}^{\{b_1\}} \dots \backslash\text{int}\_{\{a_n\}}^{\{b_n\}} \backslash\text{color}\{\text{red}\}\{\backslash\text{sin } x_1+x_2\}\text{d}x_n \dots \text{d}x_1$

We can do that with the following notations:

```
@(int, ranges(@((iv, a, b)), β(lam, vars(x), f))
  ⊢ ((format = latex) :
      for(ranges){\int_{\{a^\infty\}}^{\{b^\infty\}} f^\infty for(vars, -1){d x^\infty})^{-\infty}
      α(a, color ↪ col) ⊢ ((format = latex) : {\color{col} a^\infty})^{-\infty}
      @plus, args(arg)) ⊢ ((format = latex) : for(args, +){arg})^{10}
      @sin, arg) ⊢ ((format = latex) : \sin arg)^0
```

The first notation matches the application of the symbol *int* to a list of ranges and a lambda abstraction binding a list of variables. The rendering iterates first

over the ranges rendering them as integral signs with bounds, then recurses into the function body  $\underline{f}$ , then iterates over the variables rendering them in reverse order prefixed with  $d$ . The second notation is used when  $\underline{f}$  recurses into the presentation of the function body  $\alpha(@(\text{plus}, @(\text{sin}, v(x_1)), v(x_2)), \text{color} \mapsto \text{red})$ . It matches an attribution of  $\text{color}$ , which is rendered using the L<sup>A</sup>T<sub>E</sub>X color package. The third notation is used when  $\underline{a}$  recurses into the attributed object  $@(\text{plus}, @(\text{sin}, v(x_1)), v(x_2))$ . It matches any application of  $\text{plus}$ , and the rendering iterates over all arguments placing the separator + in between. Finally,  $\text{sin}$  is rendered in a straightforward way. We omit the notation that renders variables by their name.

The output precedence  $-\infty$  of  $\text{int}$  makes sure that the integral as a whole is never bracketed. And the input precedences  $\infty$  make sure that the arguments of  $\text{int}$  are never bracketed. Both are reasonable because the integral notation provides its own fencing symbols, namely  $\int$  and  $d$ . The output precedences of  $\text{plus}$  and  $\text{sin}$  are 10 and 0, which means that  $\text{sin}$  binds stronger; therefore, the expression  $\text{sin } x$  is not bracketed either. However, an inexperienced user may wish to display these brackets: Therefore, our rendering does not suppress them. Rather, we annotate them with an elision level, which is computed as the difference of the two precedences. Dynamic output formats that can change their appearance, such as XHTML with JavaScript, can use the elision level to determine the visibility of symbols based on user-provided elision thresholds: the higher its elision level, the less important a bracket.

*Well-formed Notations* A notation definition  $\varphi_1, \dots, \varphi_r \vdash (\lambda_1 : \rho_1)^{p_1}, \dots, (\lambda_s : \rho_s)^{p_s}$  is well-formed if all  $\varphi_i$  are well-formed patterns that induce the same empty match contexts, and all  $\rho_i$  are well-formed renderings with respect to that empty match context.

Every pattern  $\varphi$  generates an *empty match context*  $\mu(\varphi)$  as follows:

- For an object joker  $\underline{o}$  occurring in  $\varphi$  but not within a list joker,  $\mu(\varphi)$  contains  $\underline{o} \mapsto \underline{\phantom{o}}$ .
- For a symbol or variable with name  $n$  occurring in  $\varphi$  but not within a list joker,  $\mu(\varphi)$  contains  $n \mapsto \text{``''}$ .
- For a list joker  $\underline{l}(\varphi')$  occurring in  $\varphi$ ,  $\mu(\varphi)$  contains
  - $\underline{l} \mapsto (\underline{\phantom{l}})$ , and
  - $\underline{l}/n \mapsto (H)$  for every  $n \mapsto H$  in  $\mu(\varphi')$ .

In an empty match context, a hole  $\underline{\phantom{o}}$  is a placeholder for an object,  $\text{``''}$  for a string,  $(\underline{\phantom{o}})$  for a list of objects,  $((\underline{\phantom{o}}))$  for a list of lists of objects, and so on. Thus, symbol, variable, or object joker in  $\varphi$  produce a single named hole, and every list joker and every joker within a list joker produces a named list of holes ( $H$ ). For example, the empty match context induced by the pattern in the notation for  $\text{int}$  above is

```
ranges  $\mapsto (\underline{\phantom{ranges}})$ , ranges/a  $\mapsto (\underline{\phantom{ranges/a}})$ , ranges/b  $\mapsto (\underline{\phantom{ranges/b}})$ , f  $\mapsto \underline{\phantom{f}}$ ,
vars  $\mapsto (\underline{\phantom{vars}})$ , vars/x  $\mapsto (\text{``''})$ 
```

A pattern  $\varphi$  is well-formed if it satisfies the following conditions:

- There are no duplicate names in  $\mu(\varphi)$ .
- List jokers may not occur as direct children of binders or attributions.
- At most one list joker may occur as a child of the same application, and it may not be the first child.
- At most one list joker may occur in the same variable context.

These restrictions guarantee that matching an OPENMATH object against a pattern is possible in at most one way. In particular, no backtracking is needed in the matching algorithm.

Assume an empty match context  $\mu$ . We define well-formed renderings with respect to  $\mu$  as follows:

- $\langle S \rangle \rho_1, \dots, \rho_r \langle / \rangle$  is well-formed if all  $\rho_i$  are well-formed.
- $S = " \rho_1, \dots, \rho_r "$  is well-formed if all  $\rho_i$  are well-formed and are of the form  $S'$  or  $\underline{n}$ . Furthermore,  $S = " \rho_1, \dots, \rho_r "$  may only occur as a child of an XML element rendering.
- $S$  is well-formed.
- $\underline{n}$  is well-formed if  $n \mapsto "$  is in  $\mu$ .
- $\underline{o}^p$  is well-formed if  $o \mapsto _-$  is in  $\mu$ .
- $\text{for}(l, I, sep)\{body\}$  is well-formed if  $l \mapsto _-$  or  $l \mapsto ("")$  is in  $\mu$ , all renderings in  $sep$  are well-formed with respect to  $\mu$ , and all renderings in  $body$  are well-formed with respect to  $\mu^l$ . The step size  $I$  and the separator  $sep$  are optional, and default to 1 and the empty string, respectively, if omitted.

Here  $\mu^l$  is the empty match context arising from  $\mu$  if every  $l/q \mapsto (H)$  is replaced with  $q \mapsto H$  and every previously existing hole named  $q$  is removed. Replacing  $l/q \mapsto (H)$  means that jokers occurring within the list joker  $l$  are only accessible within a corresponding rendering  $\text{for}(l, I, \rho^*)\{\rho^*\}$ . And removing the previously existing holes means that in  $@(\underline{o}, l(\underline{o}))$ , the inner object joker shadows the outer one.

### 3 Semantics of Notation Definitions

The rendering algorithm takes as input a notation context  $\Pi$  (a list of notation definitions, computed as described in Sect. 4), a rendering context  $\Lambda$  (a list of context annotations, computed as described in Sect. 5), an OPENMATH object  $\omega$ , and an input precedence  $p$ . If the algorithm is invoked from top level (as opposed to a recursive call),  $p$  should be set to  $\infty$  to suppress top level brackets.

It returns as output either text or an XML element. There are two output types for the rendering algorithm: text and sequences of XML elements. We will use  $O + O'$  to denote the concatenation of two outputs  $O$  and  $O'$ . By that, we mean a concatenation of sequences of XML elements or of strings if  $O$  and  $O'$  have the same type. Otherwise,  $O + O'$  is a sequence of XML elements treating text as an XML text node. This operation is associative if we agree that consecutive text nodes are always merged. The algorithm inserts brackets if necessary. And to give the user full control over the appearance of brackets, we obtain the brackets by the rendering of two symbols for left and right bracket from a special fixed content dictionary. The algorithm consists of the following three steps.

1.  $\omega$  is matched against the patterns in the notation definitions in  $\Pi$  (in the listed order) until a matching pattern  $\varphi$  is found. The notation definition in which  $\varphi$  occurs induces a list  $(\lambda_1 : \rho_1)^{p_1}, \dots, (\lambda_n : \rho_n)^{p_n}$  of context-annotations, renderings, and output precedences.
2. The rendering context  $\Lambda$  is matched against the context annotations  $\lambda_i$  in order. The pair  $(\rho_j, p_j)$  with the best matching context-annotation  $\lambda_j$  is selected (see Section 5.2 for details).
3. The output is  $\rho_j^{M(\varphi, \omega)}$ , the rendering of  $\rho_j$  in context  $M(\varphi, \omega)$  as defined below. Additionally, if  $p_j > p$ , the output is enclosed in brackets.

*Semantics of Patterns* The semantics of patterns is that they are matched against OPENMATH objects. Naturally, every OPENMATH object matches against itself. Symbol, variable, and object jokers match in the obvious way. A list joker  $\underline{l}(\varphi)$  matches against a non-empty list of objects all matching  $\varphi$ .

Let  $\varphi$  be a pattern and  $\omega$  a matching OPENMATH object. We define a match context  $M(\varphi, \omega)$  as follows.

- For a symbol or variable joker with name  $n$  that matched against the sub-object  $\omega'$  of  $\omega$ ,  $M(\varphi, \omega)$  contains  $n \mapsto S$  where  $S$  is the name of  $\omega'$ .
- For an object joker  $\underline{o}$  that matched against the sub-object  $\omega'$  of  $\omega$ ,  $M(\varphi, \omega)$  contains  $\underline{o} \mapsto \omega$ .
- If a list joker  $\underline{l}(\varphi')$  matched a list  $\omega_1, \dots, \omega_r$ , then  $M(\varphi, \omega)$  contains
  - $l \mapsto (\omega_1, \dots, \omega_r)$ , and
  - for every  $l/q$  in  $\mu(\varphi)$ :  $l/q \mapsto (X_1, \dots, X_r)$  where  $q \mapsto X_i$  in  $M(\varphi', \omega_i)$ .

We omit the precise definition of what it means for a pattern to match against an object. It is, in principle, well-known from regular expressions. Since no backtracking is needed, the computation of  $M(\varphi, \omega)$  is straightforward. We denote by  $M(q)$ , the lookup of the match bound to  $q$  in a match context  $M$ .

*Semantics of Renderings* If  $\varphi$  matches against  $\omega$  and the rendering  $\rho$  is well formed with respect to  $\mu(\varphi)$ , the intuition of  $\rho^{M(\varphi, \omega)}$  is that the joker references in  $\rho$  are replaced according to  $M(\varphi, \omega) =: M$ . Formally,  $\rho^M$  is defined as follows.

- $\langle S \rangle \rho_1 \dots \rho_r \langle / \rangle$  is rendered as an XML element with name  $S$ . The attributes are those  $\rho_i^M$  that are rendered as attributes. The children are the concatenation of the remaining  $\rho_i^M$  preserving their order.
- $S = " \rho_1 \dots \rho_r "$  is rendered as an attribute with label  $S$  and value  $\rho_1^M + \dots + \rho_n^M$  (which has type text due to the well-formedness).
- $S$  is rendered as the text  $S$ .
- $\underline{s}$  and  $\underline{v}$  are rendered as the text  $M(s)$  or  $M(v)$ , respectively.
- $\underline{o}^p$  is rendered by applying the rendering algorithm recursively to  $M(o)$  and  $p$ .
- $\text{for}(\underline{l}, I, \rho_1 \dots \rho_r)\{\rho'_1 \dots \rho'_s\}$  is rendered by the following algorithm:
  1. Let  $sep := \rho_1^M + \dots + \rho_r^M$  and  $t$  be the length of  $M(l)$ .
  2. For  $i = 1, \dots, t$ , let  $R_i := \rho'_1^{M^i} + \dots + \rho'_s^{M^i}$ .

3. If  $I = 0$ , return nothing and stop. If  $I$  is negative, reverse the list  $R$ , and invert the sign of  $I$ .
4. Return  $R_I + sep + R_{2*I} \dots + sep + R_T$  where  $T$  is the greatest multiple of  $I$  smaller than or equal to  $t$ .

Here the match context  $M_i^t$  arises from  $M$  as follows

- replace  $l \mapsto (X_1 \dots X_t)$  with  $l \mapsto X_i$ ,
- for every  $l/q \mapsto (X_1 \dots X_t)$  in  $M$ : replace it with  $q \mapsto X_i$ , and remove a possible previously defined match for  $q$ .

*Example* Consider the example introduced in Sect. 2. There we have

$$\begin{aligned}\omega = & @(\text{int}, @(\text{iv}, a_1, b_1), \dots, @(\text{iv}, a_n, b_n), \\ & \beta(\text{lam}, v(x_1), \dots, v(x_n), \alpha(@(\text{plus}, @(\text{sin}, v(x_1)), v(x_2)), \text{color} \mapsto \text{red})))\end{aligned}$$

And  $\Pi$  is the given list of notation definitions. Let  $A = (\text{format} = \text{latex})$ . Matching  $\omega$  against the patterns in  $\Pi$  succeeds for the first notation definitions and yields the following match context  $M$ :

$$\begin{aligned}\text{ranges} \mapsto & (@(\text{iv}, a_1, b_1), \dots, @(\text{iv}, a_n, b_n)), \text{ranges}/\text{a} \mapsto (a_1, \dots, a_n), \\ \text{ranges}/\text{b} \mapsto & (b_1, \dots, b_n), \text{f} \mapsto \alpha(@(\text{plus}, @(\text{sin}, v(x_1)), v(x_2)), \text{color} \mapsto \text{red}), \\ \text{vars} \mapsto & (v(x_1), \dots, v(x_n)), \text{vars}/\text{x} \mapsto (x_1, \dots, x_n)\end{aligned}$$

In the second step, a specific rendering is chosen. In our case, there is only one rendering, which matches the required rendering context  $A$ , namely

$$\rho = \text{for}(\underline{\text{ranges}}) \{ \backslash \text{int} \cdot \{ \underline{\text{a}}^\infty \} \hat{\cdot} \{ \underline{\text{b}}^\infty \} \} \ \underline{\text{f}}^\infty \ \text{for}(\underline{\text{vars}}, -1) \{ \underline{\text{d}} \ \underline{\text{x}}^\infty \} )^{-\infty}$$

To render  $\rho$  in match context  $M$ , we have to render the three components and concatenate the results. Only the iterations are interesting. In both iterations, the separator  $sep$  is empty; in the second case, the step size  $I$  is  $-1$  to render the variables in reverse order.

## 4 Choosing Notation Definitions Extensionally

In the last sections we have seen how collections of notation definitions induce rendering functions. Now we permit users to define the set  $\Pi$  of available notation definitions extensionally. In the following, we discuss the collection of notation definitions from various sources and the construction of  $\Pi_\omega$  for a concrete mathematical object  $\omega$ .

### 4.1 Collecting Notation Definitions

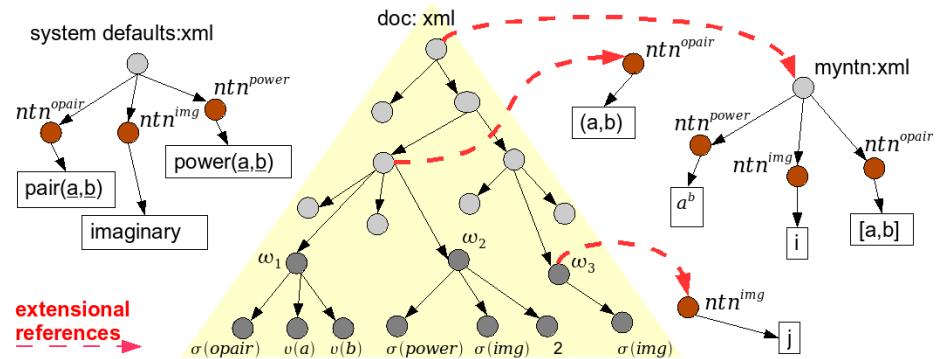
The algorithm for the collection of notation definitions takes as input a tree-structured document, e.g., an XML document, an object  $\omega$  within this document,

and a totally ordered set  $\mathcal{S}^N$  of source names. Based on the hierarchy proposed in [NW01b], we use the source names  $EC$ ,  $F$ ,  $Doc$ ,  $CD$ , and  $SD$  explained below. The user can change their priorities by ordering them.

The collection algorithm consists of two steps: The collection of notation definitions and their reorganization. In the first step the notation definitions are collected from the input sources according to the order in  $\mathcal{S}^N$ . The respective input sources are treated as follows:

- $EC$  denotes the **extensional context**, which associates a list of notation definitions or containers of notation definitions to every node of the input document. The effective extensional context is computed according to the position of  $\omega$  in the input document (see a concrete example below).  $EC$  is used by authors to reference their individual notation context.
- $F$  denotes an **external notation document** from which notation definitions are collected.  $F$  can be used to overwrite the author's extensional context declarations.
- $Doc$  denotes the **input document**. As an alternative to  $EC$ ,  $Doc$  permits authors to embed notation definitions into the input document.
- $CD$  denotes the **content dictionaries** of the symbols occurring in  $\omega$ . These are searched in the order in which the symbols occur in  $\omega$ . Content dictionaries may include or reference default notation definitions for their symbols.
- $SD$  denotes the **system default** notation document, which typically occurs last in  $\mathcal{S}^N$  as a fallback if no other notation definitions are given.

In the second step the obtained notation context  $\Pi$  is reorganized: All occurrences of a pattern  $\varphi$  in notation definitions in  $\Pi$  are merged into a single notation definition preserving the order of the  $(\lambda^i \rho)^p$  (see a concrete example below).



**Fig. 2.** Collection Example

We base our further illustration on the input document in Fig. 2 figure above, which includes three mathematical objects. For simplicity, we omit the `cbase` and `cd` attributes of symbols.

$$\omega_1 : @(\sigma(opair), v(a), v(b)) \rightsquigarrow (a, b) \quad \omega_2 : @(\sigma(power), \sigma(img), 2) \rightsquigarrow i^2 \quad \omega_3 : \sigma(img) \rightsquigarrow j$$

The dashed arrows in the figure represent extensional references: For example, the `ec` attribute of the document root `doc` references the notation document “`myntn`”, which is interpreted as a container of notation definitions.

We apply the algorithm above with the input object  $\omega_3$  and  $S^N = (EC, SD)$  and receive  $\Pi_{\omega_3}$  in return. For simplicity, we do not display context annotations and precedences.

- 
1. We collect all notation definitions yielding  $\Pi_{\omega_3}$
  - 1.1 We collect notation definitions from  $EC$ 
    - 1.1.1 We compute the effective extensional context based on the position of  $\omega_3$  in the input document:  $ec(\omega_3) = (ntn^{img}, myntn)$
    - 1.1.2 We collect all notation definition based on the references in  $ec(\omega_3)$ :  
 $\Pi_{\omega_3} = (ntn^{img}, ntn^{power}, ntn^{img}, ntn^{opair})$
  - 1.2. We collect notation definitions from  $SD$  and append them to  $\Pi_{\omega_3}$   
 $\Pi_{\omega_3} = (ntn^{img}, ntn^{power}, ntn^{img}, ntn^{opair}, ntn^{opair}, ntn^{img}, ntn^{power})$
  - 1.3. The collected notation definition form the notation context  $\Pi_{\omega_3}$ ,  
 $\Pi_{\omega_3} = (\varphi_1 \vdash j, \varphi_2 \vdash \underline{a}^b, \varphi_1 \vdash i, \varphi_3 \vdash [a, b], \varphi_3 \vdash pair(a, b), \varphi_1 \vdash imaginary, \varphi_2 \vdash power(a, b))$
  2. We reorganize  $\Pi_{\omega_3}$  yielding  $\Pi'_{\omega_3}$   
 $\Pi'_{\omega_3} = (\varphi_1 \vdash j, i, imaginary; \varphi_2 \vdash \underline{a}^b, power(a, b); \varphi_3 \vdash [a, b], pair(a, b))$
- 

To implement  $EC$  in arbitrary XML-based document formats, we propose an `ec` attribute in a namespace for notation definitions, which may occur on any element. The value of the `ec` attribute is a whitespace-separated list of URIs of either notation definitions or any other document. The latter is interpreted as a container, from which notation definitions are collected. The `ec` attribute is empty by default. When computing the effective extensional context of an element, the values of the `ec` attributes of itself and all parents are concatenated, starting with the inner-most.

## 4.2 Discussion of Collection Strategies

In [KLM<sup>+</sup>08], we provide the specific algorithms for collecting notation definitions from  $EC$ ,  $F$ ,  $Doc$ ,  $CD$  and  $SD$  and illustrate the advantages and drawbacks of basing the rendering on either one of the sources. We conclude with the following findings:

1. Authors can write documents which only include content markup and do not need to provide any notation definitions. The notation definitions are then collected from  $CD$  and  $SD$ .
2. The external document  $F$  permits authors to store their notation definitions centrally, facilitating the maintenance of notational preferences. However, authors may not specify alternative notations for the same symbol on granular document levels.

3. Authors may use the content dictionary defaults or overwrite them by providing  $F$  or  $Doc$ .
4. Authors may embed notation definitions inside their documents. However, this causes redundancy inside the document and complicates the maintenance of notation definitions.
5. Users can overwrite the specification inside the document with  $F$ . However, that can destroy the meaning of the text, since the granular notation contexts of the authors are replaced by only one alternative declaration in  $F$ .
6. Collecting notation definitions from  $F$  or  $Doc$  has benefits and drawbacks. Since users want to easily maintain and change notation definitions but also use alternative notations on granular document levels, we provide  $EC$ . This permits a more controlled and more granular specification of notations.

## 5 Choosing Renderings Intensionally

The extensional notation context declarations do not support authors to select between alternative renderings inside one notation definition. Consequently, if relying only on this mechanism, authors have to take extreme care about which notation definition they reference or embed. Moreover, other users cannot change the granular extensional declarations in  $EC$  without modifying the input document. They can only overwrite the author's granular specifications with their individual styles  $F$ , which may reduce the understandability of the document.

Consequently, we need a more *intelligent, context-sensitive* selection of renderings, which lets users guide the selection of alternative renderings. We use an intensional rendering context  $\Lambda$ , which is matched against the context annotations in the notation definitions. In the following, we discuss the collection of contextual information from various sources and the construction of  $\Lambda_\omega$  for a concrete mathematical object  $\omega$ .

### 5.1 Collecting Contextual Information

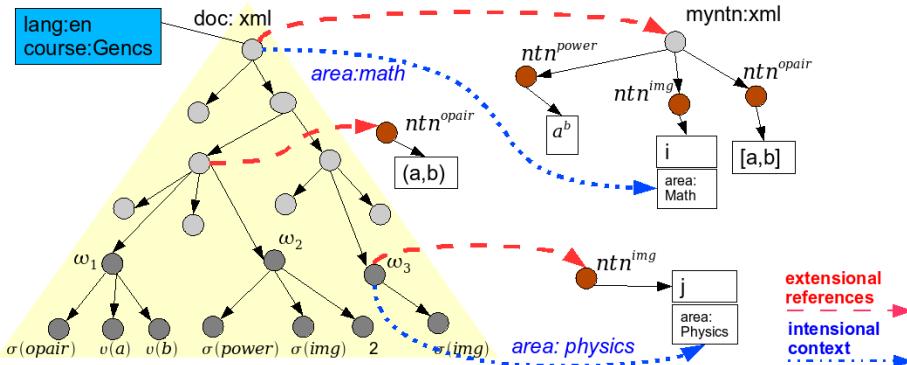
We represent contextual information by contextual key-value pairs, denoted by  $(d_i = v_i)$ . The key represents a *context dimension*, such as *language*, *level of expertise*, *area of application*, or *individual preference*. The value represents a *context value* for a specific context dimension. The algorithm for the context-sensitive selection takes as input an object  $\omega$ , a list  $L$  of elements of the form  $(\lambda : \rho)^p$ , and a totally ordered set  $\mathcal{S}^C$  of source names. We allow the names  $GC$ ,  $CCF$ ,  $IC$ , and  $MD$ . The algorithm returns a pair  $(\rho, p)$ .

The selection algorithm consists of two steps: The collection of contextual information  $\Lambda_\omega$  and the selection of a rendering. In the first step  $\Lambda_\omega$  is computed by processing the input sources in the order given by  $\mathcal{S}^C$ . The respective input sources are treated as follows:

- $GC$  denotes the **global context** which provides contextual information during *rendering time* and overwrites the author's intensional context declarations. The respective  $(d_i = v_i)$  can be collected from a user model or are explicitly entered.  $GC$  typically occurs first in  $\mathcal{S}^C$ .

- $CCF$  denotes the **cascading context files**, which permit the contextualization analogous to cascading stylesheets[Cas99].
- $IC$  denotes the **intensional context**, which associates a list of contextual key-value pairs ( $d_i = v_i$ ) to any node of the input document. These express the author’s intensional context. For the implementation in XML formats, we use an `ic` attribute similar to the `ec` attribute above, i.e., the effective intensional context depends on the position of  $\omega$  in the input document (see a concrete example below).
- $MD$  denotes **metadata**, which typically occurs last in  $S^C$ .

In the second step, the rendering context  $\Lambda_\omega$  is matched against the context annotations in  $L$ . We select the pair  $(\rho, p)$  whose corresponding context annotation satisfies the intensional declaration best (see [KLM<sup>+</sup>08] for more details).



**Fig. 3.** Rendering Example

In Fig. 3, we continue our illustration on the given input document. The dashed arrows represent extensional references, the dashed-dotted arrows represent intensional references, i.e., implicit relations between the `ic` attributes of the input document and the context-annotations in the notation document. A global context is declared, which specifies the language and course dimension of the document. We apply the algorithm above with the input object  $\omega_3$ ,  $S^C = (GC, IC)$ , and a list of context annotations and rendering pairs based on the formerly created notation context  $\Pi_{\omega_3}$ . For convenience, we do not display the system’s default notation document and the precedences.

- 
1. We compute the intensional rendering context
    - 1.1. We collect contextual information from  $GC$   
 $\Lambda_{\omega_3} = (\text{lang} = \text{en}, \text{course} = \text{GenCS})$
    - 1.2. We collect contextual information from  $IC$  and append them to  $\Lambda_{\omega_3}$   
 $\Lambda_{\omega_3} = (\text{lang} = \text{en}, \text{course} = \text{GenCS}, \text{area} = \text{physics}, \text{area} = \text{math})$

2. We match the rendering context against the context annotations of the input list  $L$  and return the rendering with the best matching context annotation:

$$L = [(\lambda_0 = j), (\lambda_1 = i), (\lambda_2 = \text{imaginary})] \\ \lambda_0 = (\text{area} = \text{physics}), \lambda_1 = (\text{area} = \text{maths}), \text{ and } \lambda_2 = \emptyset$$


---

For simplicity, we compute the similarity between  $\Lambda_{\omega_3}$  and  $\lambda_i$  based on the number of similar  $(d_i = v_i)$ :  $\lambda_0$  includes  $(\text{area} = \text{physics})$ .  $\lambda_1$  includes  $(\text{area} = \text{math})$ .  $\lambda_2$  is empty.  $\lambda_0$  and  $\lambda_1$  both satisfy one  $(d_i = v_i)$  of  $\Lambda_{\omega_3}$ . However, since  $(\lambda_0 = \rho_0)$  occurs first in  $\Pi_{\omega_3}$ , the algorithm returns  $\rho_0$ .

## 5.2 Discussion of Context-Sensitive Selection Strategies

In [KLM<sup>+</sup>08], we illustrate and evaluate the collection of contextual information from *GC*, *CCF*, *IC* and *MD* in detail. We conclude with the following findings:

1. The declaration of a *global context* provides a more intelligent intensional selection between alternative  $(\lambda : \rho)^p$  triples inside one notation definition: The globally defined  $(d_i = v_i)$  are matched against the context-annotations  $\lambda$  to select an appropriate rendering  $\rho$ . However, the approach does not let users specify intensional contexts on granular levels.
2. Considering *metadata* is a more granular approach than the global context declaration. However, metadata may not be associated to any node in the input document and cannot be overwritten without modifying the input document. Moreover, the available context dimensions and values are limited by the respective metadata format.
3. The *intensional context* supports a granular selection of renderings by associating an intensional context to any node of the input document. However, the intensional references cannot be overwritten on granular document levels.
4. *Cascading Context Files* permit a granular overwriting of contexts.

## 6 Conclusion and Future Work

We introduced a representational infrastructure for notations in living mathematical documents by providing a flexible declarative specification language for notation definitions together with a rendering algorithm. We described how authors and users can extensionally extend the set of available notation definitions on granular document levels, and how they can guide the notation selection via intensional context declarations. Moreover, we discussed different approaches for collecting notation definitions and contextual information.

To substantiate our approach, we have developed prototypical implementations of all aspects of the rendering pipeline:

- The Java toolkit `mmlkit` [MMK07] implements the conversion of OPENMATH and Content-MATHML expressions to Presentation-MATHML. It supports the collection of *notation definitions* from various sources, constructs rendering contexts based on contextual annotations of the rendered object, identifies proper renderings for the conversion.

- The semantic wiki SWIM [Lan08] supports the collaborative browsing and editing of notation definitions in OPENMATH content dictionaries.
- The *panta rhei* [Mü07] reader integrates mmlkit to present mathematical documents, provides facilities to categorize and describe notations, and uses these context annotations to adapt documents.

We will invest further work into our implementations as well as the evaluation of our approach. In particular, we want to address the following challenges:

- Write Protection: In some cases, users should be prevented to overwrite the author’s declaration. On the contrary, static notations reduce the flexibility and adaptability of a document (see [KLM<sup>+</sup>08] for more details).
- Consistency: The flexible adaptation of notations can destroy the meaning of documents, in particular, if we use the same notation to denote different mathematical concepts.
- Elision: In [KLM<sup>+</sup>08], we have already adapted the elision of arbitrary parts of formulae from [KLR07].
- Notation Management: Users want to reuse, adapt, extend, and categorize notation definitions (see [KLM<sup>+</sup>08] for more details).
- Advanced notational forms: Ellipses and Andrews’ dot are examples of advanced notations that we cannot express yet.

**Acknowledgments** Our special thanks go to Normen Müller for the initial implementation of the presentation pipeline. We would also like to thank Alberto González Palomo and Paul Libbrecht for the discussions on their work. This work was supported by JEM-Thematic-Network ECP-038208.

## References

- ABC<sup>+</sup>03. Ron Ausbrooks, Stephen Buswell, David Carlisle, Stéphane Dalmas, Stan Devitt, Angel Diaz, Max Froumentin, Roger Hunter, Patrick Ion, Michael Kohlhase, Robert Miner, Nico Poppelier, Bruce Smith, Neil Soiffer, Robert Sutor, and Stephen Watt. Mathematical Markup Language (MathML) version 2.0 (second edition). W3C recommendation, World Wide Web Consortium, 2003. Available at <http://www.w3.org/TR/MathML2>.
- BCC<sup>+</sup>04. Stephen Buswell, Olga Caprotti, David P. Carlisle, Michael C. Dewar, Marc Gaetano, and Michael Kohlhase. The Open Math standard, version 2.0. Technical report, The Open Math Society, 2004. <http://www.openmath.org/standard/om20>.
- Caj93. Florian Cajori. *A History of Mathematical Notations*. Courier Dover Publications, 1993. Originally published in 1929.
- Cas99. Cascading Style Sheets. <http://www.w3.org/Style/CSS/>, 1999.
- Kay06. Michael Kay. XSL Transformations (XSLT) Version 2.0. W3C Candidate Recommendation, World Wide Web Consortium (W3C), June 2006. Available at <http://www.w3.org/TR/2006/CR-xslt20-20060608/>.
- KLM<sup>+</sup>08. Michael Kohlhase, Christoph Lange, Christine Müller, Normen Müller, and Florian Rabe. Adaptation of notations in living mathematical documents. KWARC report, Jacobs University Bremen, 2008.

- KLR07. Michael Kohlhase, Christoph Lange, and Florian Rabe. Presenting mathematical content with flexible elisions. In Olga Caprotti, Michael Kohlhase, and Paul Libbrecht, editors, *OpenMath/ JEM Workshop 2007*, 2007.
- KMM07. Michael Kohlhase, Christine Müller, and Normen Müller. Documents with flexible notation contexts as interfaces to mathematical knowledge. In Paul Libbrecht, editor, *Mathematical User Interfaces Workshop 2007*, 2007.
- Koh06. Michael Kohlhase. OMDoc – An open markup format for mathematical documents [Version 1.2]. Number 4180 in LNAI. Springer Verlag, 2006.
- Lan08. Christoph Lange. Mathematical Semantic Markup in a Wiki: The Roles of Symbols and Notations. In Christoph Lange, Sebastian Schaffert, Hala Skaf-Molli, and Max Völkel, editors, *Proceedings of the 3<sup>rd</sup> Workshop on Semantic Wikis, European Semantic Web Conference 2008*, Costa Adeje, Tenerife, Spain, June 2008.
- MLUM05. Shahid Manzoor, Paul Libbrecht, Carsten Ullrich, and Erica Melis. Authoring Presentation for OPENMATH. In Michael Kohlhase, editor, *Mathematical Knowledge Management, MKM'05*, number 3863 in LNAI, pages 33–48. Springer Verlag, 2005.
- MMK07. Christine Müller, Normen Müller, and Michael Kohlhase. mmlkit - a toolkit for handling mathematical documents and MathML3 notation definitions. mmlkit v0.1 at <http://kwarc.info/projects/mmlkit>, seen November 2007.
- Mül07. Christine Müller. Panta Rhei. Project Home Page at <http://kwarc.info/projects/panta-rhei/>, seen August 2007.
- NW01a. Bill Naylor and Stephen M. Watt. Meta-Stylesheets for the Conversion of Mathematical Documents into Multiple Forms. In *Proceedings of the International Workshop on Mathematical Knowledge Management*, 2001.
- NW01b. Bill Naylor and Stephen M. Watt. Meta-Stylesheets for the Conversion of Mathematical Documents into Multiple Forms. In *Proceedings of the International Workshop on Mathematical Knowledge Management* [NW01a].
- POS88. IEEE POSIX, 1988. ISO/IEC 9945.
- SW06a. Elena Smirnova and Stephen M. Watt. Generating TeX from Mathematical Content with Respect to Notational Settings. In *Proceedings International Conference on Digital Typography and Electronic Publishing: Localization and Internationalization (TUG 2006)*, pages 96–105, Marrakech, Morocco, 2006.
- SW06b. Elena Smirnova and Stephen M. Watt. Notation Selection in Mathematical Computing Environments. In *Proceddings Transgressive Computing 2006: A conference in honor of Jean Della Dora (TC 2006)*, pages 339–355, 2006.
- Wol00. Stephen Wolfram. Mathematical notation: Past and future. In *MathML and Math on the Web: MathML International Conference*, Urbana Champaign, USA, October 2000. <http://www.stephenwolfram.com/publications/talks/mathml/>.

# Integrating Web Services into Active Mathematical Documents

Jana Giceva, Christoph Lange, and Florian Rabe

Computer Science, Jacobs University Bremen,  
`{j.giceva, ch.lange, f.rabe}@jacobs-university.de`

**Abstract.** Active mathematical documents are distinguished from traditional paper-oriented ones by their ability to interactively adapt to a reader’s inputs. This includes changes in the presentation of the content of the document as well as changes of that content itself.

We have developed the JOBAD architecture, a client/server infrastructure for active mathematical documents. A server-side module generates active documents, which a client-side JavaScript library makes accessible for user interaction. Further server-side modules – in the same backend, or distributed web services – dynamically respond to callbacks invoked when the user interacts with the client. These three components are tied together by the JOBAD active document format, which backwards-compatibly enhances MathML by information about interactivity.

JOBAD is designed to be modular in the specific web services offered. As examples, we present folding and elision in mathematical expressions, type and definition lookup of symbols, as well as conversion of physical units. We evaluate our framework with a case study where a large collection of lecture notes is served as an active document.

## 1 Introduction and Related Work

Documents are an important interface for distributing mathematical knowledge. Recently the technological development has shifted attention more and more towards digital documents and the added-value they can offer. This has led to a number of research efforts on interactive mathematical documents involving features such as adapting mathematical documents, interactive exercises, and connecting to mathematical web services.

The ActiveMath project investigates how to *aggregate documents* from a knowledge base such that the resulting document contains exactly the topics that the reader wants to learn and their prerequisites [Act08]. *Interactive Exercises* have been realized in ActiveMath and MathDox [GM08,CCK<sup>+</sup>08,CCJS05]. Here the user enters the result into a form and then gets feedback from a solution checker in the server backend. ActiveMath comes with its own web services [MGH<sup>+</sup>06], and MathDox has originally been designed for talking to computer algebra systems but can also connect to other services via MONET (see below). Gerdes et al. have developed a reusable exercise feedback service for exercises that has also been integrated with MathDox [GHJS08]. Besides supporting MathDox’s

own communication protocol, Gerdès's service also complies to the XML-RPC and JSON data exchange standards [GHJS08]. The services developed within the MathDox and ActiveMath projects, such as the ActiveMath course generator, are potentially open to any client, but have not been used with any frontend other than their primary one so far [Ull08, MGH<sup>+</sup>06].

There are also elaborate *web service architectures* for mathematical web services that have been designed for integration with many systems, such as the ones developed in the SCIEnce [SCI09] and MONET projects [Mon05]. SCIEnce explicitly targets symbolic computation and grid computing and does not consider documents as user interfaces. MONET is an architecture that in principle allows for any kind of mathematical web service. Still, mainly computational web services have been developed and evaluated within the framework. Web services can register with a central MONET broker that accepts requests, which do not directly call a web service but consist of a problem description (e.g., solve an equation, given as an OpenMath expression). The broker then forwards the request to the best-matching service. The above-mentioned MathDox allows for access to MONET web services via a document interface.

*Asynchronous communication* with a server backend (AJAX) allows for client/server interaction without submitting forms. It is a prerequisite for responsive browser-based applications: A client-side script can exchange small data packets with a server backend and insert responses from the server into the current page. This technique is employed by MathDox [CCK<sup>+</sup>08] and Gerdès's frontend to their feedback service [GHJS08].

Despite the efforts mentioned above, there is still a lot of static mathematical content on the web. Where documents act as frontends to web services, as in the above-mentioned systems, they have usually been designed to give access to a small selection of web services performing very specific tasks (mostly giving feedback to exercises and symbolic computation) – as is the case with ActiveMath and MathDox.

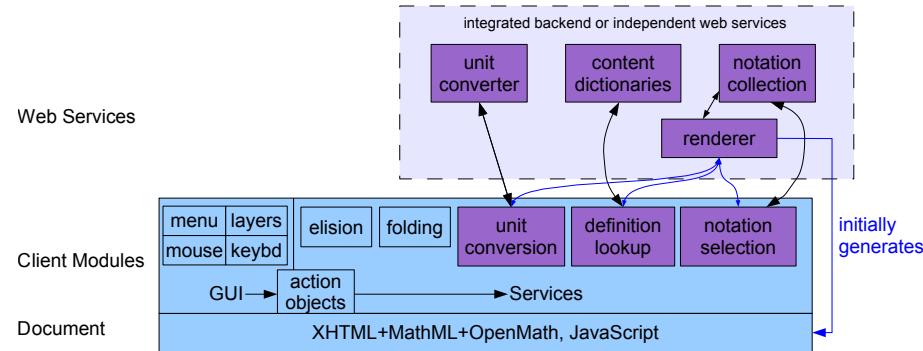
Our goal is to facilitate the integration of diverse web services into mathematical documents – inspired by the Web 2.0 technology of *mashups* [O'R05, AKTV08]. Originally, mashups were handcrafted JavaScripts pulling together web services from different sites. Since then several mashup development kits have been developed, e.g., Yahoo! Pipes [Yah09] or Ubiquity [Moz09]. We aim at a similar development kit for mathematical applications. Our vision of an *interactive document* is a document that the user can not just read, but adapt according to his preferences and interests *while* reading it – not only by customizing the display of the rendered document in the browser, but also by changing notations (which requires re-rendering) or retrieving additional information from services on the web. Consider a student reading lecture notes: Whenever he is not familiar with a mathematical symbol occurring in some formula, he should be able to look up its definition without opening another document, but right in his current reading context. Or consider the problem of converting between physical units (e.g., imperial vs. SI). There are lots of unit converters on the web (see [Str08]

for a survey), but instead of manually opening one and copying numbers into its entry form, we want to enable an in-place conversion.

In [KMR08,KMM07], we investigated how OMDoc documents containing content markup can be rendered as XHTML with embedded presentation MathML. The rendering was relative to context dimensions such as the native language of the reader or the field of knowledge. This approach used *notation definitions* to translate content markup into presentation markup. It focused on generating documents *before* the user gets to read them. In the present paper, we continue this line of research and introduce JOBAD as the client-side counterpart. JOBAD is a JavaScript API for OMDoc-based Active Documents. While its primary intended application is to be part of the active documents served by our server, it is independent of the server and can be flexibly reused to enable any mathematical document to interact with the reader or web services. Our contribution includes the JOBAD interactive document format, an XHTML+MathML-based interface language between server- and client-side computation.

In Section 2, we present the main component of JOBAD, a collection of small JavaScript modules that add interactive services to a mathematical document. Here we assume a broad notion of “service” including local interactive functionality as well as any service with an HTTP interface, regardless of whether it complies with a “heavyweight” web service standard like XML-RPC or MONET. In Section 3, we present several web services that we have implemented and describe how to integrate third-party services. In Section 4, we briefly describe a first JOBAD case study, and we conclude in Section 5.

## 2 An Architecture for Active Documents



**Fig. 1.** JOBAD Architecture. Note the central role of the rendering service, which both generates JOBAD-compliant documents and is needed for many other services.

The JOBAD architecture is divided into the actual mathematical services, the user interface elements, and generic communication and document manipulation

functions (see figure 1). On the client side, JOBAD consists of a JavaScript main module and one independent module for each service. The server controls the available functionality by loading a collection of service modules into the document. We distinguish three kinds of interactive services by the amount of data they exchange with the web service backend: 1. services that merely draw on data embedded into the document, 2. services that send a symbol identifier and an action verb to the backend in order to retrieve additional information, 3. and services that send complex mathematical content expressions to the backend. The decision which kind of service should be used for a particular functionality depends considerably on the format used for interactive documents: Some meta-information about the current appearance may be embedded into the document to permit local interaction, for example, alternative presentations or parallel content markup. If this is not feasible, further information must be embedded that instructs the JOBAD client how to retrieve the external document fragments.

## 2.1 A Document Format that Enables Services

The MathML specification leaves some details about the structure of a document to application developers [W3C]. Therefore, for JOBAD, we pose some additional requirements:

1. Alternative displays, among which the user can switch, **SHOULD**<sup>1</sup> be realized by `maction` elements and an `@actiontype` attribute that indicates the type of service [W3C, section 3.6.1]. Particularly, services that rewrite a formula **SHOULD** retain the previous state of the formula as an alternative to which the user can switch back.
2. Unless subterms are already part of a grouping operator (e. g. radicals, or super-/subscripts, or fractions; see [W3C, table 3.1.3.2] for a complete list), they **MUST** be grouped using the invisible `mrow` element (which is optional in MathML).
3. For services that need access to the semantics of mathematical expressions, the latter **MUST** be provided as parallel content markup [W3C, chapter 5.4]. There **MUST** be cross-links from the subterm-grouping elements of Item 2 to the corresponding content elements. Content elements **MAY** be annotated with additional attributions.
4. If services that directly operate on the presentation markup to customize its display require annotations that will be interpreted in a service-specific way, it may be considered impractical to add them to the parallel content markup and look them up there. In such a case, annotations **MAY** be added directly to presentation markup elements as attributes from another namespace (see [W3C, section 2.4]). It is **RECOMMENDED** that such annotations require considerably less additional space than parallel markup; otherwise parallel markup **SHOULD** be preferred.

---

<sup>1</sup> We use these capitalized keywords in accordance with RFC 2119 [Bra97]

Requirements 2 and 4 lead to a linear overhead in the size of the formulas (with a small factor), requirement 3 as well but with a factor of 2 to 4, as content markup, element IDs, and references to those IDs are added. Requirement 1 can lead to an exponential overhead when alternative displays are nested. This could be avoided if presentation MathML allowed for structure sharing between expressions (which only content MathML allows so far).

As these requirements are hard to satisfy by manual authoring, this markup is generated by our rendering service that generates presentation from content markup using pattern matching-based notation definitions for all symbols (cf. section 3.1). In the following, we will describe these requirements in more detail by discussing how they are used in a variety of services we implemented.

**Switching between Alternative Displays** An `maction` element can have multiple children, one of which is displayed at a time, controlled by the `@selection` integer attribute – whose value can be changed at runtime. The MathML specification suggests possible `@actiontype` values but does not prescribe a fixed semantics for them. We have introduced several values for that attribute and clearly specified the desired behavior of the services using them.

In particular, we use `maction`, with the action type `abbrev`, for author-defined abbreviations of complex expressions. Consider a physics document, where the author wants to provide  $W_{\text{pot}}(R)$  (potential energy) as an instructive abbreviation of the complex term  $\frac{-e^2}{4\pi\epsilon_0 R/2}$ . We introduce the OpenMath symbol `folding#abbrev` that serves as an attribution key. Our rendering algorithm uses the value of such an attribution as an alternative display.

For example,

```

<OMATTR>
<OMATP>
  <OMS cd="folding" name="abbrev"/>
   $W_{\text{pot}}(R)$ 
</OMATP>
   $\frac{-e^2}{4\pi\epsilon_0 R/2}$ 
</OMATTR>

```

is rendered as

```

<maction actiontype="abbrev" selection="1">
  
$$\frac{-e^2}{4\pi\epsilon_0 R/2}$$

  
$$W_{\text{pot}}(R)$$

</maction>
```

Other JOBAD services create mactions on the fly (using the name of the service as the value of the `actiontype` attribute) whenever they rewrite a term  $t$  into  $t'$  (e.g. by converting units; see Sect. 3.3). This allows for making interactions *undoable*: The previous state  $t$  of a term is preserved in the second, hidden child of the maction, and the user can switch back to it. Not only do interactions become *undoable* locally, but they also become *redoable*: When

information from a remote web service has been used to rewrite  $t \rightsquigarrow t'$ , as is the case with unit conversion, e.g., and the user switches back to  $t$ , he can recover  $t'$  without causing the information to be retrieved from the web once more, as it is still cached in the maction.

**Grouping Subterms for Interactive Folding** Besides author-defined abbreviations, we have implemented interactive folding of *arbitrary* subterms, so that a reader can hide them if he feels distracted. Any subterm that is properly grouped (see requirement 2 above) is eligible for folding. When the user requests folding of a subexpression for the first time, we put both the original subterm and its folded version into an maction element with actiontype folding for making the action undoable (see above).

As an example, consider the expression  $[1 + [2 \cdot x]]$ , where square brackets denote mrows. Suppose the user selects [part of] the subterm  $2 \cdot x$  or right-clicks somewhere in that term and requests it to be folded. Then, the formula will display as  $[1 + \dots]$ . Clicking on the dots and selecting the “unfold” action from the user interface (e.g. the context menu) will restore the original appearance.

**Cross-Linked Parallel Markup** MathML provides the ability to attach semantic annotations to presentation markup as so-called *parallel markup* [W3C, chapter 5.4]. We use this to obtain the content counterpart of presentation expressions selected by the user. The direction of such cross-links is generally unspecified in MathML, but we fix it to “presentation→content” as that is the most natural direction for looking up a formal representation of the user’s selection. Moreover, it is the only direction in which associative infix operators can be cross-linked, as multiple presentation operators have to link to one content symbol (cf. Fig. 2).

Therefore, we require links from symbols, numbers, identifiers, and subterm-grouping presentation elements to corresponding content elements, given by @xref attributes. If a content expression is to be looked up for a given selection of presentation elements, we find the closest common ancestor of all selected XML elements that carries an @xref attribute and dereference it to obtain the corresponding content expression. An example is given in Figure 2.

Since our rendering algorithm supports pattern matching-based and thus non-compositional translations from content to presentation markup, not every content subexpression corresponds to a presentation expression. For example, in the presentation element corresponding to  $\sin^2 x$ , there will be no subexpression pointing to the content expression  $\sin x$ .

**Lightweight Annotations for Flexible Elisions** The rendering algorithm that we introduced in [KMR08] enables a flexible elision of redundant brackets. We now describe how to utilize the output generated by that algorithm – and thus our rendering service, cf. Sect. 3.1 – for deferring the decision which brackets to elide until the document is read.

When rendering a content expression  $f(t_1, \dots, t_m)$ , brackets around the rendering of  $t_i = g(s_1, \dots, s_n)$  are redundant if the operator  $f$  binds weaker than

```

<semantics>
  <!-- a+b2 c -->
  <mrow xref="#E">
    <mi xref="#E.1">a</mi>
    <mo xref="#E.0">+</mo>
    <mrow xref="#E.2">
      <msup xref="#E.2.1">
        <mi xref="#E.2.1.1">b</mi>
        <mn xref="#E.2.1.2">2</mn>
      </msup>
      <mo xref="#E.2.0">&#x2062;
        <!-- INVISIBLE TIMES -->
      </mo>
      <mi xref="#E.2.2">c</mi>
    </mrow>
    <mo xref="#E.0">+</mo>
    <mi xref="#E.3">d</mi>
  </mrow>
</semantics>
<annotation-xml encoding="OpenMath">
  <OMA id="E">
    <OMS cd="arith1" name="plus"
      id="E.0"/>
    <OMV name="a" id="E.1"/>
    <OMA id="E.2">
      <OMS cd="arith1" name="times"
        id="E.2.0"/>
      <OMA id="E.2.1">
        <OMS cd="arith1" name="power"
          id="E.2.1.0"/>
        <OMV name="b" id="E.2.1.1"/>
        <OMI id="E.2.1.2">2</OMI>
      </OMA>
      <OMV name="c" id="E.2.2"/>
    </OMA>
    <OMV name="d" id="E.3"/>
  </OMA>
</annotation-xml>
</semantics>

```

**Fig. 2.** Parallel markup: Presentation markup elements point to content markup elements. The light gray range is the user's selection, with the start and end node in bold face. We first look up their closest common ancestor with `mrow`-like grouping property, and then look up its corresponding content markup – here: *E.2*

*g*. Binding strength is determined by comparing the *i*-th input precedence of *f* with the output precedence of *g*. Redundant brackets are retained in the output document, but annotated with the difference between these precedences as the *elision level*. Besides brackets, our rendering service supports other *elision groups*, e.g., for type annotations, some of which are essential whereas others can be inferred. Then brackets are the special case of the elision group `fence`.

All visible Presentation MathML elements and all grouping elements can carry the attributes `egroup` and `elevel` from the OMDoc namespace. The value of the former can be any string with the value `fence` being reserved for brackets. The value of the latter can be any integer or the strings `infinity` and `-infinity`. In case an element is member of multiple elision groups, the `egroup` and `elevel` attributes can contain a space-separated list. For example, a left bracket, annotated with elision information, looks as follows:

```
<mo fence="true" omdoc:egroup="fence" omdoc:elevel="100">(</mo>
```

Our elision service allows the user to choose one *visibility threshold* for each elision group. If  $T_g$  is the threshold of elision group *g*, then all elements of group *g* whose elision level is above  $T_g$  are invisible. This is realized by using `maction` elements with action type `elision` that switch between an expression and an invisible `mspace` element. This also permits a document to provide initial visibility status for its elements.

## 2.2 User Interface

JOBAD offers various user interface elements for input and output. A context menu can be requested by right-clicking for the object under the cursor or for the range of mouse-selected objects. A selection can be made in the usual way of dragging the mouse, or by repeated clicking on any part of a formula. In the latter case, the selection is extended step by step, always advancing to the parent grouping element. While performing actions on the current selection makes sense for services like folding or definition lookup (cf. Sect. 3.2), other services, such as elision, are also made available globally: If desired, bracket elision can be controlled document-wide by hotkeys from 0 (no redundant brackets displayed) to 9 (all redundant brackets displayed), and a collapsible toolbar placed next to each formula offers one slider per elision group for controlling the thresholds locally.

Calls to services are represented by generic action objects, which allows for providing diverse access to them. The same elision action can, e. g., be triggered via a local context menu, from a formula-local toolbar, and via a global keyboard shortcut.

Besides rewriting formulae, JOBAD offers tooltip-like popups for displaying information on demand. These can be annotations hidden in the document, but we mainly use it for displaying responses from web services, such as the definition of a symbol that the user wanted to look up (cf. Sect. 3.2).

## 3 Web Services and their Integration

The easiest way of realizing a mathematical web service is to expose functionality via an HTTP interface. When adopting the REST pattern ([Fie00]), URLs directly represent mathematical resources (e.g., OpenMath symbols). This can be used, for example, to retrieve the definition of a symbol. We call such services *symbol-based*. More complex services act on the selected expression. In those cases we use parallel markup to obtain the corresponding content expression and include it in the body of the HTTP request. We call such services *expression-based*.

In the JOBAD framework, we do not commit to a fixed set of web services. Rather do we specify a way of how a JOBAD-compliant document server advertises available web services. For each service (client-side service or web service), the document server chooses to offer in the active documents it serves, it MUST serve the corresponding JOBAD JavaScript module to the client. In the head of an active document, each JavaScript module MUST be initialized. To modules that access web services, a description of a web service backend they can connect to MUST be provided. In the case of an integrated backend, these can be components of that backend, but it can also be remote web services that the document server is aware of.

The description of a symbol-based service MUST consist of a name that is displayed to the user and a URL that invokes the service. The URL MAY contain placeholders for cdbase, cd, and name of the symbol. Similarly, the description of an expression-based service MUST consist of a name and a URL.

**Listing 1.1.** Service initialization code in a document

```
<!-- utility functions (module loading , document manipulation ,
client/server communication) -->
<script src="../scripts/jobad.js"/>
<!-- our own initialization follows -->
<script type="text/javascript">
// GUI elements to be enabled
jobadInit("contextmenu"); // loads the context menu
// In-document services
jobadInit("elision");
// Web services
jobadInit("definition-lookup", "Look_up_definition",
"http://jobad.mathweb.org/backend?action=definition-lookup
&cdbase=$cdbase&cd=$cd&name=$name");
</script>
```

### 3.1 Rendering

The rendering service is a prerequisite for making output from other services human-readable. In its simplest form, it accepts as input (in the body of an HTTP POST request) a fragment of OpenMath and returns the result of rendering it to JOBAD-enriched Presentation MathML. In the following, we will use  $\text{render}(c)$  for the result of rendering a content markup fragment  $c$ .

Our rendering service is implemented using the JOMDoc library, which implements the rendering algorithm described in [KLM<sup>+</sup>09,KMR08]. It has access to a collection of *notation definitions*, which map content markup patterns to presentation markup templates [KMR08].

### 3.2 Definition and Type Lookup

The definition lookup service sends the ID of a symbol  $\sigma$  to the server and expects as a response a content-markup formula containing a term that defines  $\sigma$ . The type of a symbol can be looked up analogously. Our implementation uses the RESTful URI format introduced at the beginning of this section. The information that we want to look up is encoded by the value of the *action* parameter, either *definition-lookup* or *type-lookup*. In the following, we will use  $\text{def}(\sigma)$  and  $\text{type}(\sigma)$  for the definition of a symbol, or the type, resp., as looked up by this service.

On the server side, the lookup is enabled by representing content dictionaries (CDs) in OMDoc [Koh06]. There, a symbol with type declaration and definition is represented as shown below, which allows for easy retrieval, e.g., using XPath. The situation in an OpenMath CD is similar, but as “definitional FMPs” have not yet been specified, there is no way of identifying a definition of a symbol among its various mathematical properties.

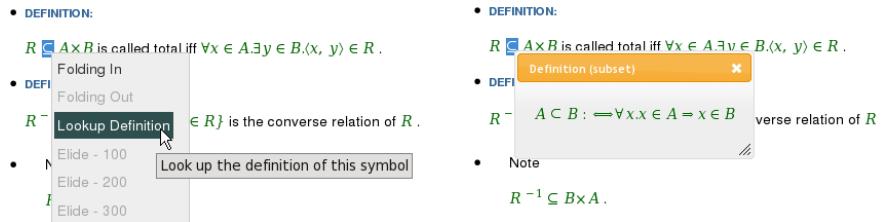
```
<!-- Content markup omitted here to save space -->
<symbol name="sin">
```

```

<type><OMOBJ> $\mathbb{C} \rightarrow \mathbb{C}$ </OMOBJ></type>
</symbol>
<definition for="#sin" type="simple">
<OMOBJ> $\sin z = \frac{1}{2i}(e^{iz} - e^{-iz})$ </OMOBJ>
</definition>

```

Our current client-side implementation displays  $\text{render}(\text{def}(\sigma))$  in a tooltip overlay at the cursor position. Alternatively, *definition expansion* is possible, which replaces the selected occurrence of a symbol with its definition and re-renders the formula. The original formula before definition expansion is kept as an action alternative using the `actiontype definition-expansion`. In contrast to definition expansion, definition lookup can also be offered for more complex types of definitions such as the implicit and inductive definitions of OMDoc [Koh06, chapter 15.2].



**Fig. 3.** Looking up a definition (left: selecting the action, right: the result); example taken from our lecture notes; cf. Sect. 4

As the desired MIME type of the response can be given in the HTTP request header for so-called *content negotiation*, we can distinguish requests for content markup from requests for a rendered formula while still using the same URL:

```

GET /backend?action=lookup-definition
&cdbase=...&cd=transcl&name=sin HTTP/1.1
Host: jobad.mathweb.org
Accept: application/openmath+xml

```

Analogously, the MIME type `application/xhtml+xml` would be used to obtain a response rendered in XHTML with Presentation MathML.

### 3.3 Unit Conversion

The unit conversion service assumes the OpenMath encoding for units as specified in [DN03]: Base units are symbols in special CDs; derived units can be formed by multiplication or division of base units with numeric factors or other base units. For example:

```

<OMA>
  <OMS cd="arith1" name="times"/>

```

```

<OMI>1</OMI>
<OMS cd="units_metric1" name="metre"/>
</OMA>

```

The unit conversion service accepts one such expression  $o$ , plus a target unit  $u$ . If a conversion is possible, the result is returned as an OpenMath expression, which we denote by  $\text{uc}(o, u)$ . On the client side, this result has to be integrated into the current formula. Let  $p$  with  $o = c(p)$  be the presentation markup that the user selected; then we add  $p' = \text{render}(\text{uc}(o, u))$  as an maction-alternative for  $p$  to the document.

We have not implemented our own unit converter but use the one developed by Stratford and Davenport [SD08,Str08], which performs conversions according to the OpenMath FMPs of the unit symbols involved. In its current version, their web service does not talk OpenMath but uses string input/output, so we have to convert values between their OpenMath and string representation (e.g. “1 metre”).

### 3.4 Integrated Backends and Environments

For a clean conceptual model, we have treated our web services separately. From an efficiency point of view it does, however, make sense to arrange multiple services in an integrated backend. Consider unit conversion: Stratford’s unit conversion web services internally relies on OpenMath CDs that declare one symbol per unit and define conversion rules for obtaining derived units [Str08]. Definitions of symbols are looked up from CDs as well. Last but not least, the rendering service needs notation definitions for the unit symbols, and CD authors often provide default notations for their symbols. Thus, offering those three services independently requires redundantly storing knowledge about symbols in three places. An integrated backend also saves time, as can be seen for definition lookup: With separate lookup and rendering services, the client-side active document has to connect to two web services in succession. An integrated backend could, however, offer readily rendered definitions by composing two of its internal functions and only minimally extending its external HTTP interface (cf. Sect. 3.2). We have implemented an integrated backend that performs rendering and definition lookup.

## 4 Evaluation

Proof-of-concept demonstrations of individual JOBAD services can be tried at the JOBAD web site [JOB09]. Besides that, we conducted two evaluations to analyze the feasibility and scalability of our framework: 1. We loaded a large OMDoc document into our server and activated the elision, subterm folding, and definition lookup services. 2. We integrated an external unit conversion service, which was added after the main phase of the development, to get an understanding of the investment needed to integrate further services.

The former involved the complete lecture notes of a first-year undergraduate computer science course. These lecture notes are originally maintained in L<sup>A</sup>T<sub>E</sub>X with semantic annotations, which can be automatically converted to OMDoc [Koh08]. The annotations in the source documents comprise content-markup formulae, informal definitions of symbols, and notation definitions. The OMDoc representation is then rendered into the JOBAD format, which is viewed using the JOBAD client, which offers flexible bracket elision, subterm folding, and definition lookup [JOB09]. So far, we have used this as a stress test, but for the Fall lecture we plan an evaluation where one group of our students will work with the static XHTML version of the lecture notes and a second group with the JOBAD-enriched active document.

The most complicated step in the latter evaluation was adapting the string-oriented interface of Stratford's unit conversion web service to our OpenMath interface. Most of the other required functionality turned out to be already available and just had to be composed. We chose the context menu interface and added a submenu containing the target units<sup>2</sup>. Checking whether the selected term was a quantity with a unit reduced to looking up its corresponding content markup (cf. Sect. 2.1) and performing a simple XPath node test on the latter. Sending a string to the web service and waiting for the response is a standard JavaScript function. Rendering the result of conversion (after converting it back to OpenMath) is done by another service. Finally, replacing two XML subtrees in a formula (both in the presentation and the content markup) and hiding the previous presentation tree in a maction, is a utility function provided by the JOBAD core and also used by other services.

## 5 Conclusion and Future Work

We presented JOBAD, our architecture for active mathematical documents. Our documents are generated dynamically from content-markup and viewed in a web browser, via which the reader can change interactively both content and form of the document. JOBAD constitutes the reader interaction component of our research group's framework for mathematical documents. As such it is fully integrated into the authoring [Koh08,Lan08], notation management [KMR08], and storage [TNT09] work flows developed by our group.

We gently extended the Presentation MathML format to create an interface language, in which the document server can embed into the served document information about interactivity or instructions how to retrieve that information. This extension is backwards compatible in the sense that markup is still valid MathML, and switching off JavaScript yields the same static documents as before.

We have implemented and evaluated an initial set of services that constitute a representative selection of the possibilities we envision. Folding and flexible elision work locally, type and definition lookup retrieve additional information based on

---

<sup>2</sup> A static list at the moment; obtaining admissible target units for a given input is neither supported by our client-side implementation nor by the unit conversion web service at the moment.

a symbol URI, unit conversion sends content markup object as an argument to web service. The former service is based on presentation markup generated by the server a priori. For the latter two, the JOBAD modules are passed initialization parameters that instruct them about the server and its URL format. For the latter one, parallel markup is utilized to obtain the content representation of a presentation expression.

A specific design feature of JOBAD is its extensibility. Offering new services for documents in the JOBAD interactive document can be achieved by adding very little new JavaScript code. Adding new user interaction components and binding them to JOBAD services is possible with minimal effort. Finally, the JOBAD client code requires only very little properties of the specific server backends, so that the same client can be easily used with different servers even in the same document.

Future work will be based on this, and we intend to rapidly develop more services. Due to the modularity of our framework, we expect that this work load can be divided into small and manageable units that can be handled efficiently by students. In particular, we intend to approach the following services:

**Notation selection:** Our rendering service can already annotate every rendered symbol with a reference to the notation definition in the backend that was used for rendering it [KLM<sup>+</sup>09]. This information can be used to ask the backend for alternative notations, to allow the user to select from them, and have the current formula re-rendered accordingly.

**Guided tour (extension of lookup):** This service generates a linear tutorial containing an explanation of every symbol in the current selection, and of every symbol occurring in these explanations, and so on, until some foundational theory is reached.

**Flattening:** Many documents consist of components that are combined by a module system (see [RK08]). A flattening service replaces import links with the (possibly translated) copy of the imported document.

**Search:** Our group has developed a semantic search engine for mathematical formulae [KAJ<sup>+</sup>08]. Therefore, a service that searches the web (or the server database) for the selected expression will be easy to realize.

**Links to web resources:** The OpenMath wiki [Lan09] not only provides symbol definitions, but also hosts discussions about them. Its architecture allows for linking symbols to further web resources, e.g. Wikipedia articles about mathematical concepts, which can then be made available in a document.

**Adaptive display of statement-level structures:** On the level of definitions, theorems, and proofs, we generate a different kind of parallel markup from OM-Doc sources, namely XHTML+RDFa [ABMP08]. We have already used this for visualizing rhetorical structures in mathematical documents (cf. [Gic08]; demo available at [JOB09]) and plan to extend it to structured proofs.

**Editing:** Our group has developed the Sentido formula editor [LGP08]. An edit service will pass the selected term to a Sentido popup window and eventually replace it in the current document.

**Saving:** After a user has adapted a document, it is desirable to upload its configuration to the database.

Furthermore, we will integrate the JOBAD architecture into our various integrated document management systems, such as the semantic wiki SWiM [LGP08], and the panta rhei document browser and community tool [pan].

*Acknowledgments :* The authors would like Christine Müller for fruitful discussions on notation selection, Jan Willem Knopper for hints on designing a modular JavaScript library, as well as David Carlisle for clarifications on MathML.

## References

- [ABMP08] B. Adida, M. Birbeck, S. McCarron, and S. Pemberton. RDFa in XHTML: Syntax and processing. Recommendation, W3C, 2008.
- [ACR<sup>+</sup>08] S. Autexier, J. Campbell, J. Rubio, V. Sorge, M. Suzuki, and F. Wiedijk, editors. *Intelligent Computer Mathematics, AISC, Calculemus, MKM*, number 5144 in LNAI. Springer, 2008.
- [Act08] ACTIVEMATH, 2008. <http://www.activemath.org/>.
- [AKTV08] A. Ankolekar, M. Krötzsch, T. Tran, and D. Vrandečić. The two cultures: Mashing up Web 2.0 and the Semantic Web. *Web Semantics*, 6(1), 2008.
- [Bra97] S. Bradner. Key words for use in RFCs to indicate requirement levels. RFC 2119, Internet Engineering Task Force, 1997.
- [CCJS05] A. M. Cohen, H. Cuypers, D. Jibetean, and M. Spanbroek. Interactive learning and mathematical calculus. In M. Kohlhase, editor, *Mathematical Knowledge Management (MKM)*, number 3863 in LNAI. Springer, 2005.
- [CCK<sup>+</sup>08] H. Cuypers, A. M. Cohen, J. W. Knopper, R. Verrijzer, and M. Spanbroek. MathDox, a system for interactive Mathematics. In *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications*. AACE, 2008.
- [DN03] J. H. Davenport and W. A. Naylor. Units and dimensions in OpenMath. <http://www.openmath.org/documents/Units.pdf>, 2003.
- [Fie00] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
- [GHJS08] A. Gerdes, B. Heeren, J. Jeuring, and S. Stuurman. Feedback services for exercise assistants. Technical Report UU-CS-2008-018, Utrecht University, 2008.
- [Gic08] J. Giceva. Capturing Rhetorical Aspects in Mathematical Documents using OMDoc and SALT. Technical report, Jacobs University, DERI Galway, 2008. [https://svn.kwarc.info/repos/supervision/intern/2008/giceva\\\_jana/project/internship\%20report.pdf](https://svn.kwarc.info/repos/supervision/intern/2008/giceva\_jana/project/internship\%20report.pdf).
- [GM08] G. Goguadze and E. Melis. Feedback in ActiveMath exercises. In *International Conference on Mathematics Education (ICME)*, 2008.
- [JOB09] <https://jomdoc.omdoc.org/wiki/JOBAD>, 2009.
- [KAJ<sup>+</sup>08] M. Kohlhase, ř. Anca, C. Jucovschi, A. González Palomo, and I. A. řucan. MathWebSearch 0.4, a semantic search engine for mathematics. manuscript, <http://mathweb.org/projects/mws/pubs/mkm08.pdf>, 2008.
- [KLM<sup>+</sup>09] M. Kohlhase, C. Lange, C. Müller, N. Müller, and F. Rabe. Notations for active documents. KWARC Report 2009-1, Jacobs University, 2009. Available from: [http://kwarc.info/publications/papers/KLMMR\\_NfAD.pdf](http://kwarc.info/publications/papers/KLMMR_NfAD.pdf).

- [KMM07] M. Kohlhase, C. Müller, and N. Müller. Documents with flexible notation contexts as interfaces to mathematical knowledge. In P. Libbrecht, editor, *Mathematical User Interfaces Workshop*, 2007.
- [KMR08] M. Kohlhase, C. Müller, and F. Rabe. Notations for Living Mathematical Documents. In Autexier et al. [ACR<sup>+</sup>08].
- [Koh06] M. Kohlhase. OMDOC – *An open markup format for mathematical documents [Version 1.2]*. Number 4180 in LNAI. Springer, 2006.
- [Koh08] M. Kohlhase. Using L<sup>A</sup>T<sub>E</sub>X as a semantic markup format. *Mathematics in Computer Science*, 2008.
- [Lan08] C. Lange. SWiM – a semantic wiki for mathematical knowledge management. In S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis, editors, *ESWC*, volume 5021 of *Lecture Notes in Computer Science*. Springer, 2008.
- [Lan09] C. Lange. OpenMath wiki. <http://wiki.openmath.org>, 2009.
- [LGP08] C. Lange and A. González Palomo. Easily editing and browsing complex OpenMath markup with SWiM. In P. Libbrecht, editor, *Mathematical User Interfaces Workshop*, 2008. Available from: <http://www.activemath.org/~paul/MathUI08>.
- [MGH<sup>+</sup>06] E. Melis, G. Goguadze, M. Homik, P. Libbrecht, C. Ullrich, and S. Winterstein. Semantic-aware components and services of activemath. *British Journal of Educational Technology*, 37(3), 2006.
- [Mon05] MONET. <http://monet.nag.co.uk/mkm>, seen March2005.
- [Moz09] Mozilla Labs. Ubiquity. <http://ubiquity.mozilla.com>, 2009.
- [O'R05] T. O'Reilly. What is Web 2.0. <http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html>, 2005.
- [pan] The panta rhei Project. <http://trac.kwarc.info/panta-rhei>. seen March 2009.
- [RK08] F. Rabe and M. Kohlhase. An exchange format for modular knowledge. In P. Rudnicki and G. Sutcliffe, editors, *Knowledge Exchange: Automated Provers and Proof Assistants (KEAPPA)*, November 2008.
- [SCI09] The SCIENCE project – symbolic computation infrastructure for europe [online]. 2009.
- [SD08] J. Stratford and J. H. Davenport. Unit knowledge management. In Autexier et al. [ACR<sup>+</sup>08].
- [Str08] J. Stratford. Creating an extensible unit converter using openmath as the representation of the semantics of the units. Technical Report 2008-02, University of Bath, 2008. <http://www.cs.bath.ac.uk/pubdb/download.php?resID=290>.
- [TNT09] TNTBase Project. <https://trac.mathweb.org/tntbase/>, 2009.
- [Ull08] C. Ullrich. *Pedagogically Founded Courseware Generation for Web-Based Learning*. LNCS. Springer, 2008.
- [W3C] W3C. Mathematical Markup Language (MathML) 3.0 (Third Edition).
- [Yah09] Yahoo! Pipes [online]. 2009.

# A [insert XML Format] Database for [insert cool application]

Vyacheslav Zholudev, Michael Kohlhase, Florian Rabe  
Computer Science  
Jacobs University Bremen, Germany

January 29, 2010

## Abstract

TNTBase is a versioned XML database; it combines XML fragment access techniques like XQuery with file system functionality and versioning features a la Subversion. We present an infrastructure how the generic TNTBase system can be specialized to include document format-specific services, such as validation, format-specific virtual files and human-oriented presentation.

## 1 Introduction and Related Work

In [ZK09] we have presented the TNTBase system, a versioned XML database, which combines XML fragment access techniques like XQuery [XQua, XQUB] with the file system functionality and versioning features of Subversion [PCSF08]. This system is intended as a generic storage solution for web applications based on collections of XML documents. However, most such applications are tailored to specific features of their respective document formats. For instance, DocBook [WM08] manuals rely on format-specific style sheets for web presentation and printing, which only work if all documents are valid instances of the DocBook format. The situation is similar for other applications; some even combine multiple document formats and thus need to support multiple validation schemata and presentation pipelines. Furthermore, even homogeneous collections may contain documents in various versions of the underlying formats: indeed the versioning capabilities of the TNTBase system should enable the management of long tails of legacy materials for which format conversion is not cost-effective.

In our experiments with the TNTBase system, it has turned out that many of the format-specific functionalities of the web application layer can be performed by the TNTBase system if we add format-specific interfaces for validation, aggregation, and presentation. We will call this layer  $\text{TNTBase}(\mathcal{F})$ , since the services are parametric in the document format  $\mathcal{F}$ . This extension leads to a refined information architecture of TNTBase-supported web

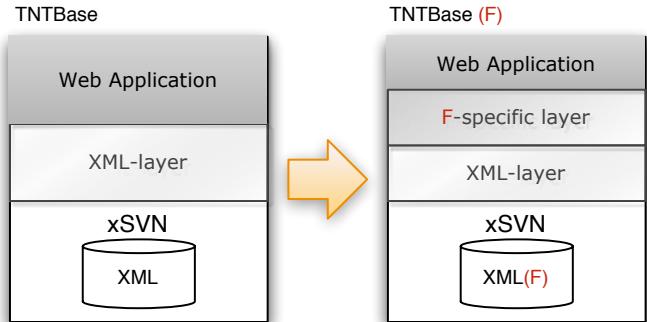


Figure 1:  $\text{TNTBase}(\mathcal{F})$ : A Basis for Web Applications applications; see Figure 1, where the generic  $\text{TNTBase}(\mathcal{F})$  layer takes over format-specific functionalities and thus decreases the effort for developing web applications.

In this paper we present the  $\text{TNTBase}(\mathcal{F})$  infrastructure and its APIs for validation (Section 3), format-specific virtual files (Section 4), and human-oriented presentation (Section 5). We will present much of the functionality of  $\text{TNTBase}(\mathcal{F})$  by using our OMDoc format [Koh06] ( $F = \text{OMDoc}$ ) as a running example, since this drives the particular development and integrates many

of the structural prerequisites for virtual documents. The work presented in this paper is based on the experience of managing a collection of more than 2000 OMDoc documents (ranging from formal representations of logics to course materials in a first-year computer science course) in TNTBase(*OMDoc*).

This paper is a short version of [ZKR], which should be consulted for details we had to omit for space reasons.

## 2 Document-Format-Specific Features for TNTBase

To make this paper self-contained we briefly recap TNTBase (see [ZK09] [TNT] for details). Then we take a closer look at the document-specific workflows in document-collection-centered applications that can be generically supported by a versioned XML storage system like TNTBase. This analysis serves as a motivation and basis for the implemented TNTBase( $\mathcal{F}$ ) layer, which we will present in detail in the following sections.

### 2.1 TNTBase State of the Art

The core of TNTBase consists of the xSVN module, which integrates Berkeley DB XML [Ber] into a Subversion server. DB XML stores HEAD revisions of XML files; non-XML content like PDF, images or L<sup>A</sup>T<sub>E</sub>X source files, differences between revisions, directory entry lists and other repository information are retained in a usual SVN back-end storage. Keeping XML documents in DB XML allows us to access those files not only via any SVN client, but also through the DB XML API that supports efficient querying of XML content via XQuery and modifying that content via XQuery Update. As many XML-native databases, DB XML also supports indexing, which improves performance of certain queries.

The TNTBase system is realized as a web-application that provides two different interfaces to communicate with: an xSVN interface and a RESTful interface for XML-related tasks. The xSVN interface behaves like the normal SVN interface — the mod\_dav\_svn Apache module serves requests from remote clients — with one exception: If one of the committed XML files is ill-formed, then xSVN will abort the commit transaction. The RESTful interface provides XML fragment access to the versioned collection of documents:

**Querying:** xSVN extends DB XML XQuery by a notion of file system path to address path-based collections of documents. For instance, a part of a query `collection("//papers*/*.xsl")` will address all XSLT stylesheets in the child folders of directories having the `papers` prefix.

**Modifying:** It is also possible to modify XML documents via XQuery Update. In contrast to pure DB XML, modified documents are versioned, i.e., a new revision is committed to xSVN.

**Querying of previous revisions:** Although xSVN's DB XML back-end by default holds only HEAD revisions of XML documents, it is also possible to access and query previous versions by providing a revision number if they have been *cached* (by user request). Previous versions cannot be modified since once a revision is committed to an xSVN repository, it becomes persistent.

**Virtual Files:** These are essentially “XML database views” analogous to views in relational databases; these are tables that are virtual in the sense that they are the results of SQL queries computed on demand from the explicitly represented database tables. Similarly, TNTBase virtual files (VF) are the results of XQueries computed on demand from the XML files explicitly represented in TNTBase, presented to the user as entities (files) in the TNTBase file system API. Like views in relational databases TNTBase virtual files are editable, and the TNTBase system transparently patches the differences into the original files in its underlying versioning system. Again, like relational database views, virtual files become very useful abstractions in the interaction with versioned XML storage.

## 2.2 Validation and Interface Extraction

One of the salient features of XML as a representation format are its well-established methods for document validation. Indeed most document management workflows take advantage of this and check grammatical constraints on documents by schema validation to simplify further document processing.  $\text{TNTBase}(\mathcal{F})$  supports this practice by validating documents upon commit by default. However, sometimes schema validity is too strong a restriction in practice (e.g., during document authoring or format migration), so the level of validity is configurable. Moreover, many document format constraints — e.g., inter-document link consistency or the absence of link cycles — cannot be checked even by modern schema languages; therefore  $\text{TNTBase}(\mathcal{F})$  provides an API for custom validation modules that can verify that high-level document (collection) integrity constraints are met, if later processing steps require them.

Both high-level validation and further document processing often rely on specific information about other parts of the document collection. To support these processes incrementally,  $\text{TNTBase}(\mathcal{F})$  supports the extraction and indexing of such information upon commit. This approach is analogous to the “separate compilation” paradigm in programming, where declared methods and their types are extracted at compile-time into “signature” files that are used when compiling other files. A prominent example in the XML arena is the extraction of RDF triples for the use in query engines from RDFa-annotated documents. As the specific information is format and application specific, we speak of *interface extraction* for the purposes of this paper.  $\text{TNTBase}(\mathcal{F})$  enables the user to configure interface extraction at commit time; in our experience, this is a crucial ability to make high-level validation tractable. For details of the realization we refer to Section 3.

## 2.3 Compilation, Browsing and Cross-Referencing

Eventually, the purpose of most document collection applications is to enable humans and machines access to (processed forms) of the document collection. Technically, this usually involves the transformation into specialized presentation-oriented formats for machine (e.g., programming languages) or human (e.g., PDF generation) consumption. Depending on locality properties of the transformation and the rate of change of the document collection, it can be more efficient to execute the transformations when a document is committed to the collection (then we can employ a process analogous to interface extraction) or when the document is requested. Both workflows need to be supported in  $\text{TNTBase}(\mathcal{F})$ , and in both cases these documents are automatically served in addition to their sources. If a post-processing of documents for human consumption is defined,  $\text{TNTBase}$  can provide an enhanced browsing interface. In the future, the  $\text{TNTBase}$  architecture will make it possible to provide higher-level presentational services such as navigation bar, cross-referencing, and in-place expansion of links, if it is told, which language features constitute document fragments and links to documents or document fragments (see Section 4.2). For details of the realization in  $\text{TNTBase}(\mathcal{F})$  we refer to Section 5.

## 2.4 Virtual Documents

$\text{TNTBase}$  virtual files as introduced above have one great disadvantage: They are not instances of one of the XML formats in the application. For instance the example of a virtual file of section headings of the SVN Book [PCSF08] used in [ZK09] groups the `title` elements in special  $\text{TNTBase}$  elements, giving a mixed-namespace format specific to the  $\text{TNTBase}$  system. One generally wants to have virtual files that are “virtual documents”, i.e. valid instances of a given format  $\mathcal{F}$ . This makes *virtual documents* into first-class citizens at the web application level. In theory, if a document format permits to be split into fragments, it becomes possible to recombine the same content fragments into multiple (virtual) documents. If virtual documents make their composition explicit, edited versions can be decomposed into edited fragments, and the changes can be propagated to the original document and other virtual documents using said fragments. But achieving this is surprisingly difficult; both in theory and in practice. In Sections 4.2 and 4.3

we discuss the properties the target format  $\mathcal{F}$  must have to allow this, and the  $\text{TNTBase}(\mathcal{F})$  implementation.

## 2.5 Realizing $\text{TNTBase}(\mathcal{F})$

As described in [ZK09],  $\text{TNTBase}$  has two major components: xSVN and Java-based Web-application that is built on top of the Java library for accessing xSVN’s DB XML directly. Most of the material described in this paper involves both parts and interactions between them. For instance, when one commits a new version of some XML documents and wants their presentation to be cached, this involves committing files in xSVN, figuring out what files the presentation should be generated for, sending the corresponding request to the Java part of  $\text{TNTBase}$ , generating presentation and finally saving it into DB XML. We omit the technical details about the interactions, since they mainly concern the (painful) integration of C++ and Java. For our initial implementation, we utilize the standard SVN hook mechanisms for pre-commit and post-commit processing. In a nutshell, we use a pre-commit or post commit hook (depending on the processing purpose, e.g. validation or generation presentation) for figuring out what subset of committed files is subject to further processing.

Once this is done, a script sends requests to the  $\text{TNTBase}$  RESTful interface, where the actual processing is executed. The return codes and error stream are used to notify the committer about the result (e.g., in case of validation errors). For an example see Figure 2. This mechanism is surprisingly flexible and naturally fits into the xSVN approach since it is inherited from SVN. In the future, a tighter integration will add additional scalability and manageability, but our approach shows that that the workflows described in this paper naturally work inside the  $\text{TNTBase}$  architecture.

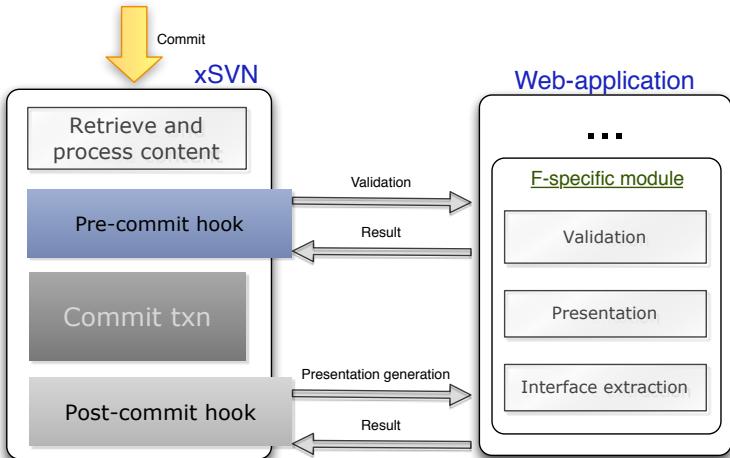


Figure 2: Interactions between xSVN and Web-application  
Figure 2 illustrates the interaction between the xSVN component and the Web-application. The xSVN component contains a 'Pre-commit hook' (highlighted in blue) which interacts with the 'F-specific module' of the Web-application. The 'F-specific module' contains 'Validation', 'Presentation', and 'Interface extraction' steps. The 'Pre-commit hook' sends 'Validation' and 'Result' to the 'F-specific module', and receives 'Presentation generation' and 'Result' from it.

## 3 Validation and Interface Extraction

$\text{TNTBase}$  provides a powerful schema and format-specific validation infrastructure. When committing a set of changes to  $\text{TNTBase}$ , a validation method is selected for every affected file, that validation is performed, and the xSVN transaction is rejected if validation fails. That ensures that only well-formed documents are stored in  $\text{TNTBase}$ . The validation method is selected per file via the `tntbase:validate` property (and thus can easily be switched off temporarily). Successful validation may also return extracted information about the committed file and its dependencies, e.g., a document format-specific index, to be stored by  $\text{TNTBase}$  for later querying.

An important property of  $\text{TNTBase}$  is that committing a change to a `tntbase:validate` property will result in automatic revalidation of all affected files. Thus,  $\text{TNTBase}$  guarantees that files with a certain `tntbase:validate` property are well-formed.

**Selecting a Validation Method** Every directory or file may have the `tntbase:validate` property whose value is a string. If the property is absent, it is assumed to be empty. For

a file its value is a string that defines the validation methods to be applied. For a folder the `tntbase:validate` property is a whitespace-separated list of strings.

If this list is of the form  $e_1, m_1, \dots, e_n, m_n$ , the every  $e_i$  is treated as a file name extension and every  $m_i$  as a validation method. The semantics is that files with the extension  $e_i$  are to be validated using method  $m_i$ . If the length of the list is odd, the last entry is treated as a default validation method that applies to any file with an extension that was not met before in the list.

When validating a file, the entry in its `tntbase:validate` property determines the validation method (or methods). If there is no matching entry, one is searched in the corresponding property of the parent directory, and so on until a matching entry is found. If the root directory is reached without finding an entry, the predefined validation method `none` is chosen, which does nothing and always succeeds.

**Defining Validation Methods** Every TNTBase repository comes with a special top-level directory called `admin`. This folder contains all configuration files for that repository so that configurations can be easily made via SVN (and are automatically versioned themselves!). The configuration of validation behavior is done in the `process` subfolder. It may contain a file `methods.xml`, which defines validation methods. If it is not present, no methods are defined.

The content of this file is of the form

---

1 <methods> $M_1 \dots M_n$ </methods>

---

where every  $M_i$  is of one of the following forms:

---

```
<schema name=" $n$ " type=" $t$ " location=" $l$ " />
<java name=" $n$ " class=" $c$ " />
```

---

In the first case, validation is performed by schema validation.  $l$  is a URL giving the location of the schema, and  $t$  is its type: Currently, a user can choose between the `rnc` and `rng` for the text and XML syntaxes of RelaxNG schemata correspondingly. Other XML schema language are planned to be added in the future, e.g. Schematron [sch]. Also, DTD and XML Schema validation is supported automatically by DB XML, and the schemas should be provided as XML files themselves.  $n$  is the name of the validation method that occurs in the `tntbase:validate` property.

While the first case, already permits simple validation through different XML schemata, the second case offers full customizability. Here,  $n$  is as for schema validation, and  $c$  gives the qualified name of the Java class implementing the validation interface. (See the project website <http://tntbase.mathweb.org> for documentation of the interface.) These implementation classes must be in the Java `classpath` provided when starting TNTBase.

**Applying Validation and Interface Extraction** Every  $m_i$  in the `tntbase:validate` property is of the form  $s_i?(+v_i)?$ , where  $s_i$  is the name of an XML schema validation method and  $v_i$  is the name of a Java validation method. The latter may generate any serializable content. The intuition here is that the validation automatically produces information that is desirable to store for later use. These can be transformations of the committed file into human- or machine-readable formats such as XHTML or RDF.

These results are stored as XML documents in TNTBase and every such document is associated with a real XML file in TNTBase. They are not shown in the TNTBase file system and can be queried via a separate TNTBase interface. For that purpose, the `method` element from above receives one additional optional attribute `output`. If the value of this attribute is `true` TNTBase will cache the files containing the extracted information and will make them accessible via dedicated RESTful methods.

In particular, the output of validating the file `PATH` is available at <http://tntbase.your.host.org/restful/integration/output/METHOD/PATH>, and XQuery access to all output files is possible via [http://tntbase.your.host.org/restful/integration/query/METHOD?query=XQUERY\\_EXPRESSION](http://tntbase.your.host.org/restful/integration/query/METHOD?query=XQUERY_EXPRESSION). In both cases, `METHOD` is the name of the validation method.

As an example, we give the current setup of the OMDoc repository used in the Latin project [LAT]. Here the top-level folder of the repository carries the property `tntbase:validate=omdoc mmt-rng+mmt`. This means that all files with the `omdoc` extension are to be validated using first the `mmt-rng` and then the `mmt` method. A subfolder `inprogress` contains work in progress and carries the property `tntbase:validate=None` to avoid validation.

The `methods.xml` file looks as follows:

---

```

<methods>
  <schema name="mmt-rng" type="rnc" location="/var/www/tntbase/schemata/mmt.rnc" type="rnc"/>
  3 <java name="mmt" class="info.kwarc.jomdoc.frontend.TNTValidate" output="true"/>
</methods>
```

---

Thus, the method `mmt-rng+mmt` consists of calling first the schema validation using the schema `mmt.rnc`, followed by a custom java implementation in the class `TNTValidate`.

The extraction method of the class `TNTValidate` returns a set of RDF triples that represent the `omdoc` file according to the `omdoc` ontology. Because the value of the `output` attribute is `true`, the generated RDF theory (ABox) is accessible via the RESTful TNTBase interface. For example, the RDF triples of the file `math/algebra/algebra1.omdoc` are accessible as <http://tntbase.your.host.org/restful/integration/output/mmt/math/algebra/algebra1.omdoc>.

## 4 Virtual Documents

In this section, we will take a closer look at virtual documents: As they are surprisingly complex to understand, we introduce the concept and its realization by an example in Section [4.1] before we develop a precise terminology to capture all aspects of the concept: Sections [4.2] and [4.3] develop a set of prerequisites for the document format to make use of virtual documents and show how they can be achieved generically.

### 4.1 Virtual Documents by Example

To fortify our intuition about virtual documents let us consider the following situation: We have a set of exercises with problem statements and solutions in our document collection. Say they are marked up in the following way:

Listing 1: An Exercise with Solution

---

```

1 <exercise>
  <problem>P</problem>
  <solution>S</solution>
</exercise>
```

---

Now we want to make them available in two forms: without solutions to students and with (master) solutions to the teaching assistants. In this situation, virtual documents are ideal. Generally, for a virtual document we specify an XML file like the one in Listing [2]. It consists of a skeleton and a list of external queries.

Listing 2: A Virtual Document for Practice Exercises

---

```

1 <tnt:virtualdocument xmlns:tnt="http://tntbase.mathweb.org/ns">
  <tnt:skeleton xml:id="exercises">
    <omdoc xmlns="http://omdoc.org/ns" xmlns:dc="http://purl.org/DublinCore">
      <dc:title>Exercises for GenCS</dc:title>
      <dc:creator>Michael Kohlhase</dc:creator>
    6 <omdoc>
      <dc:title>Acknowledgements</dc:title>
      <omtext>The following individuals have contributed material to this document</omtext>
      <tnt:xqinclude query="distinct-values(collection(/gencs/*/*.omdoc)//dc:author)">
        <tnt:return><omtext><tnt:result/></omtext></tnt:return>
      11 </tnt:xqinclude>
    </omdoc>
```

---

```

<omdoc>
  <dc:title>SML Exercises</dc:title>
  <tnt:xqinclude>
    <tnt:query name="xq.SMLex"/>
    <tnt:return><exercise><tnt:result/></exercise></tnt:return>
  </tnt:xqinclude>
</omdoc>
<omdoc> ...</omdoc>
</omdoc>
</tnt:skeleton>
<tnt:query name="xq.SMLex">
  for $i in collection(/gencs/SML/*.omdoc)//exercise
  return $i/*[local-name() ne 'solution']
</tnt:query>
...
</tnt:virtualdocument>

```

---

Conceptually, the skeleton consists of the top-level parts of the intended exercises document, where some document fragments have been replaced by embedded XQueries that generate them. In general, queries have the form

Listing 3: A XQuery Fragment Reference

```

<tnt:xqinclude>
  <tnt:query>q</tnt:query>
  <tnt:return>R</tnt:result>
</tnt:xqinclude>

```

---

where  $q$  is an XQuery and  $R$  consists of a result expression where the `tnt:result` element will be replaced with the results of  $q$ . In line 7-9 we have such a query in a syntactic variant where `<tnt:xqinclude query="q">R</tnt:xqinclude>` was used to abbreviate the primary query form above for queries that do not contain embedded elements. The result of this particular query would be a list of author names wrapped in an `omtext` element. The query in lines 15-18 uses another useful syntactic feature: it refers to the query by reference to enable reuse and sharing for virtual documents (see below). <sup>1</sup> The result of this query would be a list of exercises of the form

```

<exercise>
  <problem>P</problem>
</exercise>

```

---

Note that the default and the `dc` namespace within the query are inherited from the dominating `omdoc` node.

If we want to edit a virtual document, then we should tell TNTBase to send it to us in a special editing mode. Then the relevant parts of our virtual document will be of the form

```

<exercise>
  <problem tnt:srcfile="/gencs/SML/prob1.omdoc" tnt>xpath="(omdoc/exercise/problem)[2]">P</problem>
</exercise>

```

---

Note that the result fragments have been decorated with `tnt:srcfile` and `tnt>xpath` attributes that specify the origin of the text fragments. These attributes are syntactical variants of the source references in virtual files (see [ZK09]) and enable editability and transparent commit in the same way.

Thus, the virtual document in Listing 2 indeed expands to the desired exercises document after XQuery processing. In  $\text{TNTBase}(\mathcal{F})$ , virtual documents are created simply by committing a *virtual document specification* (the XML file in Listing 2) to the xSVN repository and executing an additional method of TNTBase that takes as an input the path of a virtual document specification and a path of a new virtual document itself. Thus, we can create multiple virtual documents based on a single virtual document specification. For instance, the above virtual document specification's

<sup>1</sup>In the future, we intend to integrate the XQuery module system into virtual documents so that individual XQuery function and variable declarations can be shared between queries.

XQueries can be changed so that they select problems in the directory where virtual document resides. This is done by for \$i in collection(./\*.omdoc)//exercise return \$i/\*[local-name() ne 'solution'], where the first “.” in the query stands for the *current directory*. Thus we write a specification once, and may use it for creating virtual documents without solutions in different folders separated by topic.

The reference-based setup caters for a wide variety of reuse scenarios. For instance the document for the GenCS teaching assistants could be specified by the following virtual document specification, which reuses the skeleton from Listing 2

---

```

<tnt:virtualfile xmlns:tnt="http://tntbase.mathweb.org/ns">
  <tnt:skeleton href="../exercises.vf#exercises"/>
  <tnt:query name="xq.SMLex">
    for $i in collection(/gencs/SML/*.omdoc)//exercise return $i
  </tnt:query>
  ...
</tnt:virtualfile>
```

---

Note that the new query does not remove the solutions. It is lexically captured by the named reference in the skeleton. One may think of method overloading in Java or C++. It is also possible to have an empty tnt:query element in the specification. Then it becomes an *abstract* specification, and can be compared to abstract methods in Java or pure virtual functions in C++.

## 4.2 Document Formats and Document Fragments

In the example above we have been very informal; to fully understand the concept of virtual documents we have to work a little harder and pin down the relevant concepts. We will provide the relevant definitions and explain them by examples that show the problems we have glossed over above.

**Definition 1** (Document Format). A *document format* consists of

1. a formal language, whose words we call *documents*,
2. a formal language of *fragments* over the same alphabet,
3. for every document  $D$  a set of pairs  $(p, F)$ , where  $F$  is a fragment occurring as a sub-word of  $D$  and  $p$  is some expression, called the *position* of  $F$  in  $D$ .

We write  $D(p) = F$  to retrieve the fragment at position  $p$ .

The intuition of fragments is that fragments occur as components of documents. Document formats are typically given as a context-free grammar that subsumes the language with some non-terminal symbols designated as the fragment-producing ones. For example, we define the format XML/XPath as follows: The documents are the XML documents, and the fragments of an XML document  $D$  are the element nodes occurring in  $D$ ; the fragment positions are given by XPath expressions [BBC<sup>+</sup>07]. Another XML-based document format is XML/id: Here, only the nodes with `xml:id` attributes [MVW05] are fragments, they are addressed by the values of `xml:id` attributes which serve as positions. For text documents, we can use substrings as fragments together with line and column numbers as positions (or paragraphs and paragraph numbers if we use empty lines as paragraph separators). Furthermore, any formal language trivially becomes a document format if every document has itself as the only fragment.

Any particular XML language becomes a document format in the sense of Definition 1 by treating it as a special case of XML/XPath. However, it is often useful to define fragments differently, in particular to restrict the fragments to ones that carry self-contained meaning in the document format. After all, documents are composed of fragments that represent logical structuring units.

The OMDoc1.6<sup>2</sup> format has been designed around the notion of “semantic fragment”. These are represented by tags such as *omdoc* (used both for documents and document sections), *theory*,

---

<sup>2</sup>The OMDoc1.6 format is the radically redesigned successor of OMDoc1.2 [Koh06] which is the first draft of the OMDoc2 format. It is based on the semantic data and referencing model of [RK09], which makes it an interesting case study for our purposes here

*symbol, axiom, notation.* Other element nodes, such as the ones representing individual mathematical expressions, are not considered fragments. In the OMDoc1.6 design the fragment nodes all have `name` attributes: Any element with a name yields a fragment, and any tag that is worthy of being extracted as a semantic fragment gets a name. Contrary to `xml:ids`, the `name` attributes have to be unique only within the scope given by their parent fragment, which gives rise to a hierarchical naming scheme that is more adequate for mathematical practice. For example, the relative URI `algebra.omdoc/groupoids?Monoid?comp`<sup>3</sup> refers to the declaration of the composition operation in the theory of monoids occurring in the section on groupoids in the document `algebra1.omdoc`. The general form of an OMDoc fragment position is  $sec_1/\dots/sec_m?mod_1/\dots/mod_m?sym$ , where the  $sec_i$  are section names, the  $mod_i$  are the names of nested modules in the OMDoc module system [RK09], and  $sym$  is a symbol name. Clearly, the set of fragments of an OMDoc document can be naturally included into the set of fragments of the same document considered as an XML/XPath document.

We use names instead of `xml:ids` as fragment positions because the latter have to be document-unique, whereas in virtual documents, we want to be able to flexibly recombine existing fragments into new documents. Already the next definition could lead to name clashes if we used `xml:ids` to identify fragments.

**Definition 2.** We say that a document format  $\mathcal{F}$  supports *reference expansion* if it provides

1. for every document  $D$  in  $\mathcal{F}$ , a distinguished set of fragments called *fragment references*,
2. for every fragment reference  $R$  in  $\mathcal{F}$ , a list  $(d_i, p_i)_{i=1}^n$ , called *results*, where  $d_i$  is a document URI and  $p_i$  is a fragment position in the document referenced by  $d_i$ ,
3. an operation  $exp(d, p)$  that given a URI  $d$  of an  $\mathcal{F}$  document and a position  $p$  in it returns a document fragment in  $\mathcal{F}$ ,

such that  $D$  is semantically equivalent to the document that arises from replacing a fragment reference  $R$  with the concatenation  $exp(d_i, p_i)_{i=1}^n$ . A document is called *fully expanded* if it contains no fragment references.

Technically, the precise definition of semantic equivalence must be given by the document format as an equivalence relation between documents. The right choice of the semantic equivalence relation can be quite difficult, and often different notions of equivalence must be employed in different contexts. For example, replacing relative URIs in an XHTML document with appropriately resolved absolute ones will usually not change the semantics of the document. However, in certain situations, it is still desirable to forbid such a transformation, e.g., when a directory of inter-linked documents is often moved to different locations. For simplicity, we will not go into details and simply assume such an equivalence relation to be present.

The intuition behind reference expansion becomes clear if we define it for the document format XML/XPath from above: It becomes a document format XML/XPath/XInclude with reference expansion by using XML inclusions via XInclude [MOV06], i.e., the fragment references are the `xi:include` elements. XInclude (essentially) requires that processors obtain the document fragment(s) referenced by the `href` and `xpointer` attributes of the `xi:include` element and replace the `xi:include` element with them, which we take as semantic equivalence. Thus, we can define that for every `xi:include` element, the result list is  $(d, p_1), \dots, (d, p_n)$  where  $d$  is given by the `href` attribute,  $p_1, \dots, p_n$  are the normalized XPath expressions identifying the nodes matched by the `xpointer` attribute.

At first it might seem that one should always put  $exp(d, p) := D'(p)$  where  $D'$  is the document at URI  $d$ . However, it is useful to permit the document format to perform an arbitrary operation instead – typically this operation consists of computing  $D'(p)$  and then applying some post-processing to it. For example, `xi:include` computes  $exp(d, p)$  by adding an attribute `xml:base` to  $D'(p)$  because  $D'(p)$  might inherit its `xml:base` from an ancestor node within  $D'$ . Similarly, XML namespace attributes inherited from an ancestor may have to be added to  $D'(p)$ .<sup>4</sup>

---

<sup>3</sup>The use of an additional `?` in the query is unusual but legal. It leads to a very compact notation, which is important in OMDoc.

<sup>4</sup>The latter is usually not necessary for XInclude, which technically acts on info sets rather than documents.

OMDoc1.6 integrates a custom syntax for reference expansion. Every tag designating a fragment may alternatively occur as an empty element node with an `href` attribute instead of a `name` attribute. The value of this attribute is of the form  $d/p$  where  $d$  is a URI without query or fragment and  $p$  is an OMDoc fragment position. The list of results always has length 1 and is given by  $(d, p)$ . (Note that since both  $d$  and  $p$  may contain slashes, the computation of  $(d, p)$  has to be carried out by a server providing the resource  $d$ ). To define reference expansion, let  $D'$  be the document at  $d$ . Then  $\exp(d, p)$  returns  $D'(p)$  with an added attribute `base` whose value references the parent fragment of  $D'(p)$  in  $D'$ . For example, assume the following OMDoc document  $D'$  at URI  $d$ :

---

```
<omdoc>
  <omdoc name="div_1">
    <theory name="th_1"><constant name="c_1"/></theory>
  </omdoc>
</omdoc>
```

---

A document  $D$  may contain a fragment reference to the theory `th_1` in the section `div_1` of the form `<omdoc href="d/div_1?th_1"/>`. Then  $\exp(d, \text{div}_1?\text{th}_1)$  yields

---

```
<theory name="th_1" base="d/div_1"><constant name="c_1"/></theory>
```

---

The added `base` attribute serves two purposes. Firstly, it functions as a base address relative to which all semantic references occurring inside `th_1` are resolved. (There are none in this example, but practical theories contain large numbers of mathematical expressions containing semantic references to theories and constants.) Secondly, it remembers the origin of the theory `th_1` so that a link between the theory defined in  $D'$  and its copy in the fully expanded version of  $D$  is maintained. This link is important for several applications, and below we will use it in particular to propagate changes made to `th_1` from  $D$  to  $D'$ .

**Definition 3** (Document Contraction). In a document format that supports reference expansion, we speak of *document contraction* if a document  $D$  is transformed into a semantically equivalent document  $D'$  by replacing a fragment of  $D$  with a fragment reference.

A fragment that is not simply a fragment reference is called *fully contracted* if no further contraction is possible. A fully contracted fragment is called *atomic* if it is also fully expanded.  $S$  is called the *skeleton* of a document  $D$  if  $S$  arises from  $D$  by contracting exactly the atomic fragments. Finally, we say that a document format supports document contraction if every document has a skeleton.

Both XML/XPath/XInclude and OMDoc support document contraction. However, for XML/XPath/XInclude, the atomic fragments are the empty elements, and thus the skeleton is not interesting because it is isomorphic to the original XML document. But for document formats with custom definitions of fragments such as OMDoc, the skeleton yields an interesting decomposition into medium-sized chunks.

A good example of a skeleton is the table of contents with every node representing one entry. This can also serve as a guiding intuition to define document formats: An XML node should count as a fragment if it could occur in a table of contents. This motivated our choice of fragments for a document format for DocBook: The skeleton of a DocBook document is just its table of contents. More precisely, we can say the following.

**Theorem 1.** *Let  $\mathcal{F}$  be a document format arising from XML/XPath/XInclude by restricting the documents to a sub-language of XML and by restricting the fragments of a document to element nodes with certain tags. Further assume that the  $\mathcal{F}$ -documents may contain `xi:include` fragments and that the specification of  $\mathcal{F}$  imposes semantic equivalence under reference expansion as defined above for XML/XPath/XInclude. Then  $\mathcal{F}$  supports reference expansion and document contraction.*

As an example, consider the left OMDoc document in Figure 3 with URI  $d$ . It can be contracted to the one on the right: Here all fragment references are relative to the base URI  $d$ , which receives a trailing slash so that, e.g., `div_1/div_1?theory_1` resolves to `d/div_1/div_1?theory_1`. Note how the

|  |  |
|--|--|
| <pre> &lt;omdoc base="d/"&gt;   &lt;omdoc name="div_1"&gt;     &lt;omdoc name="div_1"&gt;       &lt;theory name="th_1"&gt;         &lt;constant name="c_1"&gt;           &lt;type&gt;T&lt;/type&gt;           &lt;definition&gt;D&lt;/definition&gt;         &lt;/constant&gt;       &lt;/theory&gt;     &lt;/omdoc&gt;     &lt;omdoc name="div_2"&gt;       &lt;theory name="th_1"&gt;         ...       &lt;/theory&gt;       &lt;theory name="th_2"&gt;         ...       &lt;/theory&gt;     &lt;/omdoc&gt;   &lt;/omdoc&gt; &lt;/omdoc&gt; </pre> | <pre> &lt;omdoc base="d/"&gt;   &lt;omdoc name="div_1"&gt;     &lt;omdoc name="div_1"&gt;       &lt;theory name="th_1"&gt;         &lt;constant           href="div_1/div_1?th_1?c_1"/&gt;       &lt;/theory&gt;     &lt;/omdoc&gt;     &lt;omdoc name="div_2"&gt;       &lt;theory href="div_1/div_2?th_1"/&gt;       &lt;theory href="div_1/div_2?th_2"/&gt;     &lt;/omdoc&gt;   &lt;/omdoc&gt; &lt;/omdoc&gt; </pre> |
|--|--|

Figure 3: Contracting Documents

contraction status is different between the fragments at position  $\text{div\_1}/\text{div\_1}$  and  $\text{div\_1}/\text{div\_2}$ : The former is skeletal, the latter is fully contracted. Such an OMDoc document typically arises after a fully contracted document has been served and the user has interactively expanded some of the fragment references.

### 4.3 Virtual Documents

Contraction permits to decompose a document into its fragments, and by expanding fragment references this process can be inverted. This is useful in itself, but it also opens the door to a powerful application: New documents can be composed partially or completely out of fragments from existing documents. If we follow this idea systematically, we can conceptually consider all documents to be skeletons that pick fragments from a shared reservoir. In particular, there is no conceptual distinction needed between the document in which a fragment is created and the documents, from which that fragments is referenced.

Virtual documents are simple as long as they are created or read but not changed. But assume a document  $D'$  that contains a reference to a fragment  $F$  of  $D$ , and assume that the reference has been expanded. Further assume, we changed  $F$  to  $F'$  within this expanded version of  $D'$ . When these changes are committed, we have two options:

- A The changes to  $F$  are propagated to  $D$ .  $D'$  is contracted to its original form, which means  $D'$  did not change at all.
- B  $D'$  is stored in its expanded form, which means that a new atomic fragment is stored for  $F'$ , and a new revision of  $D'$  is stored where the fragment reference now points to  $F'$ .

Clearly, both options are desirable under different circumstances. The following definition is strong enough to handle both cases.

**Definition 4** (Virtual Documents). *Assume a document format  $\mathcal{F}$  such that 1. all documents are XML documents, 2. all fragments are XML element nodes, 3. reference expansion and document contraction are supported. We say that the format supports virtual documents if it provides a partial function  $\text{contr}(F')$  that takes a document fragment  $F'$  and (if defined) returns  $(d, p, F)$  such that  $\text{contr}(\text{exp}(d, p)) = (d, p, D(p))$  where  $D$  is the document at URI  $d$ .*

The intuition is that  $\text{contr}(F') = (d, p, F)$  holds iff  $F'$  arose from a reference expansion where  $(d, p)$  is the URI-position pair used to identify the referenced fragment. In that case,  $F$  arises by inverting whatever modification  $\text{exp}(d, p)$  made to  $D(p)$ . Then  $F$  represents a new version of  $F'$  that can be propagated to the document at URI  $d$ .

To see more clearly how this works, we give a document format VXML for virtual XML documents. We define it as follows:

- The documents are XML documents.
- The fragments are the XML element nodes with XPath expressions as their positions.
- The fragment references are of the form of Listing 3, where  $Q$  is an XQuery returning a list  $(d_i, p_i)_{i=1}^n$  of results, and  $R$  is arbitrary XML in particular using an element node  $\langle \text{tnt:result} \rangle$ . The namespaces of the query  $Q$  are inherited from the  $\text{tnt:query}$  node.
- For every result, the node  $\exp(d_i, p_i)$  is defined as follows:
  1. Take the node list  $R$  and add to all top-level nodes an attribute  $\text{tnt:virtual} = "q"$  where  $q$  is the XPath expression of the  $\text{tnt:xqinclude}$  element. Let this yield  $L$ .
  2. Let  $N_i$  be the node list arising from replacing the node  $\langle \text{tnt:result} \rangle$  in  $L$  with: the node  $D_i(p_i)$  (including its in-scope namespace attributes) where  $D_i$  is the document at URI  $d_i$  with the following additional attributes:
    - an attribute  $\text{xml:base}$  as in the XInclude specification,
    - an attribute  $\text{tnt:added} = "\text{xml:base}"$  if an  $\text{xml:base}$  attribute was added,
    - the attributes  $\text{tnt:srcfile} = "d_i"$  and  $\text{tnt:xpath} = "p_i"$ .
  3. Return the concatenation  $N_1 \dots N_n$ .
- For every fragment  $F'$  the function  $\text{contr}(F')$  is defined whenever  $F'$  has exactly one descendant node  $N$  with an attribute of the form  $\text{tnt:origin} = "d \# \text{xpointer}(p)"$ ; and in that case it returns  $(d, p, F)$  where  $F$  arises from  $N$  by removing  $\text{tnt:origin}$  attribute and if applicable the  $\text{xml:base}$  attribute.

After retrieving VXML documents from TNTBase they can be dynamically expanded – e.g., by using an interactive browser or a TNTBase-supporting editor – in which case they will have  $\text{tnt:virtual} = "q"$  attributes. When committing the document, TNTBase (i) replaces all elements with such attributes with the original fragment reference, which can be retrieved from the database using the XPath expression  $q$ , and (ii) propagates the respective changes to the origin document using the  $\text{contr}(-)$  operation. This corresponds to option A above. The user can choose option B by removing the  $\text{tnt:virtual}$  attribute.

The above definition of VXML assumes that all results of the XQuery correspond to physical nodes in some documents rather than being constructed during the XQuery. This is a necessary prerequisite for the propagation of changes. In general, this may only hold for some result nodes, in which case only changes to those nodes can be propagated. TNTBase is able to handle that situation as well, but we avoid the description here for simplicity. Note that in our example above, the OMDoc format supports virtual documents because the `base` attribute mentioned in the definition of the OMDoc reference expansion determines the origin of a fragment.

## 5 Presentation: Compilation and Browsing

The presentation support of TNTBase is targeted at providing document format-specific browsing interfaces. We use the term *rendering* for the process of transforming documents into human-oriented presentation formats such as XHTML.

Rendering can be done at commit or at view time. Commit-time rendering is generally preferable because it makes viewing faster. However, it is not suitable for interactive editing of documents and for browsing virtual documents which are created on the fly using dynamically executed XQueries. Furthermore, the presentation of a document can be affected by changes in other documents, e.g., if a document contains fragment references that are expanded before rendering. Therefore, TNTBase supports both commit- and view-time rendering.

Both types of *rendering* are controlled by a configuration file `browsers.xml` in the folder `admin/viewing`. Its content is of the form

---

```
<browsers>B1 ... Bn</browsers>
```

---

where every  $B_i$  is of one of the following forms:

---

```
<xslt name="n" location="l"/>
<java name="n" class="c"/>
```

---

The semantics is very similar to the one of the corresponding entries of the `methods.xml` file. The main difference is that *n* is not referenced in a subversion property but directly in a URI.

For *view-time rendering*, if `PATH` is a path in a repository with the URL `REPOS`, then the URL `REPOS/restful/presentation/render/n/PATH` yields the version of that file rendered using the method *n*. In particular, `PATH` need not refer to a file but can also contain queries that generate virtual documents on the fly. Currently two types of rendering methods are supported: XSLT transformations and custom Java rendering. In the latter case, a user has to provide an implementation of certain TNTBase interfaces that return rendered documents.

For *commit-time rendering*, the situation is very similar. The main difference is that a user has to generate (via an ad-hoc TNTBase script) the post-commit hook that internally will invoke the necessary presentational caching method via the RESTful interface. When creating such a hook, the user has to provide the name of a browser, say *n*, that is declared in `browsers.xml`. Once this is set up and rendered documents have been generated at commit time, one uses the URL `REPOS/restful/presentation/view/n/PATH` to view the cached presentation. In this case not every `PATH` will yield a document because the commit could have occurred before the commit-time rendering was configured. Furthermore, it is possible to force a refresh of the cached presentation by retrieving the URL `REPOS/restful/presentation/cache/n/PATH`. (In fact, this is what the above-mentioned post-commit hook does.)

## 6 Conclusion and Future Work

We have presented  $\text{TNTBase}(\mathcal{F})$ , a format-specific extension of the TNTBase system that allows the (web) application developer to simply configure common format-specific workflows like validation, compilation, and presentation within the TNTBase system. This considerably reduces the application logic of (web) applications that manage large, changing XML-based document collections. The  $\text{TNTBase}(\mathcal{F})$  framework is implemented as an extension of TNTBase and can be obtained from the project website <http://trac.mathweb.org/tntbase>.  $\text{TNTBase}(\mathcal{F})$  is under active development and the project website should be consulted for details and the evolving state of implementation.

To get a better intuition for the power of the  $\text{TNTBase}(\mathcal{F})$  framework, let us look at the presentation of an ontology in Figure 4. This example from [Lan10] shows a heavily cross-referenced XHTML+MathML rendering of the underlying OMDoc sources, where much of the structural relations have been annotated in RDFa. The rendering and RDFa annotation process made use of the extracted interface information (see Section 3), which is then picked up by JOBAD [GLR09], a JavaScript library that instruments the XHTML for interactivity. For instance, JOBAD can use CSS properties to collapse a proof or embedded MathML `maction` elements or pick a different (pre-generated) notation variant. But JOBAD can also use AJAX-style callbacks to the  $\text{TNTBase}(OMDoc)$  system for definition lookup: For any symbol a right-click menu item will query TNTBase for the definition of the corresponding constant, which will then be served by  $\text{TNTBase}(OMDoc)$  and displayed in a popup by JOBAD. This example uses only the techniques from Sections 3 and 5: Pre-existing validation, annotation, rendering, and interaction functionalities were integrated into  $\text{TNTBase}(\mathcal{F})$ , which also serves as a web application platform [KGLZ09]. The work reported here was influenced by discussions of how to integrate TNTBase into the systems of the KWARC research group, in particular the SWiM system. The latter is an OMDoc-based semantic Wiki for scientific/technical documents [Lan08] [Lan10], in which workflows similar to the ones described in this paper had to be established for the integration of the OMDoc format into the underlying IkeWiki platform.

We are using  $\text{TNTBase}(OMDoc)$  in daily practice exploring the possibilities of  $\text{TNTBase}(\mathcal{F})$  for a semantic document format that is designed to support document aggregation and to stretch the limits of document modularization. The newly implemented virtual documents allow to aggre-

## Friend of a Friend (FOAF) vocabulary

imports from: [wordnet](#), [dc](#), [owl](#), [quant1](#), [logic1](#)

**AXIOM:**

The [foaf:Person](#) class is a sub-class of the [foaf:Agent](#) class, since all people are considered agents.

[Person ⊑ Agent](#)

**AXIOM:**  $\text{Person} \sqcap \text{Organization} = \perp$

**CONCEPT:** [made](#)

The [foaf:made](#) property relates

**TYPE:** [ObjectProperty](#)([Agent](#), [Thing](#))

**AXIOM:**  $\text{made} = \text{maker}^{-}$

**LEMMA:**  $\text{maker} = \text{made}^{-}$

**PROOF:** [collapse]

1. We know that  $\text{made} = m$  **TYPE:** [Property](#)([Class](#), [Class](#))
2. Interpreted using the model-theoretic semantics, this means that  
 $\text{made}^I = (\text{maker}^{-})^I = (\text{maker}^I)^{-}$ .
3. Now we apply the inverse on both sides, eliminate double inverses, and obtain  
 $(\text{made}^I)^{-} = ((\text{maker}^I)^{-})^{-} = \text{maker}^I$
4. This is just the interpretation of  $\text{maker} = \text{made}^{-}$ , which we had to prove.

**CONCEPT:** [membershipClass](#)

The [foaf:membershipClass](#) property relates a [foaf:Group](#) to an RDF class representing a sub-class of [foaf:Agent](#) whose instances are all the agents that are a [foaf:member](#) of the [foaf:Group](#). See [foaf:Group](#) for details and examples.

**AXIOM:**  $\forall m, g, C. (g \ni_{\text{member}} m \wedge \text{membershipClass}(g, C) \Rightarrow m : \text{type } C)$

German DL notation  
German DL notation  
Functional-style syntax  
Manchester syntax  
Hide formulæ

Export OWL as RDF/XML

Figure 4: A complex presentation of a documented Ontology

gate fragments of the existing document base and access, edit, and commit them, much like regular documents, if the XML format supports document aggregation in the sense we have defined in this paper. We are also looking into the use of TNTBase(*OWL*) for ontologies, which is semantically structured document format, but does not (directly) support document aggregation with the intention of developing a suitable generic XML extension that generalizes XInclude [MOV06] to queries.

Even in our limited experience with virtual files, they have turned out to be an enabling technology. Take for instance the example in Section 4.1. There we can use the virtual document in Listing 2 to give students access to fragments of the original XML documents (the problems) while hiding others (the solutions<sup>5</sup>). Here, virtual documents promise to enable (some) fine-grained access control in TNTBase; we plan to study the consequences of this idea in the near future.

The next larger step in the development of TNTBase will be the introduction of distribution facilities for versioned XML document storage supporting both push and pull-based workflows. We hope to gain not only distributed document management functionalities for TNTBase, but also to offer offline capabilities for web applications, which can then simply integrate the TNTBase library for transparent caching. In such applications, the TNTBase content would take the function of a subversion working copy with the additional ability of offline commits.

## References

- [BBC<sup>+</sup>07] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. W3C recommendation, World Wide Web Consortium (W3C), January 2007.

- [Ber] Berkeley DB XML.

<sup>5</sup>Assuming of course that we have configured TNTBase's file-level authentication and authorization to restrict access to the course materials to instructors.

- [GLR09] Jana Giceva, Christoph Lange, and Florian Rabe. Integrating web services into active mathematical documents. In Jacques Carette, Lucas Dixon, Claudio Sacerdoti Coen, and Stephen M. Watt, editors, *MKM/Calculemus 2009 Proceedings*, number 5625 in LNAI, pages 279–293. Springer Verlag, 2009.
- [KGLZ09] Michael Kohlhase, Jana Giceva, Christoph Lange, and Vyacheslav Zholudev. Jobad – interactive mathematical documents. In Brigitte Endres-Niggemeyer, Valentin Zacharias, and Pascal Hitzler, editors, *AI Mashup Challenge 2009, KI Conference*, 2009.
- [Koh06] Michael Kohlhase. OMDOC – *An open markup format for mathematical documents [Version 1.2]*. Number 4180 in LNAI. Springer Verlag, 2006.
- [Lan08] Christoph Lange. SWiM – a semantic wiki for mathematical knowledge management. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, editors, *ESWC*, volume 5021 of *Lecture Notes in Computer Science*, pages 832–837. Springer, 2008.
- [Lan10] Christoph Lange. *Semantic Web Collaboration on Semiformal Mathematical Knowledge*. PhD thesis, Jacobs University Bremen, 2010. submission expected in January 2010.
- [LAT] Latin: Logic atlas and integrator.
- [MOV06] Jonathan Marsh, David Orchard, and Daniel Veillard. XML inclusions (XInclude) version 1.0 (second edition). W3C Recommendation, World Wide Web Consortium (W3C), November 2006.
- [MVW05] Jonathan Marsh, Daniel Veillard, and Norman Walsh. `xml:id` version 1.0. W3C recommendation, World Wide Web Consortium (W3C), September 2005.
- [PCSF08] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control With Subversion*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2 edition, 2008.
- [RK09] Florian Rabe and Michael Kohlhase. A web-scalable module system for mathematical theories. Manuscript, to be submitted to the Journal of Symbolic Computation, 2009.
- [sch] schematron.
- [TNT] Tntbase trac.
- [WM08] Norman Walsh and Leonard Muellner. *DocBook 5.0: The Definitive Guide*. O'Reilly, 2008.
- [XQua] XQuery: An XML Query Language.
- [XQUb] XQUpdate: XQuery Update Facility 1.0.
- [ZK09] Vyacheslav Zholudev and Michael Kohlhase. TNTBase: a versioned storage for XML. In *Proceedings of Balisage: The Markup Conference 2009*, volume 3 of *Balisage Series on Markup Technologies*. Mulberry Technologies, Inc., 2009.
- [ZKR] Vyacheslav Zholudev, Michael Kohlhase, and Florian Rabe. A [insert xml format] database for [insert cool application] (extended version).

# Towards MKM in the Large: Modular Representation and Scalable Software Architecture

Michael Kohlhase, Florian Rabe, Vyacheslav Zholudev

Computer Science, Jacobs University Bremen

{m.kohlhase, f.rabe, v.zholudev}@jacobs-university.de

**Abstract.** MKM has been defined as the quest for technologies to manage mathematical knowledge. MKM “in the small” is well-studied, so the real problem is to scale up to large, highly interconnected corpora: “MKM in the large”. We contend that advances in two areas are needed to reach this goal. We need representation languages that support incremental processing of all primitive MKM operations, and we need software architectures and implementations that implement these operations scalably on large knowledge bases.

We present instances of both in this paper: the MMT framework for modular theory-graphs that integrates meta-logical foundations, which forms the base of the next OMDoc version; and TNTBase, a versioned storage system for XML-based document formats. TNTBase becomes an MMT database by instantiating it with special MKM operations for MMT.

## 1 Introduction

[12] defines the objective of MKM to be *to develop new and better ways of managing mathematical knowledge using sophisticated software tools* and later states the “Grand Challenge of MKM” as *a universal digital mathematics library (UDML)*, which is indeed a grand challenge, as it envisions that the UDML *would continuously grow and in time would contain essentially all mathematical knowledge*, which is estimated to be well in excess of  $10^7$  published pages.<sup>1</sup> All current efforts towards comprehensive machine-organizable libraries of mathematics are at least three orders of magnitude smaller than the UDML envisioned by Farmer in 2004: Formal libraries like those of Mizar ([33], Isabelle ([25]) or PVS ([24])) have ca.  $10^{4..x}$  statements (definitions and theorems). Even the semi-formal, commercial Wolfram MathWorld which hails itself *the world’s most extensive mathematics resource* only has  $10^{4..1}$  entries. There is anecdotal evidence that already at this size, management procedures are struggling.

To meet the MKM Grand Challenge will have to develop fundamentally more scalable ways of dealing with mathematical knowledge, especially since [12] goes on to postulate that the UDML *would also be continuously reorganized and consolidated as new connections and discoveries were made*. Clearly this can only be achieved algorithmically; experience with the libraries cited above already show that manual MKM does not scale sufficiently. Most of the work in the MKM community has concentrated

---

<sup>1</sup> For instance, Zentralblatt Math contains 2.4 million abstracts of articles from mathematical journals in the last 100 years.

on what we could call “MKM in the small”, i.e. dealing with aspects of MKM that do not explicitly address issues of very large knowledge collections; these we call “MKM in the large”.

In this paper we contribute to the MKM Grand Challenge of doing formal “MKM in the large” by analyzing scalability challenges inherent in MKM and propose steps towards solutions based on our MMT format, which is the basis for the next version of OMDOC. We justify our conclusions and recommendations for scalability techniques with concrete case studies we have undertaken in the last years. Section 2 tackles scalability issues pertaining to the representation languages used in the formalization of mathematical knowledge. Section 3 discusses how the modularity features of MMT can be realized scalably by realizing basic MKM functionality like validation, querying, and presentation incrementally and carefully evaluating the on-the-fly computation (and caching) of induced representations. These considerations, which are mainly concerned with efficient computation “in memory” are complemented with a discussion of mass storage, caching, and indexing in Section 4, which addresses scalability issues in large collections of mathematical knowledge. Section 5 concludes the paper and addresses avenues of further research.

## 2 A Scalable Representation Language

Our representation language MMT was introduced in [28]. It arises from three central design goals. Firstly, it should provide an expressive but simple **module system** as modularity is a necessary requirement for scalability. As usual in language design, the goals of simplicity and expressivity form a difficult trade-off that must be solved by identifying the right primitive module constructs. Secondly, scalability across semantic domains requires **foundation-independence** in the sense that MMT does not commit to any particular foundation (such as Zermelo-Fraenkel set theory or Church’s higher-order logic). Providing a good trade-off between this level of generality and the ability to give a rigorous semantics is a unique feature of MMT. Finally, scalability across implementation domains requires **standards-compliance**, and while using XML and OPENMATH is essentially orthogonal to the language design, the use of URIs as identifiers is not as it imposes subtle constraints that can be very hard to meet a posteriori.

MMT represents logical knowledge on three levels: On the **module level**, MMT builds on modular algebraic specification languages for logical knowledge such as OBJ [14], ASL [32], development graphs [1], and CASL [7]. In particular, MMT uses theories and theory morphism as the primitive modular concepts. Contrary to them, MMT only imposes very lightweight assumptions on the underlying language. This leads to a very simple generic module system that subsumes almost all aspects of the syntax and semantics of specific module systems such as PVS [24], Isabelle [25], or Coq [3].

On the **symbol level**, MMT is a simple declarative language that uses named symbol declarations where symbols may or may not have a type or a definiens. By experimental evidence, this subsumes virtually all declarative languages. In particular, MMT uses the Curry-Howard correspondence [8,17] to represent axioms and theorem as constants, and proofs as terms. Sets of symbol declarations yield theories and correspond to OPENMATH content dictionaries.

On the **object level**, MMT uses the formal grammar of OPENMATH [6] to represent mathematical objects without committing to a specific formal foundation. The semantics of objects is given proof theoretically using judgments for typing and equality between objects. MMT is parametric in these judgments, and the precise choice is relegated to a **foundation**.

## 2.1 Module System

Sophisticated mathematical reasoning usually involves several related but different mathematical contexts, and it is desirable to exploit these relationships by moving theorems between contexts. It is well-known that modular design can reduce space to an extent that is exponential in the depth of the reuse relation between the modules, and this applies in particular to the large theory hierarchies employed in mathematics and computer science.

The first applications of this technique in mathematics are found in the works by Bourbaki ([4,5]), which tried to prove every theorem in the context with the smallest possible set of axioms. MMT follows the “little theories approach” proposed in [11], in which separate contexts are represented by separate **theories**, and structural relationships between contexts are represented as **theory morphisms**, which serve as conduits for passing information (e.g., definitions and theorems) between theories (see [10]). This yields **theory graphs** where the nodes are theories and the paths are theory morphisms.

*Example 1 (Algebra).* For example, consider the theory graph in Fig. 1 for a portion of algebra, which was formalized in MMT in [9]. It defines the theory of magmas (A magma has a binary operation without axioms.) and extends it successively to monoids, groups, and commutative groups. Then the theory of rings is formed by importing from both **CGroup** (for the additive operation) and **Monoid** (for the multiplicative operation).

A crucial property here is that the imports are named, e.g., **Monoid** imports from **Magma** via an import named **mag**. While redundant in some cases, it is essential in **Ring** where we have to distinguish two theory morphisms from **Monoid** to **Ring**: The composition **add/grp/mon** for the additive monoid and **mult** for the multiplicative monoid.

The import names are used to form qualified names for the imported symbols. For example, if **\*** is the name of the binary operation in **Magma**, then **add/grp/mon/mag/\*** denotes addition and **mult/mag/\*** multiplication in **Ring**. Of course, MMT supports the use of abbreviations instead of qualified names, but it is a crucial prerequisite for scalability to make qualified names the default: Without named imports, every time we add a new name in **Magma** (e.g., for an abbreviation or a theorem), we would have to add corresponding renamings in **Ring** to avoid name clashes.

Another reason to use named imports is that we can use them to instantiate imports with theory morphisms. In our example, distributivity is stated separately as a theory that imports two magmas. Let us assume, the left distributivity axiom is stated as

$$\forall x, y, z. x \text{ mag1/*} (y \text{ mag2/*} z) = (x \text{ mag1/*} y) \text{ mag2/*} (x \text{ mag1/*} z)$$

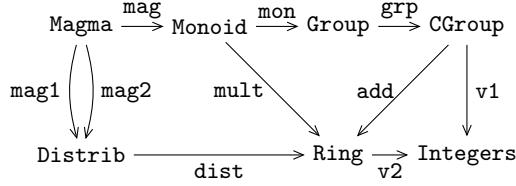
Then the import **dist** from **Distrib** to **Ring** will carry the instantiations **mag1**  $\mapsto$  **mult/mag** and **mag2**  $\mapsto$  **add/grp/mon/mag**.

In other module systems such as SML, such instantiations are called (asymmetric) sharing declarations. In terms of theory morphism, their semantics is a commutative diagram, i.e., an equality between two morphisms such as  $\text{dist}/\text{mag1} = \text{mult}/\text{mag} : \text{Magma} \rightarrow \text{Ring}$ . This provides MMT users and systems with a module level invariant for the efficient structuring of large theory graphs.

Besides imports, which induce theory morphisms into the containing theory, there is a second kind of edge in the theory graph: Views are explicit theory morphisms that represent translations between two theories. For example, the node on the right side of the graph represents a theory for the integers, say declaring the constants  $0, +, -, 1$ , and  $.$ . The fact that the integers are a commutative group is represented by the view  $v1$ : If we assume that  $\text{Monoid}$  declares a constant  $e$  for the unit element and  $\text{Group}$  a constant  $\text{inv}$  for the inverse element, then  $v1$  carries the instantiations  $\text{grp}/\text{mon}/\text{mag}/* \mapsto +$ ,  $\text{grp}/\text{mon}/e \mapsto 1$ , and  $\text{grp}/\text{inv} \mapsto -$ . Furthermore, every axiom declared or imported in  $\text{CGroup}$  is mapped to a proof of the corresponding property of the integers.

The view  $v2$  extends  $v1$  with corresponding instantiations for multiplication. MMT permits to write views modularly as well: When defining  $v2$ , we can import all instantiations of  $v1$  using  $\text{add} \mapsto v1$ . As above, the semantics of such an instantiation is a commutative diagram, namely  $v2 \circ \text{add} = v1$  as intended.

The major advantage of modular design is that every declaration — abbreviations, theorems, notations etc. — effects **induced declarations** in the importing theories. A disadvantage is that declarations may not always be located easily, e.g., the addition in a ring is declared in a theory that is four imports away. MMT finds a compromise here: Through qualified names, all induced declarations are addressable and locatable.



**Fig. 1.** Algebraic Hierarchy

*Case Study 1:* The formalization in [9] uses the Twelf module system ([31]), which is a special case of MMT. The implementation automatically flattens the theory graph, i.e., removes all modularity. The modular theory graph including all axioms and proofs can be written in 180 lines of Twelf code. The flattened graph is computed in less than half a second and requires more than 1800 lines.

The same case study defines two theories for lattices, one based on orderings and one based on algebra, and gives mutually inverse views to prove the equivalence of the two theories. Both definitions are modular: Algebraic lattices arise by importing twice from the theory of semi-lattices; order-based lattices arise by importing the infimum operation twice, once for the ordering and once for its dual. Consequently, the views can be given modularly as well, which is particularly important because views must map axioms to expensive-to-find proofs. These additional declarations take 310 lines of Twelf in modular and 3500 lines in flattened form.

These numbers already show the value of modularity in representation already in very small formalizations. If this is lacking, later steps will face severe scalability problems from blow-up in representation. Here, the named imports of MMT were the crucial innovation to strengthen modularity.

## 2.2 Foundation-Independence

Mathematical knowledge is described using very different foundations, and the most common foundations can be grouped into set theory and type theory. Within each group there are numerous variants, e.g., Zermelo-Fraenkel or Gödel-Bernays set theory, or set theories with or without the axiom of choice. Therefore, a single representation language can only be adequate if it is foundation-independent.

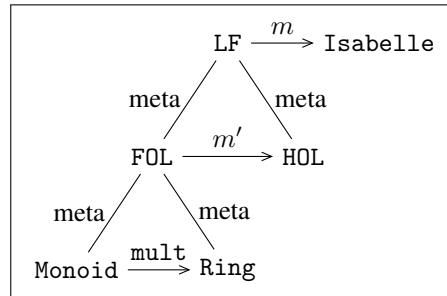
OPENMATH and OMDOC achieve this by concentrating on structural issues and leaving lexical ones to an external definition mechanism like content dictionaries or theories. In particular, this allows us to operate without choosing a particular foundational logical system, as we can just supply content dictionaries for the symbols in the particular logic. Thus, logics and in the same way foundations become theories, and we speak of the **logics-as-theories** approach.

But conceptually, it is helpful to distinguish levels here. To state a property in the theory CGroup like commutativity of the operation  $\circ := \text{grp}/\text{mon}/\text{mag}/*$  as  $\forall a, b. a \circ b = b \circ a$ , we use symbols  $\forall$  and  $=$  from first-order logic together with  $\circ$  from CGroup. Even though it is structurally possible to build algebraic theories by simply importing first-order logic, this would fail to describe the meta-relationship between the theories. But this relation is crucial, e.g., when interpreting CGroup in the integers, the symbols of the meta-language are not interpreted because a fixed interpretation is given in the context.

To understand this example better, we use the  $M/T$  notation for meta-languages.  $M/T$  refers to working in the object language  $T$ , which is defined within the meta-language  $M$ . For example, most of mathematics is carried out in  $FOL/ZF$ , i.e., first-order logic is the meta-language, in which set theory is defined.  $FOL$  itself might be defined in a logical framework such as  $LF$ , and within  $ZF$ , we can define the language of natural numbers, which yields  $LF/FOL/ZF/Nat$ . For algebra, we obtain, e.g.,  $FOL/\text{Magma}$ . MMT makes this meta-relation explicit: Every theory  $T$  may point to another theory  $M$  as its meta-theory. We can write this as  $MMT/(M/T)$  or simply  $M/T$ .

For example, in Fig. 2, the algebra example is extended by adding meta-theories. The theory  $FOL$  for first-order logic is the meta-theory for all algebraic theories, and the theory  $LF$  for the logical framework  $LF$  is the meta-theory of  $FOL$  and of the theory  $HOL$  for higher-order logic.

Now the crucial advantage of the logics-as-theories approach is that on all three levels the same module system can be used: For example, the views  $m$  and  $m'$  indicate possible translations on the levels of logical frameworks and logics, respectively. Similarly, logics and foundations can be built modularly. Thus, we can use imports to represent inheritance at the level of logical foundations and views to represent formal translations between them. Just like in the little theories approach, we



**Fig. 2.** Meta-Theories

can prove meta-logical results in the simplest foundation that is expressive enough and then use views to move results between foundations.

*Example 2 (Little Logics and Little Foundations).* In [15], we formalize the syntax, proof theory, and model theory and prove the soundness of first-order logic in MMT. Using the module system, we can treat all connectives and quantifiers separately. Thus, we can reuse these fragments to define other logics, and in [18] we do that, e.g., for sorted first-order logic and modal logic.

For the definition of the model theory, we need to formalize set theory in MMT, which is a significant investment, and even then doing proofs in set theory — as needed for the soundness proof — is tedious. Therefore, in [16], we develop the set theoretical foundation itself modularly. We define a typed higher-logic HOL first, which is expressive enough for many applications such as the above soundness proof. Then a view from HOL to ZF proves that ZF is a refinement of HOL and completes the proof of the soundness of FOL relative to models defined in ZF.

*Case Study 2:* Ex. 2 already showed that it is feasible to represent foundations and relations between foundations in MMT. Being able to this is a qualitative aspect of cross-domain scalability. We conducted another case study in [30] where we represent *LF/Isabelle* and *LF/Isabelle/HOL* ([25,23]) as well as a translation from them into *LF/FOL/ZFC*.

To our knowledge, MMT is the only declarative formalism in which comparable foundation or logic translations have been conducted. In Hets ([21]) a number of logic translations are implemented in Haskell. Twelf and Delphin provide logic and functional programming languages, respectively, on top of LF ([26,27]), which have been used to formalize the HOL-Nuprl translation ([22]).

### 2.3 Symbol Identifiers “in the Large”

In mathematical languages, we need to be able to refer to (i.e., identify) content objects in order to state the semantic relations. It was a somewhat surprising realization in the design of MMT that to understand the symbol identifiers is almost as difficult as to understand the whole module system. Theories are containers for symbol declarations, and relations between theories define the available symbols in any given theory. Since every available symbol should have a canonical identifier, the syntax of identifiers is inherently connected to the possible relations between theories.

In principle, there are two ways to identify content object: **by location** (relative to a particular document or file) and **by context** (relative to a mathematical theory). The first one essentially makes use of the organizational structure of files and file systems, and the second makes use of mathematical structuring principles supplied by the representation format.

As a general rule, it is preferable to use identification by context as the distribution of knowledge over file systems is usually a secondary consideration. Then the mapping between theory identifiers and physical theory locations can be deferred to an extralinguistic catalog. Resource identification by context should still be compatible with the URI-based approach that mediates most resource transport over the internet. This is

common practice in scalable programming languages such as Java where package identifiers are URIs and classes are located using the `classpath`.

For logical and mathematical knowledge, the OPENMATH 2 standard ([6]) and the current OMDOC version 1.2 define URIs for symbols. A symbol is identified by the symbol name and content dictionary, which in turn is identified by the CD name and the CD base, i.e., the URI where the CD is located. From these constituents, symbol URIs are formed using URI fragments (the part after the # delimiter). However, OPENMATH imposes a one-CD-one-file restriction, which is too restrictive in general. While OMDOC 1.2 permits multiple theories per file, it requires file-unique identifiers for all symbols. In both cases, the use of URI fragments, which are resolved only on the client, forces clients to retrieve the complete file even if only a single symbol is needed.

Furthermore, many module systems have features that impede or complicate the formation of canonical symbol URIs. Such features include unnamed imports, unnamed axioms, overloading, opening of modules, or shadowing of symbol names. Typically, this leads to a non-trivial correspondence between user-visible and application-internal identifiers. But this impedes or complicates cross-application scalability where all applications (ranging from, e.g., a Javascript GUI to a database backend) must understand the same identifiers.

MMT avoids the above pitfalls and introduces a simple yet expressive web-scalable syntax for symbol identifiers. An MMT-URI is of the form  $doc?mod?sym$  where

- $doc$  is a URI without query or fragment, e.g., <http://cds.omdoc.org/math/algebra/algebral.omdoc> which identifies (but not necessarily locates) an MMT document,
- $mod$  is a /-separated sequence of local names that gives the path to a nested theory in the above document, e.g., `Ring`,
- $sym$  is a /-separated sequence  $imp_1/\dots/imp_n/con$  of local names such that  $imp_i$  is an import and  $con$  a symbol name, e.g., `mult/mon/*`,
- a local name is of the form  $pchar^+$  where  $pchar$  is defined as in RFC 3986 [2], which — possibly via %-encoding — permits almost all Unicode characters.

In our running example, the canonical URI of multiplication in a ring is [http://cds.omdoc.org/math/algebra/algebral.omdoc?Ring?mult/mon/\\*](http://cds.omdoc.org/math/algebra/algebral.omdoc?Ring?mult/mon/*). Note that the use of two `?` characters in a URI is unusual outside of MMT, but legal wrt. RFC 3986. Of course, MMT also defines relative URIs that are resolved against the URI of the containing declaration. The most important case is when  $doc$  is empty. Then the resolution proceeds as in RFC 3986, e.g., `?mod'?sym'` resolves to  $doc?mod'?sym'$  relative to  $doc?mod?sym$  (Note that this differs from RFC 2396.). MMT defines some additional cases that are needed in mathematical practice and go beyond the expressivity of relative URIs: Relative to  $doc?mod?sym$ , the resolution of `??sym'` and `?/mod'?sym'` yields  $doc?mod?sym'$  and  $doc?mod/mod'?sym'$ , respectively.

*Case Study 3:* URIs are the main data structure needed for cross-application scalability, and our experience shows that they must be implemented by almost every peripheral system, even those that do not implement MMT itself. Already at this point, we had to implement them in SML ([31]), Javascript ([13]), XQuery ([35]), Haskell (for Hets, [21]), and Bean Shell (for a jEdit plugin) — in addition to the Scala-based reference API (Sect. 3).

This was only possible because MMT-URIs constitute a well-balanced trade-off between mathematical rigor, feasibility, and URI-compatibility: In particular, due to the use of the two separators / and ? (rather than only one), they can be parsed locally, i.e., without access to or understanding of the surrounding MMT document. And they can be dereferenced using standard URI libraries and URI-URL translations. At the same time, they provide canonical names for all symbols that are in scope, including those that are only available through imports.

### 3 A Scalable Implementation

As the implementation language for the MMT reference API, we pick Scala, a programming language designed to be *scalable*. Being functional, Scala permits elegant code, and based on and fully compatible with Java, it offers cross-application and web-level scalability.

The MMT API implements the syntax and semantics of MMT. It compiles to a 1 MB Java archive file that can be readily integrated into applications. Library and documentation can be obtained from [29]. Two technical aspects are especially notable from the point of view of scalability. Firstly, all MMT functionality is exposed to non-Java applications via a scriptable shell and via an HTTP servlet. Secondly, the API maintains an abstraction layer that separates the backends that physically store MMT documents (URLs) from the document identifiers (URIs). Thus, it is configurable which MMT documents are located, e.g., in a remote database or on the local file system. In the following section we describe some of the advanced features.

#### 3.1 Validation

Validation describes the process of checking MMT theory graphs against the MMT grammar and type system. MMT validation is done in three increasingly strict **stages**.

The first stage is XML validation against a context-free RelaxNG grammar. As this only catches errors in the surface syntax, MMT documents are validated **structurally** in a second stage. Structural validity guarantees that all declarations have unique URIs and that all references to symbols, theories, etc. can be resolved. This is still too lax for mathematics as it lacks type-checking. But it is exactly the right middle ground between the weak validation against a context-free grammar and the expensive and complex validation against a specific type system: On the one hand, it is efficient and foundation-independent, and on the other hand, it provides an invariant that is sufficient for many MKM services such as storage, navigation, or search.

Type-checking is foundation-specific, therefore each foundation must provide an MMT plugin that implements the typing and equality judgments. More precisely, the plugin must provide function that (semi-)decide for two given terms  $A$  and  $B$  over a theory  $T$ , the judgments  $\vdash_T A = B$  and  $\vdash_T A : B$ . Given such a plugin, a third validation stage can refine structural validity by additionally validating well-typedness of all declarations. For scalability, it is important that (i) these plugins are stateless as the theory graph is maintained by MMT, and that the (ii) modular structure is transparent to the plugin. Thus plugin developers only need to provide the core algorithms that

characterize a specific type system, and all MKM issues can be relegated to dedicated implementations.

Context-free validation is well-understood. Moreover, MMT is designed such that foundation-specific validation is obtained from structural validation by using the same inference system with some typing and equality constraints added. This leaves structural validation as the central issue for scalability.

*Case Study 4:* We have implemented structural validation by decomposing an MMT theory graph into a list of atomic declarations. For example, the declaration  $T = \{s_1 : \tau_1, s_2 : \tau_2\}$  of a theory  $T$  with two typed symbols yields the atomic declarations  $T = \{\}, T?s_1 : \tau$ , and  $T?s_2 : \tau_2$ . This “unnesting” of declarations is a special property of the MMT type system that is not usually found in other systems. It is possible because every declaration has a canonical URI and can therefore be taken out of its context.

This is important for scalability as it permits **incremental** processing. In particular, large MMT documents can be processed as streams of atomic declarations. Furthermore, the semantics of MMT guarantees that the processing order of these streams never matters if the (easily-inferable) dependencies between declarations are respected. This would even permit parallel processing, another prerequisite for scalability.

### 3.2 Querying

Once a theory graph has been read, MMT provides two ways how to access it: MMT-URI dereferencing and querying with respect to a simple ontology.

Firstly, a theory graph always has two forms: the modular form where all nodes are partial theories whose declarations are computed using imports, and the non-modular form where all imports are replaced with translated copies of the imported declarations. This is sometimes called **flattening**. Many implementations of module systems, e.g., Isabelle’s locales, automatically compute the flat form and do not maintain the modular form. This can be a threat to scalability as it can induce combinatorial explosion.

MMT maintains only the modular form. However, as every declaration present in the flat form has a canonical URI, the access to the flat form is possible via MMT-URI dereferencing: Dereferencing computes (and caches) the induced declarations present in the flat form. Thus, applications can ignore the modular structure and interact with a modular theory graph as if it were flattened, but the exponentially expensive flattening is performed transparently and incrementally.

Secondly, the API computes the ABox of a theory graph with respect to the MMT ontology. It has MMT-URIs as individuals and 10 types like `theory` or `symbol` as unary predicates. 11 binary predicates represent relations between individuals such as `HasDomain` relating an import to a theory or `HasOccurrenceOfInType` relating two symbols. These relations are structurally complete: The structure of a theory graph can be recovered from the ABox. The computation time is negligible as it is a byproduct of validation anyway.

The API includes a simple query language for this ontology. It can compute all individuals in the theory graph that are in a certain relation to a given individual. The possible queries include composition, union, transitive closure, and inverse of relations.

The ABox can also be regarded as the result of **compiling** an MMT theory graph. Many operations on theory graphs only require the ABox: for example the computation of the forward or backward dependency cone of a declaration which are needed to generate self-contained documents and in change management, respectively. This is important for cross-application scalability because applications can parse the ABox very easily. Moreover, we obtain a notion of **separate compilation**: ABox-generation only requires structural validity, and the latter can be implemented if only the ABoxes of the referenced files are known.

*Case Study 5:* Since all MMT knowledge items have a globally unique MMT-URI, being able to dereference them is sufficient to obtain complete information about a theory graph. We have implemented a web servlet for MMT that can act as a local proxy for MMT-URIs and as a URI catalog that maps MMT-URIs into (local or remote) URLs. The former means that all MMT-URIs are resolved locally if possible. The latter means that the MMT-URI of a module can be independent from its physical location. The same servlet can be run remotely, e.g., on the same machine as a mathematical database and configured to retrieve files directly from there or from other servers.

Thus systems can access all their input documents by URI via a local service, which makes all storage issues transparent. (Using presentation, see below, these can even be presented in the system's native syntax.) This solves a central problem in current implementations of formal systems: the restriction to in-memory knowledge. Besides the advantages of distributed storage and caching, a simple example application is that imported theories are automatically included when remote documents are retrieved in order to avoid successive lookups.

### 3.3 Presentation

MMT comes with a declarative language for notations similar to [19] that can be used to transform MMT theory graphs into arbitrary output formats. Notations are declared by giving parameters such as fixity and input/output precedence, and snippets for separators and brackets. Notations are not only used for mathematical objects but also for all MMT expressions, e.g. theory declarations and theory graphs.

Two aspects are particularly important for scalability. Firstly, sets of notations (called *styles*) behave like theories, which are sets of symbols. In particular, styles and notations have MMT-URIs (and are part of the MMT ontology), and the MMT module system can be used for inheritance between styles.

Secondly, every MMT expression has a URI  $E$ , for declarations this is trivial, for most mathematical objects it is the URI of the head symbol. Correspondingly, every notation must give an MMT-URI  $N$ , and the notation is applicable to an expression if  $N$  is a prefix of  $E$ . More specific notations can inherit from more general ones, e.g., the brackets and separators are usually given only once in the most general notation. This simplifies the authoring and maintenance of styles for large theory graphs significantly.

*Case Study 6:* In order to present MMT content as, e.g., HTML with embedded presentation MATHML, we need a style with only the 20 generic notations given in <http://alpha.tntbase.mathweb.org/repos/cds/omdoc/mathml.omdoc>.

For example, the notation declaration on the right applies to all constants whose `cdbase` starts with `http://cds.omdoc.org/` and renders OMS elements as `mo` elements. The latter has an `xref` attribute that links to the parallel markup (which is included by notations at higher levels). The content of the `mo` elements is a “hole” that is by default filled with the second component, for constants that is the name (0 and 1 are `cdbase` and `cd`).

---

```

<notation for="http://cds.omdoc.org/" role="constant">
  <element name="mo">
    <attribute name="xref">
      <text value="#" /><id/>
    </attribute>
    <hole><component index="2"/></hole>
  </element>
</notation>

```

---

This scales well because we can give notations for specific theories, e.g., by saying that `?Magma?*` is associative infix and optionally giving a different operator symbol than `*`. We can also add other output formats easily: Our implementation (see [18]) extends the above notation with a `jobad:href` attribute containing the MMT-URI — this URI is picked up by our JOBAD Javascript ([13]) for hyperlinking.

## 4 A Scalable Database

The TNTBase system [34] is a versioned XML-database with a client-server architecture. It integrates Berkeley DB XML into a Subversion server. DB XML stores HEAD revisions of XML files; non-XML content like PDF, images or L<sup>A</sup>T<sub>E</sub>X source files, differences between revisions, directory entry lists and other repository information are retained in a usual SVN back-end storage (Berkeley DB in our case). Keeping XML documents in DB XML allows accessing files not only via any SVN client but also through the DB XML API that supports efficient querying of XML content via XQuery and (versioned) modification of that content via XQuery Update.

In addition, TNTBase provides a plugin architecture for document format-specific customizations [35]. Using OMDOC as concrete syntax for MMT and the MMT API as a TNTBase plugin, we have made TNTBase MMT-aware so that data-intensive MMT algorithms can be executed within the database.

The TNTBase system and its documentation are available at <http://tntbase.org>. Below we describe some of the features particularly relevant for scalability.

### 4.1 Generating Content

Large scale collaborative authoring of mathematical documents requires **distributed** and **versioned** storage. On the language end, MMT supports this by making all identifiers URIs so that MMT documents can be distributed among authors and networks and reference each other. On the database end, TNTBase supports this by acting as a versioned MMT database.

In principle, versioning and distribution could also be realized with a plain SVN server. But for mathematics, it is important that the storage backend is aware of at least some aspects of the mathematical semantics. In large scale authoring processes, an important requirement is to guarantee consistency, i.e., it should be possible to reject commits of invalid documents. Therefore, TNTBase supports document format-specific **validation**.

For scalability, it is crucial that validation of interlinked collections of MMT documents is **incremental**, i.e., only those documents added or changed during a commit are validated. This is a significant effect because the committed documents almost always import modules from other documents that are already in the database, and these should not be revalidated — especially not if they contain unnecessary modules that introduce further dependencies.

Therefore, we integrate MMT separate compilation into TNTBase. During a commit TNTBase validates all committed files structurally by calling the MMT API. After successful validation, the ABox is generated and immediately stored in TNTBase. References to previously committed files are not resolved; instead their generated ABox is used for validation. Thus, validation is limited to the committed documents.

*Case Study 7:* In the LATIN project [18], we create an atlas of logics and logic translations formalized in MMT. At the current early stage of the project 5 people are actively editing so far about 100 files. These contain about 200 theories and 50 views, which form a single highly inter-linked MMT theory graph. We use TNTBase as the validity-guaranteeing backend storage.

The LATIN theory graph is highly modular. For example, the document giving the set-theoretical model theory of first-order logic from [16] depends on about 100 other theories. (We counted them conveniently using an XQuery, see below.) Standalone validation of this document takes about 15 seconds if needed files are retrieved from a local file system. Using separate compilation in TNTBase, it is almost instantaneous.

In fact, we can configure TNTBase so that structural validation is preceded by RelaxNG validation. This permits the MMT application to drop inefficient checks for syntax errors. Similarly, structural validation could be preceded by foundation-specific validation, but often we do not have a well-understood notion of separate compilation for specific foundations. But even in that case, we can do better than naive revalidation. MMT is designed so that it is foundation-independent which modules a given document depends on. Thus, we can collect these modules in one document using an efficient XQuery (see below) and then revalidate only this document. Moreover, we can use the presentation algorithm from Sect. 3.3 to transform the generated document into the input syntax of a dedicated implementation.

## 4.2 Retrieving Content

While the previous section already showed some of the strength of an MMT-aware TNTBase, its true strength lies in retrieving content. As every XML-native database, TNTBase supports XQuery but extends the DB XML syntax by a notion of file system path to address path-based collections of documents. Furthermore, it supports indexing to improve performance of queries and the querying of previous revisions. Finally, custom XQuery modules can be integrated into TNTBase.

MMT-aware retrieval is achieved through two measures. Firstly, **ABox caching** means that for every committed file, the MMT ABox is generated and stored in TNTBase. The ABox contains only two kinds of declarations — instances of unary and binary predicates — and is stored as a simple XML document. The element types in these documents are **indexed**, which yields efficient global queries.

*Example 3.* An MMT document for the algebra example from Sect. 2.1 can be obtained at <http://alpha.tntbase.mathweb.org/repos/cds/math/algebra/algebra1.omdoc>, its ABox is cached at <http://alpha.tntbase.mathweb.org:8080/tntbase/cds/restful/integration/validation/mmt/content/math/algebra/algebra1.omdoc>.

Secondly, custom **XQuery functions** utilize the cached and indexed ABoxes to provide efficient access to frequently needed aggregated documents. These include in particular the forward and backward dependency cones of a module. The backward dependency cone of a module  $M$  is the minimal set of modules needed to make  $M$  well-formed. Dually, the forward cone contains all modules that need  $M$ . If it were not for the indexed ABoxes, the latter would be highly expensive to compute: linear in the size of the database.

*Case Study 8:* The MMT presentation algorithm described in Sect. 3.3 takes a set of notations as input. However, additional notations may be given in imported theories, typically format-independent notations such as the one making ?Magma?\* infix. Therefore, when an MMT expression is rendered, all imported theories must be traversed for the sole reason of obtaining all notations.

Without MMT awareness in TNTBase, this would require multiple successive queries which is particularly harmful when presentation is executed locally while the imported theories are stored remotely. But even when all theories are available on a local disk, these successive calls already take 1.5 seconds for the above algebra document. (Once the notations are retrieved, the presentation itself is instantaneous.)

In MMT-aware TNTBase, we can retrieve all notations in the backward dependency closure of the presented expression with a single XQuery. ABox-indexing made this instantaneous up to network lag.

TNTBase does not only permit the efficient retrieval of such generated documents, but it also permits to commit edited versions of them. We call these **virtual documents** in [35]. These are essentially “XML database views” analogous to views in relational databases. They are editable, and TNTBase transparently patches the differences into the original files in the underlying versioning system.

*Case Study 9:* While manual refactoring of large theory graphs is as difficult as refactoring large software, there is virtually no tool support for it. For MMT, we obtain a simple renaming tool using a virtual document for the one-step (i.e., non-transitive) forward dependency cone of a theory  $T$  (see [35] for an example). That contains all references to  $T$  so that  $T$  can be renamed and all references modified in one go.

## 5 Conclusion and Future Work

This paper aims to pave the way for MKM “in the large” by proposing a theoretical and technological basis for a “Universal Digital Mathematics Library” (UDML) which has been touted as the grand challenge for MKM. In a nutshell, we conclude that the problem of scalability has to be addressed on all levels: we need modularity and accessibility of induced declarations in the representation format, incrementality and memoization in the implementation of the fundamental algorithms, and a mass storage solution that

supports fragment access and indexing. We have developed prototypical implementations and tested them on a variety of case studies.

The next step will be to integrate the parts and assemble a UDML installation with these. We plan to use the next generation of the OMDOC format, which will integrate the MMT infrastructure described in this paper as an interoperability layer; see [20] for a discussion of the issues involved. In the last years, we have developed OMDOC translation facilities for various fully formal theorem proving systems and their libraries. In the LATIN project [18], we are already developing a graph of concrete “logics-as-theories” to make the underlying logics interoperable.

## References

1. S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an Evolutionary Formal Software-Development Using CASL. In D. Bert, C. Choppy, and P. Mosses, editors, *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1999.
2. T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, Internet Engineering Task Force, 2005.
3. Y. Bertot and P. Castéran. *Coq’Art: The Calculus of Inductive Constructions*. Springer, 2004.
4. N. Bourbaki. *Theory of Sets*. Elements of Mathematics. Springer, 1968.
5. N. Bourbaki. *Algebra I*. Elements of Mathematics. Springer, 1974.
6. S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See <http://www.openmath.org/standard/om20>.
7. CoFI (The Common Framework Initiative). *CASL Reference Manual*, volume 2900 (IFIP Series) of *LNCS*. Springer, 2004.
8. H. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.
9. S. Dumbrava, F. Horozal, and K. Sojakova. A Case Study on Formalizing Algebra in a Module System. In *Workshop on Modules and Libraries for Proof Assistants*, 2009. To appear.
10. W. Farmer. An Infrastructure for Intertheory Reasoning. In D. McAllester, editor, *Conference on Automated Deduction*, pages 115–131. Springer, 2000.
11. W. Farmer, J. Guttman, and F. Thayer. Little Theories. In D. Kapur, editor, *Conference on Automated Deduction*, pages 467–581, 1992.
12. William M. Farmer. Mathematical Knowledge Management. In David G. Schwartz, editor, *Encyclopedia of Knowledge Management*, pages 599–604. Idea Group Reference, 2005.
13. J. Giceva, C. Lange, and F. Rabe. Integrating Web Services into Active Mathematical Documents. In J. Carette and L. Dixon and C. Sacerdoti Coen and S. Watt, editor, *Intelligent Computer Mathematics*, volume 5625 of *Lecture Notes in Computer Science*, pages 279–293. Springer, 2009.
14. J. Goguen, Timothy Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. In J. Goguen, D. Coleman, and R. Gallimore, editors, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.
15. F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. In *Fourth Workshop on Logical and Semantic Frameworks, with Applications*, volume 256 of *Electronic Notes in Theoretical Computer Science*, pages 49–65, 2009.
16. F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. Under review, see [http://kwarc.info/frabe/Research/EArabe\\_folsound\\_10.pdf](http://kwarc.info/frabe/Research/EArabe_folsound_10.pdf), 2010.

17. W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.
18. M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN Project, 2009. See <https://trac.omdoc.org/LATIN/>.
19. M. Kohlhase, C. Müller, and F. Rabe. Notations for Living Mathematical Documents. In S. Autexier and J. Campbell and J. Rubio and V. Sorge and M. Suzuki and F. Wiedijk, editor, *Mathematical Knowledge Management*, volume 5144 of *Lecture Notes in Computer Science*, pages 504–519, 2008.
20. M. Kohlhase, F. Rabe, and C. Sacerdoti Coen. A Foundational View on Integration Problems. Submitted to CALCULEMUS, 2010.
21. T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522, 2007.
22. P. Naumov, M. Stehr, and J. Meseguer. The HOL/NuPRL proof translator - a practical approach to formal interoperability. In *14th International Conference on Theorem Proving in Higher Order Logics*. Springer, 2001.
23. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.
24. S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752. Springer, 1992.
25. L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
26. F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.
27. A. Poswolsky and C. Schürmann. System Description: Delphin A Functional Programming Language for Deductive Systems. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Metalanguages: Theory and Practice*, pages 135–141. ENTCS, 2008.
28. F. Rabe. *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen, 2008. Available at <http://kwarc.info/frabe/Research/phdthesis.pdf>.
29. F. Rabe. The MMT Project, 2008. See <https://trac.kwarc.info/MMT/>.
30. F. Rabe and M. Iancu. A Formalized Set-Theoretical Semantics of Isabelle/HOL. Under review, see [http://kwarc.info/frabe/Research/rabeEA\\_isabelle\\_10.pdf](http://kwarc.info/frabe/Research/rabeEA_isabelle_10.pdf), 2010.
31. F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, pages 40–48. ACM Press, 2009.
32. D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. In M. Karpinski, editor, *Fundamentals of Computation Theory*, pages 413–427. Springer, 1983.
33. A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985.
34. V. Zholudev and M. Kohlhase. TNTBase: a Versioned Storage for XML. In *Proceedings of Balisage: The Markup Conference 2009*, volume 3. Mulberry Technologies, Inc., 2009.
35. V. Zholudev, M. Kohlhase, and F. Rabe. A [insert XML Format] Database for [insert cool application]. In *Proceedings of XMLPrague*, 2010. To appear.

# Combining Source, Content, Presentation, Narration, and Relational Representation

Fulya Horozal, Alin Iacob, Constantin Jucovschi,

Michael Kohlhase, Florian Rabe

{f.horozal,a.iacob,c.jucovschi,m.kohlhase,f.rabe}@jacobs-university.de

Computer Science, Jacobs University, Bremen

**Abstract.** In this paper, we try to bridge the gap between different dimensions/incarnations of mathematical knowledge: MKM representation formats (content), their human-oriented languages (source, presentation), their narrative linearizations (narration), and relational presentations used in the semantic web. The central idea is to transport solutions from software engineering to MKM regarding the parallel interlinked maintenance of the different incarnations. We show how the integration of these incarnations can be utilized to enrich the authoring and viewing processes, and we evaluate our infrastructure on the LATIN Logic Atlas, a modular library of logic formalizations, and a set of computer science lecture notes written in  $\text{\LaTeX}$  – a modular, semantic variant of  $\text{\TeX}$ .

## 1 Introduction

The Mathematical Knowledge Management (MKM) community has developed XML-based content representations of mathematical formulae and knowledge that are optimized for machine-to-machine communication. They serve as archiving formats, make mathematical software systems and services interoperable and allow to develop structural services like search, documentation, and navigation that are independent of mathematical foundations and logics.

However, these formats are — by their nature — inappropriate for the communication with humans. Therefore the MKM community uses languages that are less verbose, more mnemonic, and often optimized for a specific domain for authoring. Such human-oriented languages (we call them source languages) are converted — via a complex compilation process — into the content representations for interaction with MKM services, ideally without the user ever seeing them. In addition, we have designed presentation-oriented languages that permit an enriched reading experience compared to the source language.

This situation is similar to software engineering, where programmers write code, run the compiled executables, build HTML-based API documentations, but expect, e.g., the documentation and the results of debugging services in terms of the sources. In software engineering, scalable solutions for this problem have been developed and applied successfully, which we want to transfer to MKM.

The work described here originates from our work on two large collections of mathematical documents: our LATIN logic atlas [KMR09] formalized in the logical framework LF; and our General Computer Science lecture notes written in L<sup>A</sup>T<sub>E</sub>X. Despite their different flavor, both collections agree in some key aspects: They are large, highly structured, extensively inter-connected, and both authoring and reading call for machine support.

Moreover, they must be frequently converted between representation dimensions optimized for different purposes: a human-friendly input representation (source), a machine-understandable content markup (content), interactive documents for an added-value reading experience (presentation-content parallel markup), a linearized structure for teaching and publication (narration), and a network of linked data items for integration with the semantic web (relational).

In Sect. 2, we first give an overview over these document collections focusing on the challenges they present to knowledge management. Then we design a knowledge representation methodology that integrates these different dimensions in Sect. 3. In Sect. 4 and 5, we show how we leverage this methodology in the authoring and the viewing process both of which benefit from a seamless integration of the knowledge dimensions.

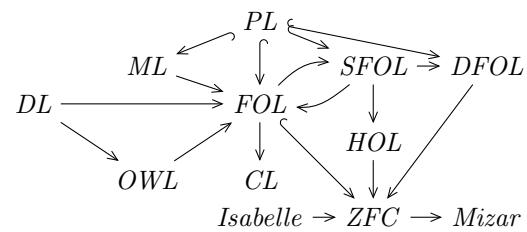
## 2 Structured Document Collections

### 2.1 The LATIN Logic Atlas

The LATIN Logic Atlas is a library of formalizations of logics and related formal systems as well as translations between them. It is intended as a reference and documentation platform for logics commonly used in mathematics and computer science. It uses a foundationally unconstrained logical framework based on modular LF and its Twelf implementation [HHP93; PS99; RS09] and focuses on modularity and extensibility.

The knowledge in the Logic Atlas is organized as a graph of LF signatures and signature morphisms between them. The latter are split into inheritance translations (inclusions/imports) and representation theorems, which have to be proved. It contains formalizations of type theories, set theories, and logics. Among them are, for example, propositional (*PL*), first (*FOL*) and higher-order logic (*HOL*), sorted (*SFOL*) and dependent first-order logic (*DFOL*), description logics (*DL*), modal (*ML*) and common logic (*CL*) as illustrated in the diagram below. Single arrows ( $\rightarrow$ ) in this diagram denote translations between formalizations and hooked arrows ( $\hookrightarrow$ ) denote imports.

All logics are designed modularly formed from orthogonal theories for individual connectives, quantifiers, and axioms. For example, the classical  $\wedge$  connective is only declared once in the whole Logic Atlas, and



the axiom of excluded middle and its consequences reside in a separate signature. We also use individual theories for syntax, proof theory, model theory so that the same syntax can be combined with different interpretations.

As a running example, we introduce a very simple fragment of the formalization of the syntax of propositional logic.

*Example 1* (Propositional Logic) The formalization of propositional logic syntax consists of the LF signatures illustrated in Fig. 1. We focus on the structural aspects and omit the details of LF. We four signatures living in two different namespaces. `BASE` declares a symbol for the type of propositions. It is imported into `CONJ` and `IMP` which declare conjunction and implication, respectively, and these are imported `PROP`.

```
%namespace = "http://cds.omdoc.org/logics"
%sig BASE = {
    %% Type of propositions
    o : type.
}

%namespace = "http://cds.omdoc.org/logics/propositional"
%sig CONJ = { %include BASE. ... }
%sig IMP = { %include BASE. ... }
%sig PROP = { %include CONJ. %include IMP. }
```

**Fig. 1.** Formalization of Propositional Logic Syntax in Twelf

Overall, the atlas contains over 500 LF signatures, and their highly modular structure yields a large number of inheritance edges. Additionally, the representation theorems include:

- the translation from unsorted to sorted first-order logic (which is almost but not quite an inclusion),
- the translations by relativization of quantifiers from sorted first-order, modal, and description logics to unsorted first-order logic, in most cases including the translation of the model theory,
- the translation from propositional and sorted first-order logic to Andrews-style higher-order logic,
- the negative translation from classical to intuitionistic logic,
- the translation from type theory to set theory that interprets types as sets and terms as elements, including a translation of Isabelle/HOL to ZF set theory,
- the Curry-Howard correspondence between logic, type theory, and category theory.

All translations include translations of the proof theory, which guarantees their proof theoretical soundness due to the type-preservation of signature morphisms.

This leads to a highly interlinked non-linear structure of the atlas. Moreover, it is designed highly collaboratively with strong interdependence between the developers. Therefore, it leads to a number of MKM challenges.

For example, the LF modules are distributed over files and these files over directories. This structure is semantically transparent because all references to modules are made by URIs. The URIs are themselves hierarchical grouping the modules into nested namespaces. It is desirable that these namespaces do not have to correspond to source files or directories. Therefore, the mapping between URIs and URLs is non-trivial and a separate management challenge.

Another problem is that encodings in LF are typically very difficult to read for anybody but the author. In a collaborative setting, it is desirable to interact with the logic graph not only through the LF source syntax but also through browsable, cross-referenced XHMTL+MathML. These should be interactive and for example permit looking up the definition of a symbol or displaying the reconstructed type of a variable. While we have presented such an interface in [GLR09] already, the systematic integration into the authoring process, where the state of the art is a text editor, has so far been lacking.

## 2.2 Computer Science Lecture Notes

The GenCS corpus consists of the course notes and problems of a two-semester introductory course in Computer Science [Koh] held at Jacobs University by one of the authors in the last eight years. The course notes currently comprise 300 pages with over 500 slides organized in over 800 files; they are accompanied by a database of more than 1000 homework/exam problems. All course materials are authored and maintained in the `STEX` format [Koh08], a modular, semantic variant of `LATEX` that shares the information model with OMDoc; see Fig. 2 for an example. In our nomenclature, `STEX` is used as a source language that is transformed into OMDoc via the `LATEXML` daemon [GSK11]. For debugging and high-quality print the `STEX` sources can also be typeset via `pdflatex`, just as ordinary `LATEX` documents. The encoding makes central use of the modularity afforded by the theory graph approach; knowledge units like slides are encoded as “modules” (theories in OMDoc) and are interconnected by theory morphisms (module imports). Modules also introduce concepts via `\definiendum` and semantic macros via `\symdef`, these are inherited via the module import relation.

# 3 A Multi-Dimensional Knowledge Representation

## 3.1 Dimensions of Knowledge

In order to address the knowledge management challenges outlined above, we devise a methodology that permits the parallel maintenance of the orthogonal dimensions of the knowledge contained in a collection of mathematical documents. It is based on two key concepts: (i) a hierachic organization of dimensions and knowledge items in a file-system-like manner inspired by the project view from software engineering, and (ii) the use of MMT URIs [RK10] as a standardized way to interlink both between different knowledge items and between the different dimensions of the same knowledge item.

```

\begin{module}[id=trees]
  \symdef[name=tdepth]{tdepthFN}{\text{dp}}
  \symdef[tdepth][1]{\prefix\tdepthFN{\#1}}
  \begin{definition}[id=tree-depth.def]
    Let $\$ \text{defeq} \{T\} \{ \text{tup}\{V,E\} \}$ be tree, then the $\{\text{definiendum}\} \{ \text{tree-depth} \} \{ \text{depth} \}$
    $\$ \text{tdepth}\{v\} \$ $ of a node $\$ \text{inset}\{V\} \$ $ is defined recursively: $\$ \text{tdepth}\{r\}=0 \$ $ for
    the root $\$ r \$ $ of $\$ T \$ $ and $\$ \text{tdepth}\{w\}=1+\text{tdepth}\{w\} \$ $ if $\$ \text{inset}\{ \text{tup}\{v,w\} \} E \$ $.
  \end{definition}
  ...
\end{module}

\begin{module}[id=binary-trees]
  \importmodule[\KWARCslides{graphs-trees/en/trees}]{trees}
  ...
  \begin{definition}[id=binary-tree.def, title=Binary Tree]
    A $\{\text{definiendum}\} \{ \text{binary-tree} \} \{ \text{binary tree} \}$ is a $\{\text{termref}\} \{ \text{cd=trees, name=tree} \} \{ \text{tree} \}$
    where all $\{\text{termref}\} \{ \text{cd=graphs-intro, name=node} \} \{ \text{nodes} \}$
    have $\{\text{termref}\} \{ \text{cd=graphs-intro, name=out-degree} \} \{ \text{out-degree} \} 2 \text{ or } 0$.
  \end{definition}
  ...
\end{module}

```

**Fig. 2.** Semiformalization of two course modules

The MMT URI of a toplevel knowledge item is of the form  $g?M$  where  $g$  is the namespace and  $M$  the module name. Namespaces are URIs of the form  $\langle\!\langle \text{scheme} \rangle\!\rangle://[\langle\!\langle \text{userinfo} \rangle\!\rangle @]D_1\dots D_m[:\langle\!\langle \text{port} \rangle\!\rangle]/S_1/\dots/S_n$  where the  $D_i$  are domain labels and the  $S_i$  are path segments. Consequently,  $g?M$  is a well-formed URI as well.  $\langle\!\langle \text{userinfo} \rangle\!\rangle$ , and  $\langle\!\langle \text{port} \rangle\!\rangle$  are optional, and  $\langle\!\langle \text{userinfo} \rangle\!\rangle$ ,  $\langle\!\langle \text{scheme} \rangle\!\rangle$ , and  $\langle\!\langle \text{port} \rangle\!\rangle$  are only permitted so that users can form URIs that double as URLs — MMT URIs differing only in the scheme, userinfo, or port are considered equal.

We arrange a collection of mathematical documents as a folder containing the following subfolders, all of which are optional:

`source` contains the source files of a project. This folder does not have a predefined structure.

`content` contains a semantically marked up representation of the source files in the OMDoc format. Every namespace is stored in one file whose path is determined by its URI. Modules with namespace  $D_1. \dots .D_m/S_1/\dots/S_n$  reside in an OMDoc file with path `content/Dm/.../D1/S1/.../Sn.omdoc`. Each module carries an attribute `source="/PATH?colB:lineB-colE:lineE"` giving its physical location as a URL. Here PATH is the path to the containing file in the source, and colB, lineB, colE, and lineE give the begin/end column/line information.

`presentation` contains the presentation of the source files in the XHTML+MathML format with JOBAD annotations [GLR09]. It has the same file structure as the folder `content`. The files contain XHTML elements whose body has one child for every contained module. Each of these module has the attribute `jobad:href="URI"` giving its MMT URI.

`narration` contains an arbitrary collection of narratively structured documents. These are OMDoc files that contain narrative content such as sectioning and transitions, but no modules. Instead they contain reference elements of the form `<mref target="MMTURI"/>` that refer to MMT modules. It is common but not necessary that these modules are present in the `content` folder.

`relational` contains two files containing an RDF-style relational representation of the content according to the MMT ontology. Both are in XML format with toplevel element `mmtabox` and a number of children. In `individuals.abox`, the children give instances of unary predicates such as `<individual type="IsTheory" uri="MMTURI" source="PATH"/>`. In `relations.abox`, the children give instances of binary predicates such as `<relation subject="MMTURI1" predicate="ImportsFrom" object="MMTURI2" source="PATH"/>`. Usually, the knowledge items occurring in unary predicates or as the subject of a binary predicate are present in the content. However, the object of a binary predicate is often not present, namely when a theory imports a remote theory. In both cases, we use an attribute `source` to indicate the source that induced the entry; this is important for change management when one of the source files was changed.

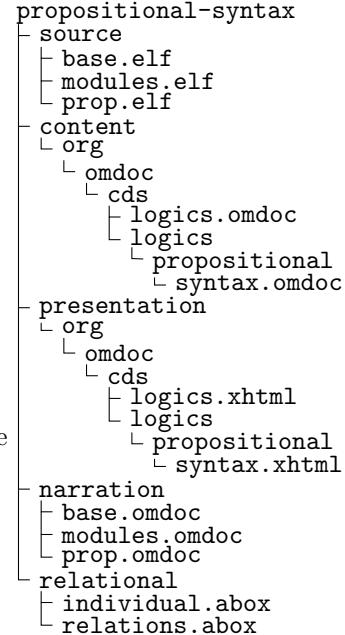
#### Example 2 (Continuing Ex. 1)

The directory structure for the signatures from Ex. 1 is given in Fig. 3 using a root folder named `propositional-syntax`. Here we assume that the subfolder `source` contains the Twelf source files `base.elf` which contains the signature `BASE`, `modules.elf` which contains `CONJ` and `IMP`, and `prop.elf` which contains `PROP`.

Based on the MMT URIs of the signatures in the source files, their content representation is given as follows. The signature `BASE` has the MMT URI `http://cds.omdoc.org/logics?BASE`. The other signatures have MMT URIs such as `http://cds.omdoc.org/logics/propositional/syntax?CONJ`. The content representation of the signature `BASE` is given in the OMDoc file `content/org/omdoc/cds/logics.omdoc`. The other content representations reside in the file `content/org/omdoc/cds/logics/propositional/syntax.omdoc`.

The subfolder `presentation` contains the respective XHTML files, `logics.xhtml` and `syntax.xhtml`. All files in Fig. 3 can be downloaded at <https://svn.kwarc.info/repos/twelf/projects/propositional-syntax>.

Our methodology integrates various powerful conceptual distinctions that have been developed in the past. Firstly, our distinction between the source and the content representation corresponds to the distinction between source and binary in software engineering. Moreover, our directory structure is inspired by software projects, such as in Java programming. In particular, the use of URIs



**Fig. 3.** Files of the Running Example

to identify content (binary) items corresponds to identifiers for Java classes. Therefore, existing workflows and implementations from software engineering can be easily adapted, for example in the use of project-based IDEs (see Sect. 4).

Secondly, the distinction between content and presentation has been well studied in the MKM community and standardized in MathML [Aus+03]. In particular, the cross-references from presentation to content correspond to the interlinking of content and presentation in the parallel markup employed in MathML, which we here extend to the level of document collections.

Thirdly, the distinction between content and narrative structure was already recognized in the OMDoc format. The general intuition there is that narrative structures are “presentations” at the discourse level. But in contrast to the formula level, presentations cannot be specified and managed via notation definitions. Instead we add narrative document structure fragments, i.e. document-structured objects that contain references to the content representations and transition texts as lightweight structures to the content commons; see [Mü10] for details and further references.

Finally, the distinction between tree-structured content representation and the relational representation corresponds to the practice of the semantic web where RDF triples are used to represent knowledge as a network of linked data. see [Lan11] for an overview.

### 3.2 A Mathematical Archive Format

We will now follow the parallelism to software engineering developed in the previous section: We introduce mathematical archives — `mar` files — that correspond to Java archives, i.e., `jar` files [Ora]. We define a mathematical archive to be a zip file that contains the directory structure developed in Sect. 3.1. By packaging all knowledge dimensions in a single archive, we obtain a convenient and lightweight way of distributing multi-dimensional collections of interlinked documents.

To address into the content of a `mar` archive, we also define the following URL scheme: Given a `mar` whole URL is `file:/A` and which contains the source file `source/S`, then the URL `mar:/A/S` resolves to that source file. We define the URL `mar:/A/S?Pos` accordingly if `Pos` is the position of a module given by its line/column as above.

Similarly, to the compilation and building process that is used to create `jar` files, we have implemented a building process for `mar` files. It consists of three stages. The first stage (compilation) depends on the source language and produce one OMDoc file for every source file whose internal structure corresponds to the source file. This is implemented in close connection with dedicated tools for the source language. In particular, we have implemented a translation from LF to OMDoc as part of the Twelf implementation of LF [PS99; RS09]. Moreover, we have implemented a translation from `STEX` to OMDoc based on the `LATEX` XML daemon [GSK11].

The second stage (building) is generic and produces the remaining knowledge dimensions from the OMDoc representation. In particular, it decomposes

the OMDoc documents into modules and reassembles them according to their namespaces to obtain the content representation. The narrative dimension is obtained from the initial OMDoc representation by replacing all modules with references to the respective content item. We have implemented this as a part of the existing MMT API [KRZ10]. Finally, the API already includes a rendering engine that we use to produce the presentation and the relational representation.

Then the third stage (packaging) collects all folders in a zip archive. For LF, we integrate all three stages into a flexible command line application.

*Example 3* (Continuing Ex. 2) The mathematical archive file for the running example can be obtained at <https://svn.kwarc.info/repos/twelf/projects/propositional-syntax.mar>.

### 3.3 Catalog Services

The use of URIs as knowledge identifiers (rather than URLs) is crucial in order to permit collaborative authoring and convenient distribution of knowledge. However, it requires a catalog that translates an MMT URIs to the physical location, given by a URL, of a resource. Typical URLs are those in a file system, in a mathematical archive, or a remote or local repository. It is trivial to build the catalog if the knowledge is already present in content form where locations are derived from the URI.

But the catalog is already needed during the compilation process: For example, if a theory imports another theory, it refers to it by its MMT URI. Consequently, the compilation tool must already be aware of the URI-to-URL mapping before the content has been produced. However, the compilation tool is typically a dedicated legacy system that natively operates on URLs already and does not even recognize URIs. This is the case for both Twelf and L<sup>A</sup>T<sub>E</sub>X.

Therefore, we have implemented standalone catalog services for these two tools and integrated them with the respective system. In the case of Twelf, the catalog maintains a list of local directories, files, and **mar** archives that it watches. It parses them whenever they change and creates the URI-URL mapping. When Twelf encounters a URI, it asks the catalog via HTTP for the URL. This parser only parses the outer syntax that is necessary to obtain the structure of the source file; it is implemented generically so that it can be easily adapted to other formal declarative languages.

An additional strength of this catalog is that it can also handle ill-formed source representations that commonly arise during authoring. Moreover, we also use the catalog to obtain the line/column locations of the modules inside the source files so that the content-to-source references can be added to the content files.

In the case of S<sup>T</sup>E<sub>X</sub>, a poor man's catalog services is implemented directly in T<sub>E</sub>X: the base URIs of the GenCS knowledge collection (see \KWARCslides in Fig 2) is specified by a \defpath statement in the document preamble and can be used in the \importmodule macros. The module environments induce internal T<sub>E</sub>X structures that store information about the imports (\importmodule)

structure and semantic macros (`\symdef`), therefore these three  $\text{\TeX}$  primitives have to be read whenever a module is imported. To get around difficulties with selective input in  $\text{\TeX}$ , the  $\text{\TeX}$  build process excerpts a  $\text{\TeX}$  signature module `⟨module⟩.sms` from any module `⟨module⟩.tex`. So `\importmodule{⟨module⟩.sms}` simply reads `⟨module⟩.sms`.

## 4 The Author’s Perspective

The translation from source to a content-like representation has been well-understood. For languages like LF, it takes the form of a parsing and type reconstruction process that transforms external to internal syntax. The translation from internal syntax to an OMDoc-based content representation is conceptually straightforward.

However, it is a hard problem to use the content representation to give the author feedback about the document she is currently editing. For most formal mathematical languages, the state of the art is an emacs mode with syntax highlighting. Only a few systems offer further functionality. For example, the Agda [Nor05] emacs mode can follow cross-references and show reconstructed types of missing terms. The Isabelle [Pau94] jEdit can follow cross-references and show tooltips derived from the static analysis.

A more powerful solution is possible if we always produce all knowledge dimensions using the compilation and building process as described in Sect. 3.2. Then generic services can be implemented easily, each of them based on the most suitable dimension, and we give a few examples in Sect. 4.2.

Note that this is not an efficiency problem: Typically the author only works on a few files that can be compiled constantly. It is even realistic to hold all dimensions in memory. The main problem is an architectural one, which is solved by our multi-dimensional representation. Once this architecture is setup and made available to IDE developers, it is very easy for them to quickly produce powerful generic services.

### 4.1 Multi-Dimensional Knowledge in an IDE

In previous work, we have already presented an example of a semantic IDE [JK10] based on Eclipse. We can now strengthen it significantly by basing it on our multi-dimensional representation. Inspired by the project metaphor from software engineering, we introduce the notion of a mathematical project in Eclipse.

A mathematical project consists of a folder containing the subfolder from Sect. 3.1. The author works on a set of source files in the `source` directory. Moreover, the project maintains a mathpath (named in analogy to Java’s classpath) that provides a set of `mar` archives that the user wishes to include.

The IDE offers the build functionality that runs the compilation and building processes described in Sect. 3.2 to generate the other dimensions from the source dimension. The key requirement here is to gracefully degrade in the presence of errors in the source file. Therefore, we provide an adaptive parser component that consists of three levels:

The **regex level** uses regular expressions to spot important structural properties in the document (e.g. the namespace and signature declarations in the case of LF). This compilation level never fails, and its result is an OMDoc file that contains only the spotted structures and lacks any additional information of the content.

The **CFG parser level** uses a simple context-free grammar to parse the source. It is able to spot more complicated structures such as comments and nested modules and can be implemented very easily within Eclipse. Like the previous level, it produces an approximate OMDoc file, but contrary to the previous level, it may find syntax errors that are then displayed to the user.

The **full parser level** uses the dedicated tool (Twelf or L<sup>A</sup>T<sub>E</sub>X). The resulting OMDoc fail includes the full content representation. In particular, in the case of Twelf, it contains all reconstructed types and implicit arguments. However, it may fail in the case of ill-typed input.

The adaptive parser component runs all parser in order, and retains the best OMDoc file any of them returns. This file is then used as the input to produce the remaining content dimensions.

## 4.2 Added-Value Services

In this section we present several services typically found in software engineering tools which aim at supporting authoring process. We analyze each of these services and show that they can be efficiently implemented by using one or several dimensions of knowledge.

**project explorer** is a widget giving an integrated view on a project's content by abstracting from the file system location where the sources are defined. It groups objects by their content location, i.e., their MMT URI. To implement this widget, we populate the non-leaf nodes of the tree from the directory structure of the content dimension. The leaf nodes are generated by running simple XPath queries on the OMDoc files.

**outline view** is a source level widget which visualizes the main structural components. For LF, these include definitions of signatures and namespaces as well as constant declarations within signatures. Double-clicking on any such structural components opens the place in the source code where the component is defined. Alternatively, the corresponding presentation can be opened.

**autocompletion** assists the user with getting location and context specific suggestions, e.g., listing declarations available in a namespace. Fig. 4a shows an example. Note how the namespace prefix `base` is declared to point to a certain namespace, and the autocompletion suggests only signatures names declared in that namespace. The implementation of this feature requires information about the context where autocompletion is requested, which is obtained from the interlinked source and content dimensions. Moreover, it needs the content dimension to compute all possible completions. In more complicated scenarios, it can also use the relational dimension to compute the possible completions using the relational queries.

**hover overlay** is a feature that shows in-place meta-data about elements at the position of the mouse cursor such as the full URIs of a symbol, its type or definitions, a comment, or inferred types of variables. Fig. 4b) shows an example. The displayed information is retrieved from the content dimension. It is also possible to display the information using the presentation dimension. **definition/reference search** makes it easy for a user to find where a certain item is defined or used. Although the features require different user interfaces the functionality is very similar, namely, finding relations. Just like in the hover overlay feature, one first finds the right item in the content representation and then use the relation dimension to find the requested item(s). **theory-graph display** provides a graphical visualization of the relations among knowledge items. To implement this feature we apply a filter on the multi-graph from the relations dimension and uses 3rd party software to render it.

The figure consists of two screenshots of a code editor. On the left, a code snippet in OMDoc syntax is shown. Line 11 contains the word 'Base'. A small red rectangular box highlights this word, indicating it is being selected. On the right, a larger screenshot shows the same code with a green callout bubble hovering over the highlighted word 'Base'. The callout bubble contains the text 'Type of propositions.' followed by a URL: 'http://cds.omdoc.org/logics#proposition'. The rest of the code is visible below the callout.

```

1 %namespace "http://cds.omdoc.org/logics/propositional/syntax".
2 %namespace base = "http://cds.omdoc.org/logics".
3
4 %sig Truth = {
5   %include base.Base  %open.
6   true : o.
7 }.
8
9 %sig Falsity = {
10  %include base.Base
11  Base
12}.
13
14 %sig NEG = {
15  %include base.Base  %open.
16  not : o -> o.  %prefix 20 not.

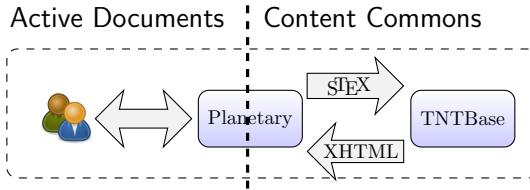
```

**Fig. 4.** a) Context aware Auto-Completion b) Metadata information on hover

## 5 The Reader's Perspective

We have developed the Planetary system (see [Koh+11; Dav+10; Pla] for an introduction) as the reader's complement to our IDE. Planetary is a Web 3.0 system<sup>1</sup> for semantically annotated document collections in Science, Technology, Engineering and Mathematics (STEM). In our approach, *documents published in the Planetary system become flexible, adaptive interfaces to a content commons* of domain objects, context, and their relations.

We call this framework the **Active Documents Paradigm** (ADP), since documents can also actively adapt to user preferences and environment rather than only executing services upon user request. Our



**Fig. 5.** The Active Documents Architecture

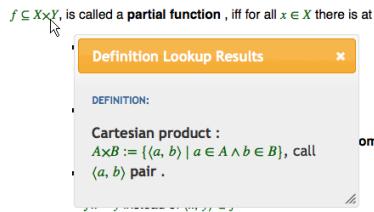
<sup>1</sup> We adopt the nomenclature where Web 3.0 stands for extension of the Social Web with Semantic Web/Linked Open Data technologies.

framework is based on *semantically annotated documents* together with semantic background ontologies (which we call the **content commons**). This information can then be used by user-visible, semantic services like program (fragment) execution, computation, visualization, navigation, information aggregation and information retrieval [GLR09].

The Planetary system directly uses all five incarnations/dimensions of mathematical knowledge specified in Sect 3.1. In the **content** incarnation, Planetary uses OMDoc for representing content modules, but authors create and maintain these using STEX in the **source** dimension and the readers interact with the active documents encoded as dynamic XHTML+MathML+RDFa (the **source** incarnation of the material). We use the LATEXML daemon [Mil; GSK11] for the transformation from STEX to OMDoc, this is run on every change to the STEX sources. The basic presentation process [KMR08] process for the OMDoc content modules is provided by the TNTBase system [ZK09].

But Planetary also uses the **narrative** dimension: content modules are used not only for representing mathematical theories, but also for document structures: **narrative modules** consist of a mixture of sectional markup, inclusion references, and narrative texts that provide transitions (narrative glue) between the other objects. Graphs of narrative modules whose edges are the inclusion references constitute the content representations of document fragments, documents, and document collections in the Planetary system, which generates active documents from them. It uses a process of separate compilation and dynamic linking to equip them with document (collection)-level features like content tables, indexes, section numbering and inter-module cross-references; see [Dav+11] for details.

As the active documents use identifiers that are relative to the base URI of the Planetary instance, whereas the content commons uses MMT URIs, the **semantic publishing map** which maintains the correspondence between these is a central, persistent data structure maintained by the Planetary system.



embedded) and **content** (from which the definition is fetched to be presented by the service) incarnations as well.

In the future we even want to combine this with the **source** dimension by combining it with a `\symdef`-look service that makes editing easier. This can be thought of as a presentation-

This is one instance of the **relational** dimension, another is used in the For instance, the RDFa embedded in the presentation of a formula (and represented in the linking part of the math archive) can be used for definition lookup as shown on the left. Actually the realization of the definition lookup service involves **presentation** (where the service is embedded) and **content** (from which the definition is fetched to be presented by the service) incarnations as well.

```
\begin{omgroup}[id=sec.conf funcs]{Continuous Functions}
\begin{module}[id=continuous]
\importmodule{./background/functions}{functions}
\importmodule{./background/reals}{reals}[false]
\syndef[continuous functions][2]{mathcal{C}^0(\#1,\#2)}
\abbrev[ContRR][2]{continuous functions:RealNumbers:RealNumbers}
\begin{definition}[for=continuous functions]
A function $ \text{fun} \quad \text{RealNumbers} \quad \text{A function } f:A \rightarrow B \text{ is a left-total, right-unique} \\
\end{definition}
\end{module}
\end{omgroup}
```

triggered complement to the editor-based service on the right that looks up \symdefs by their definienda.

## 6 Conclusion

We have presented an infrastructure for creating, storing, managing, and distributing mathematical knowledge in various pragmatic forms. We have identified five aspects that have to be taken into account here: *(i)* human-oriented *source languages* for efficient editing of content, *(ii)* the modular *content representation* that has been the focus of attention in MKM so far, *(iii)* *active presentations* of the content for viewing, navigation, and interaction, *(iv)* *narrative structures* that allow binding the content modules into self-contained documents that can be read linearly, and *(v)* *relational structures* that cross-link all these aspects and permit keeping them in sync.

We have developed and tested this infrastructure on the LATIN logic atlas, a highly modular graph of logics and logic morphisms using Twelf as the source language. Our focus was on the authoring perspective using our semantic IDE. The other experiment that informed the development was the Planetary system, a semantic publishing system that we use for our lecture notes with STEX as a surface language and provides an infrastructure for assembling content modules into document and collection structures.

In the future, we want to combine these systems and perspectives more tightly. For example, we could use Planetary to discuss and review logic formalizations in Twelf, or write papers about the formalizations in STEX. This should not pose any fundamental problems as the surface languages are interoperable by virtue of having the same, very general data model: the OMDoc ontology. By the same token we want to add additional surface languages and presentation targets that allow to include other user groups. High-profile examples include the Mizar Mathematical Language and Isabelle/ISAR.

Finally, there is a sixth aspect that may be added to the math archive infrastructure: discussions. The Planetary system already allows localized discussions on the content modules/presentations, which form an important part of the system content. These would probably be worth saving in math archives.

## References

- [Aus+03] R. Ausbrooks et al. *Mathematical Markup Language (MathML) Version 2.0 (second edition)*. Tech. rep. See <http://www.w3.org/TR/MathML2>. World Wide Web Consortium, 2003.
- [Dav+10] Catalin David et al. “eMath 3.0: Building Blocks for a social and semantic Web for online mathematics & ELearning”. In: *1<sup>st</sup> International Workshop on Mathematics and ICT: Education, Research and Applications*. (Bucharest, Romania, Nov. 3, 2010). Ed. by Ion Mierlus-Mazilu. 2010. URL: <http://kwarc.info/kohlhase/papers/malog10.pdf>.

- [Dav+11] Catalin David et al. “A Framework for Modular Semantic Publishing with Separate Compilation and Dynamic Linking”. 2011. URL: <https://svn.mathweb.org/repos/planetary/doc/sepublica11/paper.pdf>.
- [GLR09] J. Gićeva, C. Lange, and F. Rabe. “Integrating Web Services into Active Mathematical Documents”. In: *Intelligent Computer Mathematics*. Ed. by J. Carette and L. Dixon and C. Sacerdoti Coen and S. Watt. Vol. 5625. Lecture Notes in Computer Science. Springer, 2009, pp. 279–293.
- [GSK11] Deyan Ginev, Heinrich Stamerjohanns, and Michael Kohlhase. “The LATEXML Daemon: A LATEX Entrance to the Semantic Web”. submitted. 2011. URL: <https://kwarc.eecs.iu-bremen.de/repos/arXMLiv/doc/cicm-systems11/paper.pdf>.
- [HHP93] R. Harper, F. Honsell, and G. Plotkin. “A framework for defining logics”. In: *Journal of the Association for Computing Machinery* 40.1 (1993), pp. 143–184.
- [JK10] Constantin Jucovschi and Michael Kohlhase. “sTeXIDE: An Integrated Development Environment for sTeX Collections”. In: *Intelligent Computer Mathematics*. Ed. by Serge Autexier et al. LNAI 6167. Springer Verlag, 2010, pp. 336–344. arXiv:1005.5489v1 [cs.OH].
- [KMR08] Michael Kohlhase, Christine Müller, and Florian Rabe. “Notations for Living Mathematical Documents”. In: *Intelligent Computer Mathematics*. 9<sup>th</sup> International Conference, AISC, 15<sup>th</sup> Symposium, Calculemus, 7<sup>th</sup> International Conference MKM (Birmingham, UK, July 28–Aug. 1, 2008). Ed. by Serge Autexier et al. LNAI 5144. Springer Verlag, 2008, pp. 504–519. URL: <http://omdoc.org/pubs/mkm08-notations.pdf>.
- [KMR09] M. Kohlhase, T. Mossakowski, and F. Rabe. *The LATIN Project*. See <https://trac.omdoc.org/LATIN/>. 2009.
- [Koh] [Koh] General Computer Science: GenCS I/II Lecture Notes. <http://gencs.kwarc.info/book/1>. Semantic Course Notes in Panta Rei. 2011. URL: <http://gencs.kwarc.info/book/1>.
- [Koh08] Michael Kohlhase. “Using LATEX as a Semantic Markup Format”. In: *Mathematics in Computer Science* 2.2 (2008), pp. 279–304. URL: <https://svn.kwarc.info/repos/stex/doc/mcs08/stex.pdf>.
- [Koh+11] Michael Kohlhase et al. “The Planetary System: Web 3.0 & Active Documents for STEM”. In: accepted for publication at ICCS 2011 (Finalist at the Executable Papers Challenge). 2011. URL: <https://svn.mathweb.org/repos/planetary/doc/epc11/paper.pdf>.
- [KRZ10] M. Kohlhase, F. Rabe, and V. Zholudev. “Towards MKM in the Large: Modular Representation and Scalable Software Architecture”. In: *Intelligent Computer Mathematics*. Ed. by S. Autexier et al. Vol. 6167. Lecture Notes in Computer Science. Springer, 2010, pp. 370–384.

- [Lan11] Christoph Lange. “Enabling Collaboration on Semiformal Mathematical Knowledge by Semantic Web Integration”. submitted January 31, defended March 13. PhD thesis. Jacobs University Bremen, 2011. URL: <https://svn.kwarc.info/repos/swim/doc/phd/phd.pdf>.
- [Mil] Bruce Miller. *LaTeXML: A L<sup>A</sup>T<sub>E</sub>X to XML Converter*. URL: <http://dlmf.nist.gov/LaTeXML/> (visited on 03/03/2011).
- [Mü10] Christine Müller. “Adaptation of Mathematical Documents”. PhD thesis. Jacobs University Bremen, 2010. URL: <http://kwarc.info/cmueller/papers/thesis.pdf>.
- [Nor05] U. Norell. *The Agda WiKi*. <http://wiki.portal.chalmers.se/agda>. 2005.
- [Ora] Oracle. *JDK 6 Java Archive (JAR)*. <http://download.oracle.com/javase/6/docs/technotes/guides/jar>.
- [Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*. Vol. 828. Lecture Notes in Computer Science. Springer, 1994.
- [Pla] *Planetary Developer Forum*. URL: <http://trac.mathweb.org/planetary/> (visited on 01/20/2011).
- [PS99] F. Pfenning and C. Schürmann. “System Description: Twelf - A Meta-Logical Framework for Deductive Systems”. In: *Lecture Notes in Computer Science* 1632 (1999), pp. 202–206.
- [RK10] F. Rabe and M. Kohlhase. “A Scalable Module System”. To be submitted, see <http://kwarc.info/frabe/Research/mmt.pdf>. 2010.
- [RS09] F. Rabe and C. Schürmann. “A Practical Module System for LF”. In: *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*. Ed. by J. Cheney and A. Felty. ACM Press, 2009, pp. 40–48.
- [ZK09] Vyacheslav Zholudev and Michael Kohlhase. “TNTBase: a Versioned Storage for XML”. In: *Proceedings of Balisage: The Markup Conference*. Vol. 3. Balisage Series on Markup Technologies. Mulberry Technologies, Inc., 2009. DOI: [10.4242/BalisageVol3.Zholudev01](https://doi.org/10.4242/BalisageVol3.Zholudev01).

# Management of Change in Declarative Languages

Mihnea Iancu and Florian Rabe

Jacobs University, Bremen, Germany

**Abstract.** Due to the high degree of interconnectedness of formal mathematical statements and theories, human authors often have difficulties anticipating and tracking the effects of a change in large bodies of symbolic mathematical knowledge. Therefore, the automation of change management is often desirable. But while computers can in principle detect and propagate changes automatically, this process must take the semantics of the underlying mathematical formalism into account. Therefore, concrete management of change solutions are difficult to realize. The MMT language was designed as a generic declarative language that captures universal structural features while avoiding a commitment to a particular formalism. Therefore, it provides a promising framework for the systematic study of changes in declarative languages. We leverage this framework by providing a generic change management solution at the MMT level, which can be instantiated for arbitrary specific languages.

## 1 Introduction

Mathematical knowledge is growing at an enormous rate. Even if we restrict attention to formalized mathematics, libraries are reaching sizes that users have difficulties overseeing. Since this knowledge is also highly interconnected, it is getting increasingly difficult for humans to anticipate and follow the effects of changes. Therefore, management of change (MoC) for mathematics has received attention recently.

In this paper, we focus on change management for formalized mathematics, which — contrary to traditional, semi-formal mathematics — permits mechanically computing and verifying declarations. In principle, this should permit change management tools to automatically identify and recheck those declarations that are affected by a change. However, current computer algebra and deduction systems have not been designed systematically with change management in mind. In fact, the question of how to do that is still open.

A major motivation of our work was to provide change management for the LATIN library [CHK<sup>+</sup>11], a collection of formalizations of logics and related languages in a logical framework. Using the Little Theories approach [FGT92], the LATIN library takes the form of a highly modular and inter-connected network of theories, which creates an urgent need for change management.

We contribute to the solution of this problem by studying change management for the MMT language [RK11]. Because it was introduced as a foundation-

independent, modular, and scalable representation language for formal mathematical knowledge, it is a very promising framework for change management. Firstly, **foundation-independence** means that MMT avoids a syntactic or semantic commitment to any particular formalism. Thus, an MMT-based change management system could be applied to virtually any formal system. Secondly, **modularity** is a well-known strategy to rein in the impacts of changes and has been the basis of successful change management solutions such as [AHMS02]. Thirdly, MMT deemphasizes sequential in-memory processing of declarations in favor of maintaining a **large scale** network of declarations that are retrieved on demand, a crucial prerequisite for revisiting exactly the affected declarations.

We introduce a formal notion of differences between MMT documents, an abstract notion of semantic dependency relation, and a change propagation algorithm that guarantees that validity is preserved. We state our results for a small fragment of MMT, but our treatment extends to the full language. Our solution is implemented within the MMT system [Rab08], thus providing a generic change management system for formal mathematical languages.

In Sect. 2, we briefly introduce the MMT language in order to be self-contained. In Sect. 3, we refine our problem statement and compare it to related work. Then we develop the theory of change management in MMT in Sect. 4 and give an overview of our implementation in Sect. 5.

## 2 The MMT Language

|                        |               |  |
|------------------------|---------------|--|
| Theory Graph           | $\mathcal{G}$ | $::= \cdot \mid \mathcal{G}, Mod$  |
| Module Declaration     | $Mod$         | $::= T = \{Sym^*\} \mid v : T \rightarrow T = \{Ass^*\}$                           |
| Symbol Declaration     | $Sym$         | $::= c : \omega = \omega$  |
| Assignment Declaration | $Ass$         | $::= c := \omega$  |
| Term                   | $\omega$      | $::= \perp \mid T?c \mid x \mid \omega\omega^+ \mid \omega X.\omega \mid \omega^v$ |
| Variable Context       | $X$           | $::= \cdot \mid X, x : \omega = \omega$  |
| Module Identifier      | $M$           | $::= T \mid v$   |
| Theory Identifier      | $T$           | $::= \text{MMT URI}$   |
| Morphism Identifier    | $v$           | $::= \text{MMT URI}$   |
| Local Declaration Name | $c$           | $::= \text{MMT Name}$  |

**Fig. 1.** Simplified MMT Grammar

We will only give a brief overview of MMT and refer to [RK11] for details. The fragment of the MMT grammar that we discuss in this paper is given in Fig. 1. In particular, we have to omit the MMT module system for simplicity. The central notion is that of a **theory graph**, a list of modules, which are theories  $T$  or theory morphisms  $v$ .

A **theory** declaration  $T = \{Sym^*\}$  introduces a theory with name  $T$  containing a list of symbol declarations. A **symbol** declaration  $c : \omega = \omega'$  introduces

a symbol named  $c$  with **type**  $\omega$  and **definiens**  $\omega'$ . Both type and definiens are optional. However, in order to reduce the number of case distinctions, we use the special term  $\perp$ : If the type or definiens is omitted, we assume they are  $\perp$ .

**Terms**  $\omega$  over a theory  $T$  are formed from constants  $T?c$  declared in  $T$ , bound variables  $x$ , application  $\omega\omega_1\dots\omega_n$  of a function  $\omega$  to a sequence of arguments, bindings  $\omega X.\omega'$  using a binder  $\omega$ , a bound variable context  $X$ , and a scope  $\omega'$ , and morphism application  $\omega^v$ . Except for morphism application, this is a fragment of the OPENMATH language [BCC<sup>+</sup>04], which can express virtually every object.

**Theory morphism** declarations  $v : T \rightarrow T' = \{\text{Ass}^*\}$  introduce a morphism with name  $v$  from  $T$  to  $T'$  containing a list of assignment declarations. Such a morphism must contain exactly one assignment  $c := \omega'$  for each undefined symbol  $c : \omega = \perp$  in  $T$ ; here  $\omega'$  is some term over  $T'$ . Theory morphisms extend homomorphically to a mapping of  $T$ -terms to  $T'$  terms.

Intuitively, a theory morphism formalizes a translation between two formal languages. For example, the inclusion from the theory of semigroups to the theory of monoids (which extends the former with two declarations for the unit element and the neutrality axiom) can be formalized as a theory morphism. More complex examples are the Gödel-Gentzen negative translation from classical to intuitionistic logic or the interpretation of higher-order logic in set theory.

Every MMT declaration is identified by a canonical, globally unique URI. In particular, the URIs of symbol and assignment declarations are of the form  $T?c$  and  $v?c$ .

MMT symbol declarations subsume most semantically relevant statements in declarative mathematical languages including function and predicate symbols, type and universe symbols, and — using the Curry-Howard correspondence — axioms, theorems, and inference rules. Their syntax and semantics is determined by the foundation, in which MMT is parametric. In particular, the validity of a theory graph is defined relative to a type system provided by the **foundation**:

**Definition 1.** A **foundation** provides for every theory graph  $\mathcal{G}$  a binary relation on terms that is preserved under morphism application. This relation is denoted by  $\mathcal{G} \vdash \omega : \omega'$ , i.e., we have  $\mathcal{G} \vdash \omega : \omega'$  implies  $\mathcal{G} \vdash \omega^v : \omega'^v$ .

Constant declarations  $c : \omega = \omega'$  in a theory graph  $\mathcal{G}$  are valid if  $\mathcal{G} \vdash \omega' : \omega$ . Thus, a foundation also has to define typing for the special term  $\perp$ : The judgment  $\mathcal{G} \vdash \perp : \omega$  is interpreted as “ $\omega$  is a well-typed universe, i.e., it is legal to declare constants with type  $\omega$ ”. Similarly,  $\mathcal{G} \vdash \omega : \perp$  means that  $\omega$  may occur as the definiens of an untyped constant. This way the foundation can precisely control what symbol declarations are well-formed. Similarly, an assignment  $c := \omega$  in a morphism  $v$  is valid if  $\mathcal{G} \vdash \omega : \omega'^v$  where  $\omega'$  is the type of  $c$  in the domain of  $v$ .

**Running Example 1** Below we present a simple MMT theory for propositional logic over two revisions  $\text{Rev}_1$  and  $\text{Rev}_2$ . For simplicity, we will assume that the MMT module system is used and that the symbols **type**,  $\rightarrow$ , and  $\lambda$  have been imported from a theory representing the logical framework LF, and that all theory graphs are validated relative to a fixed foundation for LF. PL of  $\text{Rev}_1$  introduces

a type *bool* of formulas and three binary connectives, the last of which is defined in terms of the other two. This theory is valid. In *Rev*<sub>2</sub>, *bool* is renamed to *form*,  $\vee$  is deleted, and  $\neg$  is added. The other declarations remain unchanged, thus making the theory invalid.

| <i>Rev</i> <sub>1</sub>  | <i>Rev</i> <sub>2</sub>  |
|--|--|
| $PL = \{$ $\begin{aligned} \textit{bool} : \text{type} &= \perp \\ \vee : \textit{bool} \rightarrow \textit{bool} \rightarrow \textit{bool} &= \perp \\ \wedge : \textit{bool} \rightarrow \textit{bool} \rightarrow \textit{bool} &= \perp \\ \Rightarrow : \textit{bool} \rightarrow \textit{bool} \rightarrow \textit{bool} \\ &= \lambda x. \lambda y. y \vee (x \wedge y) \end{aligned}$ $\}$ | $PL = \{$ $\begin{aligned} \textit{form} : \text{type} &= \perp \\ \neg : \textit{form} \rightarrow \textit{form} &= \perp \\ \wedge : \textit{bool} \rightarrow \textit{bool} \rightarrow \textit{bool} &= \perp \\ \Rightarrow : \textit{bool} \rightarrow \textit{bool} \rightarrow \textit{bool} \\ &= \lambda x. \lambda y. y \vee (x \wedge y) \end{aligned}$ $\}$ |

### 3 Related Work

MoC has been applied successfully in a number of **domains** such as software engineering (e.g., [EG89]) or file systems ([Apa00, CVS, Git]). A typical MoC work flow in this setting uses *compilation units*, e.g., the classes of a Java program: These are compiled independently, and a compilation manager can record the dependency relation between the units. In particular, if compilation units correspond to source files, changes in a file can be managed by recompiling all depending source files.

Intuitively, this work flow can be applied to declarative languages for mathematics as well if we replace “compilation” with “validation” where the latter includes, e.g., type reconstruction, rewriting, and theorem proving. However, there are a few key differences. Firstly, the validation units are individual types and definitions (which includes assertions and proofs in MMT), of which there are many per source file (around 50 on average in the Mizar library [TB85]). Their validation can be expensive, and there may be many dependencies within the same theory and many little theories in the same source file. Therefore, validation units cannot be mapped to files so that the notions of change and dependency must consider fragments of source files. Moreover, since foundations may employ search with backtracking, the validation of a unit *U* may access more units than the validity of *U* depends on. Therefore, the dependency relation should not be recorded by a generic MMT validation manager but produced by the foundation. Recently several systems have become able to produce such dependency relations, in particular Coq and Mizar [AMU11].

MoC systems for mathematical languages can be classified according to the **nature** of changes. In principle, any change in a declarative language can be expressed as a sequence of *add* and *delete* operations on declarations. But using additional change natures is important for scalability. We use *updates* to change the type or definiens of a declaration without changing its MMT URI, and *renames* to change only the MMT URI. We do not use *reordering* operations because MMT already guarantees that the order of declarations has no effect on

the semantics. More complex natures have been studied in [BC08], which uses *splits* in ontologies to replace one concept with two new ones. Dually, we could consider *merge* changes, which identify two declarations.

Moreover, MoC systems can be classified by the **abstraction level** of their document model. The most concrete physical and bit level are relatively boring, and standard MoC tools operate at the character level treating documents as arrays of lines [Apa00, CVS, Git]. More abstract document models such as XML are better suited for mathematical content [AM10, Wag10] and have been applied to document formats for mathematics [Wag10, ADD<sup>+</sup>11]. Our work continues this development to more abstract document models by using MMT, which specifically models mathematical data structures. For example, the order of declarations, the flattening of imports, and the resolution of relative identifiers are opaque in XML but transparent in MMT representations. Moreover, MMT URIs are more suitable to identify the validation units than the XPath-based URLs usually employed in generic XML-based change models.

The development graph model [AHMS99], which has been applied to change management in [Hut00, AHMS02], is very similar to MMT: Both are parametric in the underlying formal language, and both make the modular structure of mathematical theories explicit. The main difference is that MMT uses a concrete (but still generic) declarative language for mathematical theories; modular structure is represented using special declarations. Somewhat dually, development graphs use an abstract category of theories using diagrams to represent modular structure; the declarations within a theory can be represented by refining the abstract model as done in [AHM10].

At an even more abstract level, document models can be specific to one foundational language. While foundation-independent approaches like ours can only identify potentially affected validation units, those could determine and possibly repair the impact of a change. That would permit treating even subobjects as validation units. However, presently no systems exists that can provide such foundation-specific information so that such MoC systems remain future work.

Finally we can classify systems based on how they **propagate** changes. Our approach focuses on the theoretical aspect of identifying the (potentially) affected parts. The most natural post-processing steps are to revalidate them, as, e.g., in [AHMS02], or to present them for user interaction as in [ADD<sup>+</sup>11]. The MMT system can be easily adapted for either one. A very different treatment is advocated in [KK11] based on using only references that include revision numbers so that changes never affect other declarations (because each change generates a new revision number).

## 4 A Theory of Changes

### 4.1 A Data Structure for Changes

Just like we can consider only an exemplary fragment of MMT here, we can only consider some of the possible changes. We will only treat changes of declarations

within modules. This is justified because these occur most frequently. However, our treatment can be generalized to changes of any declaration in the full MMT language. The grammar for our formal **language of changes** is given in Fig. 2.

|           |          |  |
|-----------|----------|--|
| Diff      | $\Delta$ | $::= \cdot \mid \Delta \bullet \delta$   |
| Change    | $\delta$ | $::= \mathcal{A}(M, c : \omega = \omega) \mid \mathcal{D}(M, c : \omega = \omega) \mid \mathcal{U}(M, c, o, \omega, \omega') \mid \mathcal{R}(T, c, c')$ |
| Component | $o$      | $::= \text{def} \mid \text{tp}$  |
| Box Terms | $\omega$ | $::= [\omega] \mid [\cdot]$ in addition to existing productions for $\omega$   |

**Fig. 2.** The Grammar for MMT Changes

We use terms as validation units because they are the smallest units that can be validated separately by foundations. Therefore, besides adding and deleting whole declarations, we use updates that change a term. In updates, we use **components**  $o$  to distinguish between changes to the type ( $o = \text{tp}$ ) or the definiens ( $o = \text{def}$ ). More precisely:

- $\mathcal{A}(M, c : \omega = \omega')$  adds a declaration to the module  $M$
- $\mathcal{D}(M, c : \omega = \omega')$  deletes a declaration from the module  $M$ .
- $\mathcal{U}(M, c, o, \omega, \omega')$  updates component  $o$  of declaration  $M?c$  from  $\omega$  to  $\omega'$ .
- $\mathcal{R}(T, c, c')$  renames the declaration  $c$  in theory  $T$  to  $c'$ .

Finally, **Diffs**  $\Delta$  are sequences of changes. In our implementation, we locate changes even more precisely by referring to subobjects of type and definiens. This is important for user interaction: If an impact has been detected, this permits showing the user exactly what change caused the impact.

*Notation 1.* In order to unify the cases of changing symbols in a theory and assignments in a morphism, we use the following convention: A declaration  $c := \omega'$  in a morphism  $v$  abbreviates a declaration  $c : \omega = \omega'$  and the components **tp** and **def** are defined accordingly. The type  $\omega$  is uniquely determined by MMT to ensure the type preservation of morphisms: Its value is  $\tau^v$  where  $\tau$  is the type of  $c$  in the domain of  $v$ . Updates to assignments work in the same way as updates to symbols except that the component **tp** cannot be changed.

We need one additional detail in our grammar: We add two special productions for terms  $\omega$ , which we call **box terms**. These represent invalid terms that are introduced during change propagation.

$[\cdot]$  represents a missing term.  $[\omega]$  represents a possibly invalid term  $\omega$ . More sophisticated box terms can also record the required type, which gives users a hint what change is needed and permits applications to type-check a declaration relative to the box terms in it. We omit this here for simplicity.

**Algebraically**, the set of diffs  $\Delta$  is the free monoid generated from changes  $\delta$ . As we will see below, the operation of applying a diff to a theory graph can be regarded as this monoid acting on the set of theory graphs.

As seen on the right, our diffs are **invertible**. This permits transactions where partially applied diffs are rolled back if they cause an error. This is also useful to offer undo-redo functionality in a user interface.

In order to talk efficiently about MMT theory graphs, we introduce a few definitions that permit looking up information in the theory graph:

**Definition 2 (Lookup in Theory Graphs).** *For a theory graph  $\mathcal{G}$ , we write*

- $\vdash \mathcal{G}(M) = Mod$  if a module declaration  $Mod$  with URI  $M$  is present in  $\mathcal{G}$ .
- $\mathcal{G} \vdash T?c : \omega = \omega'$  if  $T$  is a theory URI in  $\mathcal{G}$  and the symbol declaration  $c : \omega = \omega'$  exists in the body of  $T$ . We also define the corresponding notation for morphisms.
- $\vdash \mathcal{G}(M?c) = Sym$  if  $\vdash \mathcal{G}(M) = Mod$  and  $Sym$  is the declaration with name  $c$  in the body of  $Mod$ .
- $\vdash \mathcal{G}(M?c/o) = \omega$  if  $\vdash \mathcal{G}(M?c) = Sym$  and  $\omega$  is the component  $o$  of  $Sym$ .
- $\mathcal{G} \vdash \pi$  if  $\vdash \mathcal{G}(\pi) = Dec$  for some module or symbol declaration  $Dec$ .

We will now define the **application** of diffs  $\Delta$  on theory graphs  $\mathcal{G}$ , which we denote by  $\mathcal{G} \ll \Delta$ . In MoC tools, this is sometimes called *patching*.

**Definition 3.** *A diff  $\Delta$  is called **applicable** to the theory graph  $\mathcal{G}$  if  $\mathcal{G} \vdash \Delta$  according to the rules in Fig. 3.*

|   |   |
|---|---|
| $\frac{\mathcal{G} \vdash M \quad \mathcal{G} \not\vdash M?c}{\mathcal{G} \vdash \mathcal{A}(M, c : \omega = \omega')} \mathcal{A}_{dec}$ | $\frac{\mathcal{G} \vdash M?c : \omega = \omega'}{\mathcal{G} \vdash \mathcal{D}(M, c : \omega = \omega')} \mathcal{D}_{dec}$                 |
| $\frac{\vdash \mathcal{G}(T?c/o) = \omega}{\mathcal{G} \vdash \mathcal{U}(T, c, o, \omega, \omega')} \mathcal{U}_{sym}$                   | $\frac{\vdash \mathcal{G}(v?c/\mathbf{def}) = \omega}{\mathcal{G} \vdash \mathcal{U}(v, c, \mathbf{def}, \omega, \omega')} \mathcal{U}_{ass}$ |
| $\frac{\mathcal{G} \vdash T?c \quad \mathcal{G} \not\vdash T?c'}{\mathcal{G} \vdash \mathcal{R}(T, c, c')} \mathcal{R}_{dec}$             |   |
| $\frac{}{\mathcal{G} \vdash \Delta_{base}}$   | $\frac{\mathcal{G} \vdash \Delta \quad \mathcal{G} \ll \Delta \vdash \delta}{\mathcal{G} \vdash \Delta \bullet \delta} \Delta_{dec}$          |

**Fig. 3.** Applicability of Changes

**Definition 4 (Change Application).** Given a theory graph  $\mathcal{G}$  and a  $\mathcal{G}$ -applicable change  $\delta$ , we define  $\mathcal{G} \ll \delta$  as follows:

- If  $\delta = \mathcal{A}(M, c : \omega = \omega')$  then  $\mathcal{G} \ll \delta$  is the graph constructed from  $\mathcal{G}$  by adding the declaration  $c : \omega = \omega'$  to module  $M$ .
- If  $\delta = \mathcal{D}(M, c : \omega = \omega')$  then  $\mathcal{G} \ll \delta$  is the graph constructed from  $\mathcal{G}$  by deleting the declaration  $c : \omega = \omega'$  from module  $M$ .
- If  $\delta = \mathcal{U}(M, c, o, \omega, \omega')$  then  $\mathcal{G} \ll \delta$  is the graph constructed from  $\mathcal{G}$  by updating the component at  $M?c/o$  from  $\omega$  to  $\omega'$ .
- If  $\delta = \mathcal{R}(T, c, c')$  then  $\mathcal{G} \ll \delta$  is the graph constructed from  $\mathcal{G}$  by renaming the declaration at  $T?c$  to  $T?c'$ .

Moreover, we define  $\mathcal{G} \ll \Delta$  by  $\mathcal{G} \ll \cdot = \mathcal{G}$  and  $\mathcal{G} \ll (\Delta \bullet \delta) = (\mathcal{G} \ll \Delta) \ll \delta$ .

**Running Example 2 (Continuing Ex. 1)** We have  $Rev_1 \ll \Delta = Rev_2$  where  $\Delta$  is the diff:  $\mathcal{D}(PL, \text{bool} : \text{type} = \perp) \bullet \mathcal{A}(PL, \text{form} : \text{type} = \perp) \bullet \mathcal{D}(PL, \vee : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} = \perp) \bullet \mathcal{A}(PL, \neg : \text{form} \rightarrow \text{form} = \perp)$ . Alternatively, we could use a rename  $\mathcal{R}(PL, \text{bool}, \text{form})$  instead of the add-delete pair.

The following simple theorem permits lookups in a hypothetical patched theory graph. This is important for scalability in the typical case where a large  $\mathcal{G}$  should be neither changed nor copied:

**Theorem 1.** Assume a theory graph  $\mathcal{G}$  and a  $\mathcal{G}$ -applicable diff  $\Delta$ . Then

$$\begin{aligned} \vdash (\mathcal{G} \ll \cdot)(M?c/o) &= \mathcal{G}(M?c/o) \\ \vdash (\mathcal{G} \ll (\Delta \bullet \mathcal{A}(M, c : \omega = \omega')))(M?c/o) &= \begin{cases} \omega & \text{if } o = \text{tp} \\ \omega' & \text{if } o = \text{def} \end{cases} \\ \vdash (\mathcal{G} \ll (\Delta \bullet \mathcal{D}(M, c : \omega = \omega')))(M?c/o) &= \text{undefined} \\ \vdash (\mathcal{G} \ll (\Delta \bullet \mathcal{U}(M, c, o, \omega, \omega')))(M?c/o) &= \omega' \\ \vdash (\mathcal{G} \ll (\Delta \bullet \mathcal{R}(M, c', c)))(M?c/o) &= (\mathcal{G} \ll \Delta)(M?c'/o) \\ \vdash (\mathcal{G} \ll (\Delta \bullet \_))(M?c/o) &= (\mathcal{G} \ll \Delta)(M?c/o) \end{aligned}$$

where  $\_$  is any change not covered by the previous cases.

*Proof.* This is straightforward to prove using the definitions.

We will now introduce and study an equivalence relation between diffs. Intuitively, two diffs are equivalent if their application has the same effect:

**Definition 5.** Given a theory graph  $\mathcal{G}$ , two  $\mathcal{G}$ -applicable diffs  $\Delta$  and  $\Delta'$  are called  $\mathcal{G}$ -equivalent iff  $\mathcal{G} \ll \Delta = \mathcal{G} \ll \Delta'$ . We write this as  $\Delta \equiv^{\mathcal{G}} \Delta'$ .

Our main theorem about change application is that diffs can be normalized. We need some auxiliary definitions first:

**Definition 6.** The referenced URIs of a change are defined as follows: For both  $\mathcal{A}(M, c : \omega = \omega')$  and  $\mathcal{D}(M, c : \omega = \omega')$  they are  $M?c/\text{tp}$  and  $M?c/\text{def}$ , for  $\mathcal{U}(M, c, o, \omega, \omega')$  it is only  $M?c/o$ , and for  $\mathcal{R}(T, c, c')$  they are  $T?c/\text{tp}$ ,  $T?c/\text{def}$ ,

$T?c' / \text{tp}$  and  $T?c' / \text{def}$ . Two changes  $\delta$  and  $\delta'$  have a **clash** if they reference the same URI.

A diff  $\Delta$  is called **minimal** if there are no clashes between any two changes in  $\Delta$ . A minimal diff is called **normal** if it is of the form  $\Delta_1 \bullet \Delta_2$  where  $\Delta_1$  contains no renames and  $\Delta_2$  contains only renames.

**Theorem 2.** Reordering the changes in a minimal diff yields an equivalent diff.

*Proof.* In a minimal diff, each change affects a different declaration so the order of application is irrelevant.

**Definition 7.**  $\mathcal{G}' - \mathcal{G}$  is obtained as follows:

1. The diff  $\Delta$  contains the following changes (in any order):

$$\begin{array}{lll} \mathcal{U}(M, c, o, \omega, \omega') & \text{for} & \mathcal{G}(M?c/o) = \omega, \quad \mathcal{G}'(M?c/o) = \omega', \quad \omega \neq \omega' \\ \mathcal{D}(M?c : \omega = \omega') & \text{for} & \mathcal{G} \vdash M?c : \omega = \omega', \quad \mathcal{G}' \not\vdash M?c \\ \mathcal{A}(M?c : \omega = \omega') & \text{for} & \mathcal{G}' \vdash M?c : \omega = \omega', \quad \mathcal{G} \not\vdash M?c \end{array}$$

2. We say that a pair  $(A, D)$  of changes in  $\Delta$  matches if  $A = \mathcal{A}(T, c : \omega = \omega')$  and  $D = \mathcal{D}(T, c' : \omega = \omega')$ . They match uniquely if there is no other  $A'$  that matches  $D$  and no other  $D'$  that matches  $A$ .
3.  $\mathcal{G}' - \mathcal{G}$  arises from  $\Delta$  by removing every uniquely matching pair  $(A, D)$  and appending the respective rename  $\mathcal{R}(T, c, c')$ .

This definition first generates an add or delete for every URI that exists only in  $\mathcal{G}'$  or  $\mathcal{G}$ , respectively, and 0–2 updates for every URI that exists in both. Then uniquely matching add-delete pairs are replaced with renames. The uniqueness constraint is necessary to make the last step deterministic.

**Running Example 3 (Continuing Ex. 2)** The first step of the computation of the difference  $\text{Rev}_2 - \text{Rev}_1$  yields the diff from Ex. 2. The next steps simplify this diff to  $\mathcal{D}(PL, \vee : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} = \perp) \bullet \mathcal{A}(PL, \neg : \text{form} \rightarrow \text{form} = \perp) \bullet \mathcal{R}(PL, \text{bool}, \text{form})$ .

**Theorem 3.**  $\mathcal{G}' - \mathcal{G}$  is normal,  $\mathcal{G}$ -applicable, and  $\mathcal{G} \ll (\mathcal{G}' - \mathcal{G}) = \mathcal{G}'$ .

*Proof.* The proof is straightforward from the definition.

**Theorem 4.** If  $\mathcal{G}' = \mathcal{G} \ll \Delta$ , then there is a normal diff  $\Delta'$  such that  $\Delta \equiv^{\mathcal{G}} \Delta'$ .

*Proof.* We put  $\Delta' = (\mathcal{G} \ll \Delta) - \mathcal{G}$ . Then the result follows from Thm. 3.

## 4.2 A Data Structure for Dependencies

As our validation units are the components of MMT declarations, we need to formulate the validity of MMT theory graphs in a way that permits separate validation of each component:

**Definition 8.** A theory graph  $\mathcal{G}$  is called **foundationally valid** if for all symbol or assignment declarations  $\mathcal{G} \vdash M?c : \omega = \omega'$  (recall Not. 1), we have  $\mathcal{G} \vdash \omega' : \omega$ .

Now we can make formal statements how the validity of a theory graph is affected by changes. First, a typical property of typing relations is that they satisfy a weakening property: Additional information can not invalidate a type inference:

**Definition 9.** A foundation is called **monotonous** if the following rules are admissible for any  $A = \mathcal{A}(M, c : \_ \rightarrow \_)$  and for any  $U = \mathcal{U}(M, c, o, \perp, \_)$ :

$$\frac{\mathcal{G} \vdash \omega : \omega' \quad \mathcal{G} \vdash A}{\mathcal{G} \ll A \vdash \omega : \omega'} \quad \frac{\mathcal{G} \vdash \omega : \omega' \quad \mathcal{G} \vdash U}{\mathcal{G} \ll U \vdash \omega : \omega'}$$

Almost all practical foundations for MMT are monotonous. This includes even substructural type theories like linear LF [CP02] because we only require weakening for the set of global declarations, not for local contexts. A simple counter-example is a type theory with induction in which constructors can be added as individual declarations: Then adding a constructor will break an existing induction. But most type theories introduce all constructors in the same declaration.

While monotony permits handling additions to a theory graphs in general, we must introduce dependency relations between components to handle updates and deletes. Intuitively, if a validation unit  $U$  does not depend on  $U'$ , then deleting  $U'$  is guaranteed not to affect the validity of  $U$ :

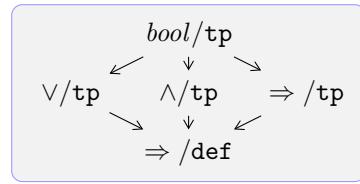
**Definition 10.** A *dependency relation* for a theory graph  $\mathcal{G}$  is a binary relation  $\rightarrowtail$  between declaration components  $M?c/o$  and  $M'?c'/o'$  such that the following rules are admissible:

$$\frac{\mathcal{G} \vdash M?c/o = \omega'' \quad \mathcal{G} \vdash M'?c' : \omega = \omega' \quad M?c/o \not\rightarrow M'?c' / \text{tp}}{\mathcal{G} \ll \mathcal{U}(M, c, o, \omega'', \perp) \vdash \perp : \omega}$$


---


$$\frac{\mathcal{G} \vdash M?c/o = \omega'' \quad \mathcal{G} \vdash M'?c' : \omega = \omega' \quad M?c/o \not\rightarrow M'?c' / \text{def}}{\mathcal{G} \ll \mathcal{U}(M, c, o, \omega'', \perp) \vdash \omega' : \omega}$$

Note that dependency relations are not necessarily transitive. That way changes can be propagated one dependency step at a time, and intermediate revalidation can show that no further propagation is necessary. Of course, the transitive closure (in fact: any larger relation) is again a dependency relation. Our definition of a dependency relation was inspired by the one in [RKS11].



**Running Example 4 (Continuing Ex. 3)** For the theory graph  $\text{Rev}_1$ , we obtain a dependency relation by assuming a dependency whenever a constant occurs in a component. We also assume a dependency from each definiens to its type. The graph in the figure above illustrates this relation.

### 4.3 Change Propagation

It is tempting to study the propagation of only a change  $\delta$ . But this does not cover the important case of transactions, where multiple changes are propagated together. This is typical in practice when an author makes multiple related changes. But it is very complicated to propagate an arbitrary diff. Our key insight is to focus on the **propagation of minimal diffs**. These are very easy to work with, and due to Thm. 3, this is not a loss of generality.

The central idea of our propagation algorithm is to introduce box terms that mark expressions as impacted. This has the advantage that propagation can be formalized as a closure operator on sets of changes so that no additional data structures for impacts are needed.

**Definition 11.** For a term  $\omega$  and a rename  $R = \mathcal{R}(T, c, c')$ , we define  $\omega^R$  as the term obtained from  $\omega$  by replacing all occurrences of  $T?c$  with  $T?c'$ . Similarly, if  $\Delta$  contains only renames, we define  $\omega^\Delta$  by  $\omega^\Delta \bullet R = (\omega^\Delta)^R$  and  $\omega^\cdot = \omega$ .

**Definition 12.** For the purposes of Def. 13, we say that a component  $M?c/o$  is **modified** by  $\Delta$  if  $\Delta$  contains a change of the form  $\mathcal{D}(M, c : \_ = \_)$  or  $\mathcal{U}(M, c, o, \omega, \_)$ .

The following definition and theorem express our main result. We state them for the special case for a diff that does not add or delete assignments. The general case holds as well but is more complicated.

**Definition 13 (Propagation).** Assume a fixed theory graph  $\mathcal{G}$  and a fixed dependency relation  $\nrightarrow$  (which we omit from the notation). Assume a  $\mathcal{G}$ -applicable diff in normal form  $\Delta = \Delta_1 \bullet \Delta_2$  that does not contain any adds or deletes of assignments. We define the propagation  $\overline{\Delta}$  of  $\Delta$  in multiple steps as follows:

1.  $\Delta'_1$  contains the following changes (in any order): whenever  $M?c/o \nrightarrow M'?c'/o'$  and  $M?c/o$  is modified by  $\Delta_1$ , the change

$$\mathcal{U}(M', c', o', \omega, \boxed{\omega}) \quad \text{for} \quad \mathcal{G} \ll \Delta \vdash M'?c'/o' = \omega$$

2.  $\Delta'_2$  contains the following changes (in any order): whenever  $\mathcal{R}(T, c, \_) \in \Delta_2$  and  $T?c/o \nrightarrow M'?c'/o'$ , the change

$$\mathcal{U}(M', c', o', \omega, \omega^{\Delta_2}) \quad \text{for} \quad \mathcal{G} \ll \Delta \bullet \Delta'_1 \vdash M'?c'/o' = \omega$$

3.  $\Delta'_3$  contains the following changes for every morphism  $\mathcal{G} \vdash v : T \rightarrow T'$  (in any order):

- whenever  $\mathcal{A}(T, c : \_ = \perp) \in \Delta$  or  $\mathcal{U}(T, c, \mathbf{def}, \_, \perp) \in \Delta$ , the change

$$\mathcal{A}(v, c := \boxed{\cdot})$$

- whenever  $\mathcal{D}(T, c : \_ = \perp) \in \Delta$  or  $\mathcal{U}(T, c, \mathbf{def}, \perp, \_) \in \Delta$ , the change

$$\mathcal{D}(v, Ass) \quad \text{for} \quad \mathcal{G} \ll \Delta \bullet \Delta'_1 \bullet \Delta'_2 \vdash v?c = Ass$$

- whenever  $\mathcal{R}(T, c, c') \in \Delta$  and  $\mathcal{G} \ll \Delta \bullet \Delta'_1 \bullet \Delta'_2 \vdash T?c/\text{def} = \perp$  and  $\mathcal{G} \ll \Delta \bullet \Delta'_1 \bullet \Delta'_2 \vdash v?c := \omega$ , the changes

$$\mathcal{D}(v, c := \omega), \mathcal{A}(v, c' := \omega)$$

4.  $\overline{\Delta}$  is obtained as  $\Delta'_1 \bullet \Delta'_2 \bullet \Delta'_3$ .

Intuitively,  $\Delta'_1$  updates all impacted terms to box terms. If  $\vdash$  is transitive, this includes all terms that depended on the now boxed terms.  $\Delta'_2$  updates all references to renamed declarations to the new name.

$\Delta'_3$  ensures that all morphisms have exactly one assignment for every undefined constant in the domain. The first two subcases add empty assignments or delete existing ones if necessary. The third subcase renames those assignments where the corresponding constant in the domain has been renamed.

**Theorem 5.** Consider the situation of Def. 13. Assume that the foundation is monotonous, that  $\mathcal{G}$  is foundationally valid, and that  $\vdash$  is transitive. Let  $\mathcal{G}' = \mathcal{G} \ll \Delta \bullet \overline{\Delta}$ , and let  $\mathcal{G}^*$  be a theory graph that arises from  $\mathcal{G}'$  by replacing every box term with a term that type checks in the sense of Def. 8. Then  $\mathcal{G}^*$  is foundationally valid.

*Proof.* Let us first consider the special case without renames or morphisms. We apply Def. 8 to  $\mathcal{G}^*$ . Due to  $\Delta'_1$  and the transitivity of  $\vdash$ , all possibly ill-typed terms have been replaced with box terms in  $\mathcal{G}'$ ; and according to the assumptions, these are replaced with well-typed terms in  $\mathcal{G}^*$ . Thus, the claim follows.

If there are renames, care must be taken to update all references to the renamed declarations. If there are adds, care must be taken to guarantee the totality of morphisms. Both conditions are already fulfilled in  $\mathcal{G}'$ . We omit the details.

A typical situation where we would apply Thm. 5 is after a user made the changes  $\Delta$ . Then propagation marks all terms that have to be revalidated and — if not well-typed — replaced interactively with well-typed terms. The theorem guarantees the resulting graph is valid again.

$$\begin{aligned} PL = \{ & \\ & \text{form : type} = \perp \\ & \neg : \text{form} \rightarrow \text{form} = \perp \\ & \wedge : \text{form} \rightarrow \text{form} \rightarrow \text{form} = \perp \\ & \Rightarrow : \text{form} \rightarrow \text{form} \rightarrow \text{form} \\ & \quad = \boxed{\lambda x. \lambda y. y \vee (x \wedge y)} \\ & \} \end{aligned}$$

### Running Example 5 (Continuing Ex. 3 and 4)

Using  $\Delta = \text{Rev}_2 - \text{Rev}_1$  and the dependency relation from Ex. 4, we compute  $\overline{\Delta}$ . First  $\Delta = \Delta_1 \bullet \Delta_2$  with  $\Delta_1 = \mathcal{D}(PL, \vee : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} = \perp) \bullet \mathcal{A}(PL, \neg : \text{form} \rightarrow \text{form} = \perp)$  and  $\Delta_2 = \mathcal{R}(PL, \text{bool}, \text{form})$ . Then

$$\overline{\Delta'_1} = \mathcal{U}(PL, \Rightarrow, \text{def}, \lambda x. \lambda y. y \vee (x \wedge y), \boxed{\lambda x. \lambda y. y \vee (x \wedge y)})$$

as well as  $\overline{\Delta'_2} = \mathcal{U}(PL, \wedge, \text{tp}, \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}, \text{form} \rightarrow \text{form} \rightarrow \text{form}) \bullet \mathcal{U}(PL, \Rightarrow, \text{tp}, \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}, \text{form} \rightarrow \text{form} \rightarrow \text{form})$  and  $\overline{\Delta'_3} = \dots$ . Finally, the theory graph  $\text{Rev}_1 \ll \Delta \bullet \overline{\Delta}$  is shown above. As stated in Thm. 5, it becomes foundationally valid after replacing the box term with a term of the right type.

## 5 A Generic Change Management API

*Implementation* We have implemented the data structures and algorithms from Sect. 4 as a part of the MMT API [Rab08]. In fact, our implementation covers a much larger fragment of MMT than discussed in this paper.

In particular, the API now contains functions that compute the **difference**  $\mathcal{G}' - \mathcal{G}$  of two theory graphs. The difference of two modules can be computed as well. The two arguments can either be provided directly or the previous revision can be pulled automatically from an SVN repository.

We also added functions for **change propagation** that enrich a normal diff with its direct impacts according to Def. 13. The generated box terms are represented as OPENMATH error objects. During the propagation algorithm, we make crucial use of Thm. 1 to increase the efficiency.

Both of these algorithms are implemented foundation-independently. The foundation is only needed to obtain the dependency relation and to revalidate the impacted declarations. Both are special cases of type checking.

The MMT API delegates a **type checking** obligation  $\omega' : \omega$  to a plugin for the respective foundation. In particular, there is a plugin for a monotonous foundation for the logical framework LF [HHP93], which induces implementations of type checking for all formal systems represented in LF (i.e., for a lot of formal systems [CHK<sup>+</sup>11]).

The plugin interface is such that the plugin calls back to the main system whenever it needs to look up any component  $M?c/o$ . In the simplest case, we can trace these callbacks to obtain the set of components  $Used(\omega', \omega)$  that were used to validate  $\omega' : \omega$ . When the system validates a theory graph  $\mathcal{G}$  according to Def. 8, we obtain a **dependency relation** by putting for every symbol or assignment  $\mathcal{G} \vdash M'?c' : \omega = \omega'$

$$\begin{aligned} M?c/o &\rightsquigarrow M'?c'/\text{tp} & \text{if } M?c/o \in Used(\perp, \omega) \\ M?c/o &\rightsquigarrow M'?c'/\text{def} & \text{if } M?c/o \in Used(\omega', \omega) \\ M'?c'/\text{tp} &\rightsquigarrow M'?c'/\text{def} \end{aligned}$$

Note that we first check the type of the declaration and then separately check the definiens against that type even though the latter implies the former. This is important because the type will usually have much less dependencies than the definiens.

This dependency relation is stored in the MMT ontology, which MMT maintains together with the content [HIJ<sup>+</sup>11]. Alternatively, the foundation can explicitly provide a dependency relation, or we can import dependency relations externally, e.g., the ones from [AMU11].

*Application* We have applied the resulting system to obtain a change management API for the LATIN library. Using the MMT plugins for LF — the language underlying the LATIN library — we obtain a foundation that validates the library and computes a dependency relation for it. Fig. 4 gives a summary of the

dependency relation, where we include only the about 1700 components falling into the fragment of MMT treated in this paper. The tables group the components by the number of components that they depend on (left) or that depend on them (right). This includes only direct dependencies — taking the transitive closure increases the numbers by about 20 %.

| dependencies | components (%) | impacts  | components (%) |
|--------------|----------------|----------|----------------|
| 0 – 5        | 1373 (79)      | 0 – 5    | 1504 (86.5)    |
| 6 – 10       | 271 (15.6)     | 6 – 10   | 101 (5.8)      |
| 11 – 15      | 81 (4.7)       | 11 – 25  | 76 (4.4)       |
| 16 – 26      | 13 (0.7)       | 26 – 50  | 31 (1.8)       |
|              |                | 50 – 449 | 26 (1.5)       |

**Fig. 4.** Components grouped by dependencies and impacts

The number of dependencies and impacts is generally low. This is a major benefit of our choice of using type and definiens as separate validation units, which avoids the exponential blowup one would otherwise expect. Indeed, on average a type has 3 times as many impacts as a definiens.

Our differencing algorithm can detect and propagate changes easily, and it is straightforward to revalidate the impacted components. The numbers show that even manual inspection (as opposed to automatic revalidation) is feasible in most cases: For example, changes to 86 % of the components impact only 5 or less components. Even if the number of impacted components is so small, it is usually very difficult for humans to identify exactly which components are impacted. Our MoC infrastructure, on the other hand, does not only identify them automatically but also guarantees that all other components stay valid.

## 6 Conclusion

We have presented a theory of change management based on the MMT language including difference, dependency, and impact analysis. As MMT is foundation-independent, our work yields a theory of change management for an arbitrary declarative language. Our work is implemented as a part of the MMT API and thus immediately applicable to any language that is represented in MMT. The latter includes in particular the logical framework LF and thus every language represented in it.

Because we use fine-grained dependencies, change propagation can identify individual type checking obligations (which subsume proof obligations) that have to be revalidated. The MMT API already provides a scalable framework for validating individual such obligations efficiently. Therefore, our work provides the foundation for a large scale change management system for declarative languages.

While our presentation has focused on a small fragment of MMT, the results can be generalized to the whole MMT language, in particular the module system.

Presently the most important missing feature is a connection between the MMT abstract syntax and the concrete syntax of individual languages. Therefore, change management currently requires an export into MMT’s abstract syntax (which exists for, e.g., Mizar [TB85], TPTP [SS98], and OWL [W3C09]). Consequently, **future work** will focus on developing fast bidirectional translations between human-friendly source languages and their MMT content representation. If these include fine-grained cross-references between source and content, MMT can propagate changes into the source language; this could happen even while the user is typing.

More generally, this approach extends to pure mathematics where the source language is, e.g., L<sup>A</sup>T<sub>E</sub>X. If the source is formalized manually, it is sufficient to include cross-references in the above sense. Then changes in the L<sup>A</sup>T<sub>E</sub>X source can be treated and propagated like changes in the formalization. Alternatively, we can avoid a manual formalization if certain annotations are present in the source: firstly, annotations that map a line number to the identifier of the statement (definition, theorem, etc.) made at that line; secondly, annotations that explicate the dependency relation between statements. For example, the sTeX package for L<sup>A</sup>T<sub>E</sub>X permits such annotations in a way that supports automated extraction.

## References

- ADD<sup>+</sup>11. S. Autexier, C. David, D. Dietrich, M. Kohlhase, and V. Zholudev. Workflows for the Management of Change in Science, Technologies, Engineering and Mathematics. In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 164–179. Springer, 2011.
- AHM10. S. Autexier, D. Hutter, and T. Mossakowski. Change Management for Heterogeneous Development Graphs. In S. Siegler and N. Wasser, editors, *Verification, Induction, Termination Analysis, Festschrift in honor of Christoph Walther*. Springer, 2010.
- AHMS99. S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an Evolutionary Formal Software-Development Using CASL. In D. Bert, C. Chopp, and P. Mosses, editors, *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1999.
- AHMS02. S. Autexier, D. Hutter, T. Mossakowski, and A. Schairer. The Development Graph Manager Maya (System Description). In H. Kirchner and C. Ringeissen, editors, *Algebraic Methods and Software Technology, 9th International Conference*, pages 495–502. Springer, 2002.
- AM10. S. Autexier and N. Müller. Semantics-Based Change Impact Analysis for Heterogeneous Collections of Documents. In M. Gormish and R. Ingold, editors, *Proceedings of 10th ACM Symposium on Document Engineering (DocEng2010)*, 2010.
- AMU11. Jesse Alama, Lionel Mamane, and Josef Urban. Dependencies in Formal Mathematics. *CoRR*, abs/1109.3687, 2011.
- Apa00. Apache Software Foundation. Apache Subversion, 2000. see <http://subversion.apache.org/>.
- BC08. A. Bundy and M. Chan. Towards Ontology Evolution in Physics. In W. Hodges and R. de Queiroz, editors, *Logic, Language, Information and Computation*, pages 98–110. Springer, 2008.

- BCC<sup>+</sup>04. S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See <http://www.openmath.org/standard/om20>.
- CHK<sup>+</sup>11. M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. Project Abstract: Logic Atlas and Integrator (LATIN). In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 287–289. Springer, 2011.
- CP02. I. Cervesato and F. Pfenning. A Linear Logical Framework. *Information and Computation*, 179(1):19–75, 2002.
- CVS. Concurrent Versions System: The open standard for Version Control. Web site at <http://cvs.nongnu.org/>. seen February 2012.
- EG89. C. Ellis and S. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 399–407. ACM, 1989.
- FGT92. W. Farmer, J. Guttman, and F. Thayer. Little Theories. In D. Kapur, editor, *Conference on Automated Deduction*, pages 467–581, 1992.
- Git. Git. Web Site at: <http://git-scm.com/>.
- HHP93. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
- HIJ<sup>+</sup>11. F. Horozal, A. Iacob, C. Jucovschi, M. Kohlhase, and F. Rabe. Combining Source, Content, Presentation, Narration, and Relational Representation. In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 211–226. Springer, 2011.
- Hut00. D. Hutter. Management of change in structured verification. In *Proceedings Automated Software Engineering, ASE-2000*, pages 23–34, 2000.
- KK11. Andrea Kohlhase and Michael Kohlhase. Versioned links. In *Proceedings of the 29<sup>th</sup> annual ACM international conference on Design of communication (SIGDOC)*, 2011.
- Rab08. F. Rabe. The MMT System, 2008. see <https://trac.kwarc.info/MMT/>.
- RK11. F. Rabe and M. Kohlhase. A Scalable Module System. see <http://arxiv.org/abs/1105.0548>, 2011.
- RKS11. F. Rabe, M. Kohlhase, and C. Sacerdoti Coen. A Foundational View on Integration Problems. In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 106–121. Springer, 2011.
- SS98. G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
- TB85. A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985.
- W3C09. W3C. OWL 2 Web Ontology Language, 2009. <http://www.w3.org/TR/owl-overview/>.
- Wag10. M. Wagner. *A change-oriented architecture for mathematical authoring assistance*. PhD thesis, Universität des Saarlands, 2010.

# A Query Language for Formal Mathematical Libraries

Florian Rabe

Jacobs University Bremen, Germany

**Abstract.** One of the most promising applications of mathematical knowledge management is search: Even if we restrict attention to the tiny fragment of mathematics that has been formalized, the amount exceeds the comprehension of an individual human.

Based on the generic representation language MMT, we introduce the mathematical query language QMT: It combines simplicity, expressivity, and scalability while avoiding a commitment to a particular logical formalism. QMT can integrate various search paradigms such as unification, semantic web, or XQuery style queries, and QMT queries can span different mathematical libraries.

We have implemented QMT as a part of the MMT API. This combination provides a scalable indexing and query engine that can be readily applied to any library of mathematical knowledge. While our focus here is on libraries that are available in a content markup language, QMT naturally extends to presentation and narration markup languages.

## 1 Introduction and Related Work

Mathematical knowledge management applications are particularly strong at large scales, where automation can be significantly superior to human intuition. This makes search and retrieval pivotal MKM applications: The more the amount of mathematical knowledge grows, the harder it becomes for users to find relevant information. Indeed, even expert users of individual libraries can have difficulties reusing an existing development because they are not aware of it. Therefore, this question has received much attention.

**Object query languages** augment standard text search with phrase queries that match mathematical operators and with wild cards that match arbitrary mathematical expressions. Abstractly, an object query engine is based on an index, which is a set of pairs  $(l, o)$  where  $o$  is an object and  $l$  is the location of  $o$ . The index is built from a collection of mathematical documents, and the result of an object query is a subset of the index. The object model is usually based on presentation MathML and/or content MathML/OpenMath [W3C03,BCC<sup>+</sup>04], but importers can be used to index other formats such as LaTeX. Examples for object query languages and engines are given in [MY03,MM06,MG08,K<sup>+</sup>06,SL11]. A partial overview can be found in [SL11]. A central question is the use of wild cards. An example language with complex wild cards is given in [AY08]. Most generally, [K<sup>+</sup>06] uses unification queries that return all objects that can be unified with the query.

**Property query languages** are similar to object query languages except that both the index and the query use relational information that abstracts from the mathematical objects. For example, the relational index might store the toplevel symbol of every object or the “used-in” relation between statements. This approximates an object index, and many property queries are special cases of object queries. But property queries are simpler and more efficient, and they still cover many important examples. Such languages are given in [GC03,AS04] and [BR03] based on the Coq and Mizar libraries, respectively.

**Compositional query languages** focus on a complex language of query expressions that are evaluated compositionally. The atomic queries are provided by the elements of the queried library. SQL [ANS03] uses  $n$ -ary relations between elements, and query expressions use the algebra of relations. The SPARQL [W3C08] data model is RDF, and queries focus on unary and binary predicates on a set of URIs of statements. This could serve as the basis for mathematics on the semantic web. Both data models match bibliographical meta-data and property-based indices and could also be applied to the results of object queries (seen as sets of pairs); but they are not well-suited for expressions. The XQuery [W3C07] data model is XML, and query expressions are centered around operations on lists of XML nodes. This is well-suited for XML-based markup languages for mathematical documents and expressions and was applied to OMDoc [Koh06] in [ZK09]. In [KRZ10], the latter was combined with property queries. Very recently [ADL12] gave a compositional query language for hiproof proof trees that integrates concepts from both object and property queries.

A number of **individual libraries** of mathematics provide custom query functionality. Object query languages are used, for example, in [LM06] for Activemath or in Wolfram|Alpha. Most interactive proof assistants permit some object or property queries, primarily to search for theorems that are applicable to a certain goal, e.g., Isabelle, Coq, and Matita. [Urb06] is notable for using automated reasoning to prepare an index of all Mizar theorems.

It is often desirable to combine several of the above formalisms in the same query. Therefore, we have designed the query language QMT with the goal of permitting as many different query paradigms as possible. QMT uses a simple kernel syntax in which many advanced query paradigms can be defined. This permits giving a formal syntax, a formal semantics, and a scalable implementation, all of which are presented in this paper.

QMT is grounded in the MMT language (Module System for Mathematical Theories) [RK11], a scalable, modular representation language for mathematical knowledge. It is designed as a scalable trade-off between (i) a logical framework with formal syntax and semantics and (ii) an MKM framework that does not commit to any particular formal system. Thus, MMT permits both adequate representations of virtually any formal system as well as the implementation of generic MKM services. We implement QMT on top of our MMT system, which provides a flexible and scalable query API and query server.

Our design has two pivotal strengths. Firstly, QMT can be applied to the libraries of any formal system that is represented as MMT. Queries can even span

| Declaration          | Intended Semantics                |
|----------------------|-----------------------------------|
| base type $a$        | a set of objects                  |
| concept symbol $c$   | a subset of a base type           |
| relation symbol $r$  | a relation between two base types |
| function symbol $f$  | a sorted first-order function     |
| predicate symbol $p$ | a sorted first-order predicate    |

| Kind of Expression            | Intended Semantics              |
|-------------------------------|---------------------------------|
| Type $T$                      | a set                           |
| Query $Q : T$                 | an element of $T$               |
| element query $Q : t$         | an element of $t$               |
| set query $Q : \text{set}(t)$ | a subset of $t$                 |
| Relation $R < a, a'$          | a relation between $a$ and $a'$ |
| Proposition $F$               | a boolean truth value           |

**Fig. 1.** QMT Notions and their Intuitions

libraries of different systems. Secondly, QMT queries can make use of other MMT services. For example, queries can access the inferred type and the presentation of a found expression, which are computed dynamically.

We split the definition of QMT into two parts. Firstly, Sect. 2 defines QMT signatures in general and then the syntax and semantics of QMT for an arbitrary signature. Secondly, Sect. 3 describes a specific QMT signature that we use for MMT libraries. Our implementation, which is based on that signature, is presented in Sect. 4.

## 2 The QMT Query Language

### 2.1 Syntax

Our syntax arises by combining features of sorted first-order logic – which leads to very intuitive expressions – and of description logics – which leads to efficient evaluations. Therefore, our **signatures**  $\Sigma$  contain five kinds of declarations as given in Fig. 1.

For a given signature, we define four kinds of **expressions**: types  $T$ , relations  $R$ , propositions  $F$ , and typed queries  $Q$  as listed in Fig. 1. The grammar for signatures and expressions is given in Fig. 2.

The intuitions for most expression formation operators can be guessed easily from their notations. In the following we will discuss each in more detail.

Regarding **types**  $T$ , we use product types and power type. However, we go out of our way to avoid arbitrary nestings of type constructors. Every type is either a product  $t = a_1 \times \dots \times a_n$  of base types  $a_i$  or the power type  $\text{set}(t)$  of such a type. Thus, we are able to use the two most important type formation operators in the context of querying: product types arise when a query contains multiple query variables, and power types arise when a query returns multiple

|               |   |
|---------------|---|
| Signatures    | $\Sigma ::= \cdot \mid \Sigma, a : type \mid \Sigma, c < a \mid \Sigma, r < a, a$<br>$\mid \Sigma, f : T, \dots, T \rightarrow T \mid \Sigma, p : T, \dots, T \rightarrow prop$ |
| Contexts      | $\Gamma ::= \cdot \mid \Gamma, x : T$   |
| Simple Types  | $t ::= a \times \dots \times a$   |
| General Types | $T ::= t \mid set(t)$   |
| Relations     | $R ::= r \mid R^{-1} \mid R^* \mid R; R \mid R \cup R \mid R \cap R \mid R \setminus R$   |
| Propositions  | $F ::= p(Q, \dots, Q) \mid \neg F \mid F \wedge F \mid \forall x \in Q. F(x)$   |
| Queries       | $Q ::= c \mid x \mid f(Q, \dots, Q) \mid Q^1 * \dots * Q^n \mid Q_i$<br>$\mid R(Q) \mid \bigcup_{x \in Q} Q'(x) \mid \{x \in Q \mid F(x)\}$                                     |

**Fig. 2.** The Grammar for Query Expressions

results. But at the same time, the type system remains very simple and can be treated as essentially first-order.

Regarding **relations**, we provide the common operations from the calculus of binary relations: dual/inverse  $R^{-1}$ , transitive closure  $R^*$ , composition  $R; R'$ , union  $R \cup R'$ , intersection  $R \cap R'$ , and difference  $R \setminus R'$ . Notably absent is the absolute complement operation  $R^c$ ; it is omitted because its semantics can not be computed efficiently in general. Note that the operation  $R^{-1}$  is only necessary for atomic  $R$ : For all other cases, we can put  $(R^*)^{-1} = (R^{-1})^*$ ,  $(R; R')^{-1} = R'^{-1}; R^{-1}$ , and  $(R * R')^{-1} = R^{-1} * R'^{-1}$  for  $* \in \{\cup, \cap, \setminus\}$ .

Regarding **propositions**, we use the usual constructors of classical first-order logic: predicates, negation, conjunction, and universal quantification. As usual, the other constructors are definable. However, there is one specialty: The quantification  $\forall x \in Q. F(x)$  does not quantify over a type  $t$ ; instead, it is relativized by a query result  $Q : set(t)$ . This specialty is meant to support efficient evaluation: The extension of a base type is usually much larger than that of a query, and it may not be efficiently computable or not even finite.

Regarding **queries**, our language combines intuitions from description and first-order logic with an intuitive mathematical notation. Constants  $c$ , variables  $x$ , and function application are as usual for sorted first-order logic.  $Q^1 * \dots * Q^n$  for  $n \in \mathbb{N}$  and  $Q_i$  for  $i = 1, \dots, n$  denote tupling and projection.  $R(Q)$  represents the image of the object given by  $Q$  under the relation given by  $R$ .  $\bigcup_{x \in Q} Q'(x)$  denotes the union of the family of queries  $Q'(x)$  where  $x$  runs over all objects in the result of  $Q$ . Finally,  $\{x \in Q \mid F(x)\}$  denotes comprehension on queries, i.e., the objects in  $Q$  that satisfy  $F$ . Just like for the universal quantification, all bound variables are relativized to a query result to support efficient evaluation.

*Remark 1.* While we do not present a systematic analysis of the efficiency of QMT, we point out that we designed the syntax of QMT with the goal of supporting efficient evaluation. In particular, this motivated our distinction between the ontology part, i.e., concept and relation symbols, and the first-order part, i.e., the function and predicate symbols.

Indeed, every concept  $c < t$  can be regarded as a function symbol  $c : set(t)$ , and every relation  $r < a, a'$  as a predicate symbol  $r : a, a' \rightarrow prop$ . Thus, the ontology symbols may appear redundant — their purpose is to permit efficient evaluations. This is most apparent for relations. For a predicate symbol  $p :$

|   |                         |  |
|---|-------------------------|--|
| $\frac{n \text{ not declared in } \Sigma}{n \notin \Sigma}$   | $\frac{}{\vdash \cdot}$ | $\frac{\vdash \Sigma \quad a \notin \Sigma}{\vdash \Sigma, a : type}$  |
| $\frac{\vdash \Sigma \quad c \notin \Sigma \quad a : type \text{ in } \Sigma}{\vdash \Sigma, c < a}$  |                         | $\frac{\vdash \Sigma \quad r \notin \Sigma \quad (a_i : type \text{ in } \Sigma)_{i=1}^2}{\vdash \Sigma, r < a_1, a_2}$                    |
| $\frac{\vdash \Sigma \quad f \notin \Sigma \quad (\vdash_\Sigma T_i : type)_{i=1}^{n+1}}{\vdash \Sigma, f : T_1, \dots, T_n \rightarrow T_{n+1}}$ |                         | $\frac{\vdash \Sigma \quad p \notin \Sigma \quad (\vdash_\Sigma T_i : type)_{i=1}^n}{\vdash \Sigma, p : T_1, \dots, T_n \rightarrow prop}$ |

**Fig. 3.** Well-Formed Signatures

$a, a' \rightarrow prop$ , evaluation requires a method that maps from  $\llbracket a \rrbracket \times \llbracket a' \rrbracket$  to booleans. But for a relation symbol  $r < a, a'$ , evaluation requires a method that returns for any  $u$  all  $v$  such that  $(u, v) \in \llbracket r \rrbracket$  or all  $v$  such that  $(v, u) \in \llbracket r \rrbracket$ . A corresponding property applies to concepts.

Therefore, efficient implementations of QMT should maintain indices for them that are computed a priori: hash sets for the concept symbols and hash tables for the relation symbols. (Note that using hash tables for all relation symbols permits fast evaluation of all relation expressions  $R$ , which is crucial for the evaluation of queries  $R(Q)$ .) The implementation of function and predicate symbols, on the other hand, only requires plain functions that are called when evaluating a query.

Thus, it is a design decision whether a certain feature is realized by an ontology or by a first-order symbol. By separating the ontology and the first-order part, we permit simple indices for the former and retain flexible extensibility for the latter (see also Rem. 2).

Based on these intuitions, it is straightforward to define the **well-formed expressions**, i.e., the expressions that will have a denotational semantics. More formally, we use the **judgments** given in Fig. 5

| Judgment                        | Intuition                                    |
|---------------------------------|--|
| $\vdash \Sigma$                 | well-formed signature $\Sigma$               |
| $\vdash_\Sigma T : type$        | well-formed type $T$                         |
| $\Gamma \vdash_\Sigma Q : T$    | well-typed query $Q$ of type $T$             |
| $\Gamma \vdash_\Sigma Q : T$    | well-typed query $Q$ of type $T$             |
| $\vdash_\Sigma R < a, a'$       | well-typed relation $R$ between $a$ and $a'$ |
| $\Gamma \vdash_\Sigma F : prop$ | well-formed proposition $F$                  |

**Fig. 5.** Judgments

to define the well-formed expressions over a signature  $\Sigma$  and a context  $\Gamma$ . The **rules** for these judgments are given in Fig. 3 and 4.

In order to give some meaningful examples, we will already make use of the symbols from the MMT signature, which we will introduce in Sect. 3.

|   |   |
|---|---|
| $\frac{(a_i : type \text{ in } \Sigma)_{i=1}^n}{\vdash_{\Sigma} a_1 \times \dots \times a_n : type}$  | $\frac{(a_i : type \text{ in } \Sigma)_{i=1}^n}{\vdash_{\Sigma} set(a_1 \times \dots \times a_n) : type}$   |
| $\frac{c < t \text{ in } \Sigma}{\Gamma \vdash_{\Sigma} c : set(t)}$  | $\frac{f : T_1, \dots, T_n \rightarrow T \text{ in } \Sigma \quad \Gamma \vdash_{\Sigma} Q_i : T_i}{\Gamma \vdash_{\Sigma} f(Q_1, \dots, Q_n) : T}$ |
| $\frac{\Gamma \vdash_{\Sigma} Q_i : t_i \text{ for } i = 1, \dots, n}{\Gamma \vdash_{\Sigma} Q_1 * \dots * Q_n : t_1 \times \dots \times t_n}$            | $\frac{\Gamma \vdash_{\Sigma} Q : t_1 \times \dots \times t_n \quad i \in \{1, \dots, n\}}{\Gamma \vdash_{\Sigma} Q_i : t_i}$                       |
| $\frac{\Gamma \vdash_{\Sigma} Q : set(t) \quad \Gamma, x : t \vdash_{\Sigma} Q'(x) : set(t')}{\Gamma \vdash_{\Sigma} \bigcup_{x \in Q} Q'(x) : set(t')}$  | $\frac{\Gamma \vdash_{\Sigma} Q : t \quad \vdash_{\Sigma} R < t, t'}{\Gamma \vdash_{\Sigma} R(Q) : set(t')}$  |
| $\frac{\Gamma \vdash_{\Sigma} Q : set(t) \quad \Gamma, x : t \vdash_{\Sigma} F(x) : prop}{\Gamma \vdash_{\Sigma} \{x \in Q   F(x)\} : set(t)}$            | $\frac{r < a, a' \text{ in } \Sigma}{\vdash_{\Sigma} r < a, a'}$  |
| $\frac{\vdash_{\Sigma} R < a, a'}{\vdash_{\Sigma} R^{-1} < a', a}$  | $\frac{\vdash_{\Sigma} R < a, a}{\vdash_{\Sigma} R^* < a, a}$   |
| $\frac{\vdash_{\Sigma} R < a, a' \quad \vdash_{\Sigma} R' < a', a''}{\vdash_{\Sigma} R; R' < a, a''}$   | $\frac{\vdash_{\Sigma} R < a, a' \quad \vdash_{\Sigma} R' < a, a' \quad * \in \{\cup, \cap, \setminus\}}{\vdash_{\Sigma} R * R' < a, a'}$           |
| $\frac{p : T_1, \dots, T_n \rightarrow prop \text{ in } \Sigma \quad \Gamma \vdash_{\Sigma} Q_i : T_i}{\Gamma \vdash_{\Sigma} p(Q_1, \dots, Q_n) : prop}$ |   |
| $\frac{\Gamma \vdash_{\Sigma} F : prop}{\Gamma \vdash_{\Sigma} \neg F : prop}$  | $\frac{\Gamma \vdash_{\Sigma} F : prop \quad \Gamma \vdash_{\Sigma} F' : prop}{\Gamma \vdash_{\Sigma} F \wedge F' : prop}$                          |
| $\frac{\Gamma \vdash_{\Sigma} Q : set(t) \quad \Gamma, x : t \vdash_{\Sigma} F(x) : prop}{\Gamma \vdash_{\Sigma} \forall x \in Q. F(x) : prop}$           |   |

**Fig. 4.** Well-Formed Expressions

*Example 1.* Consider a base type  $id : type$  of MMT identifiers in some fixed MMT library. Moreover, consider a concept symbol  $theory < id$  giving the identifiers of all theories, and a relation symbol  $includes < id, id$  that gives the relation “theory  $A$  directly includes theory  $B$ ”.

Then the query *theory* of type  $\text{set}(\text{id})$  yields the set of all theories. Given a theory  $u$ , the query  $\text{includes}^{*-1}(u)$  of type  $\text{set}(\text{id})$  yields the set of all theories that transitively include  $u$ .

*Example 2 (Continued).* Additionally, consider a concept  $\text{constant} < \text{id}$  of identifiers of MMT constants, relation symbol  $\text{declares} < \text{id}, \text{id}$  that relates every theory to the constants declared in it, a base type  $\text{obj} : \text{type}$  of OPENMATH objects, a function symbol  $\text{type} : \text{id} \rightarrow \text{obj}$  that maps each MMT constant to its type, and a predicate symbol  $\text{occurs} : \text{id}, \text{obj} \rightarrow \text{prop}$  that determines whether an identifier occurs in an object.

Then the following query of type  $\text{set}(\text{id})$  retrieves all constants that are included into the theory  $u$  and whose type uses the identifier  $v$ :

$$\{x \in (\text{includes}^*; \text{declares})(u) \mid \text{occurs}(v, \text{type}(x))\}$$

## 2.2 Semantics

A  $\Sigma$ -model assigns to every symbol  $s$  in  $\Sigma$  a denotation. The formal definition is given in Def. 1. Relative to a fixed model  $M$  (which we suppress in the notation), each well-formed expression has a well-defined denotational semantics, given by the interpretation function  $\llbracket - \rrbracket$ . The semantics of propositions and queries in context  $\Gamma$  is relative to an assignment  $\alpha$ , which assigns values to all variables in  $\Gamma$ . An overview is given in Fig. 6. The formal definition is given in Def. 2.

| Judgment                                   | Semantics   |
|--|---|
| $\vdash_{\Sigma} T : \text{type}$          | $\llbracket T \rrbracket \in \mathcal{SET}$   |
| $\Gamma \vdash_{\Sigma} Q : t$             | $\llbracket Q \rrbracket^{\alpha} \in \llbracket t \rrbracket$                              |
| $\Gamma \vdash_{\Sigma} Q : \text{set}(t)$ | $\llbracket Q \rrbracket^{\alpha} \subseteq \llbracket t \rrbracket$                        |
| $\vdash_{\Sigma} R < a, a'$                | $\llbracket R \rrbracket \subseteq \llbracket a \rrbracket \times \llbracket a' \rrbracket$ |
| $\Gamma \vdash_{\Sigma} F : \text{prop}$   | $\llbracket F \rrbracket^{\alpha} \in \{0, 1\}$   |

**Fig. 6.** Semantics of Judgments

**Definition 1 (Models).** A  $\Sigma$ -model  $M$  assigns to every  $\Sigma$ -symbol  $s$  a denotation  $s^M$  such that

- $a^M$  is a set for  $a : \text{type}$
- $c^M \subseteq \llbracket a \rrbracket$  for  $c < a$
- $r^M \subseteq \llbracket a \rrbracket \times \llbracket a' \rrbracket$  for  $r < a, a'$
- $f^M : \llbracket T_1 \rrbracket \times \dots \times \llbracket T_n \rrbracket \rightarrow \llbracket T \rrbracket$  for  $f : T_1, \dots, T_n \rightarrow T$
- $p^M : \llbracket T_1 \rrbracket \times \dots \times \llbracket T_n \rrbracket \rightarrow \{0, 1\}$  for  $p : T_1, \dots, T_n \rightarrow \text{prop}$

**Definition 2 (Semantics).** Given a  $\Sigma$ -model  $M$ , the interpretation function  $\llbracket - \rrbracket$  is defined as follows.

*Semantics of types:*

- $\llbracket a_1 \times \dots \times a_n \rrbracket$  is the cartesian product  $a_1^M \times \dots \times a_n^M$
- $\llbracket \text{set}(t) \rrbracket$  is the power set of  $\llbracket t \rrbracket$

*Semantics of relations:*

- $\llbracket r \rrbracket = r^M$
- $\llbracket R^{-1} \rrbracket$  is the dual/inverse relation of  $\llbracket R \rrbracket$ , i.e., the set  $\{(u, v) \mid (v, u) \in \llbracket R \rrbracket\}$

- $R^*$  is the transitive closure of  $\llbracket R \rrbracket$
- $R; R'$  is the composition of  $\llbracket R \rrbracket$  and  $\llbracket R' \rrbracket$ ,  
i.e., the set  $\{(u, w) \mid \text{exists } v \text{ such that } (u, v) \in \llbracket R \rrbracket, (v, w) \in \llbracket R' \rrbracket\}$
- $R \cup R', R \cap R', \text{ and } R \setminus R'$  are interpreted in the obvious way using the union, intersection, and difference of sets

Semantics of propositions under an assignment  $\alpha$ :

- $\llbracket p(Q_1, \dots, Q_n) \rrbracket^\alpha = p^M(\llbracket Q_1 \rrbracket^\alpha, \dots, \llbracket Q_n \rrbracket^\alpha)$
- $\llbracket \neg F \rrbracket^\alpha = 1 \quad \text{iff} \quad \llbracket F \rrbracket^\alpha = 0$
- $\llbracket F \wedge F' \rrbracket^\alpha = 1 \quad \text{iff} \quad \llbracket F \rrbracket^\alpha = 1 \text{ and } \llbracket F' \rrbracket^\alpha = 1$
- $\llbracket \forall x \in Q. F(x) \rrbracket^\alpha = 1 \quad \text{iff} \quad \llbracket F(x) \rrbracket^{\alpha, x/u} = 1 \quad \text{for all } u \in \llbracket Q \rrbracket^\alpha$

Semantics of queries  $\Gamma \vdash_{\Sigma} Q : T$  under an assignment  $\alpha$ :

- $\llbracket c \rrbracket^\alpha = c^M$
- $\llbracket x \rrbracket^\alpha = \alpha(x)$
- $\llbracket f(Q_1, \dots, Q_n) \rrbracket^\alpha = f^M(\llbracket Q_1 \rrbracket^\alpha, \dots, \llbracket Q_n \rrbracket^\alpha)$
- $\llbracket R(Q) \rrbracket^\alpha = \{u \in \llbracket a' \rrbracket \mid (\llbracket Q \rrbracket^\alpha, u) \in \llbracket R \rrbracket\}$  for a relation  $\vdash_{\Sigma} R < a, a'$  and a query  $\Gamma \vdash_{\Sigma} Q : a$   
informally,  $\llbracket R(Q) \rrbracket^\alpha$  is the image of  $\llbracket Q \rrbracket^\alpha$  under  $\llbracket R \rrbracket$
- $\llbracket \bigcup_{x \in Q} Q'(x) \rrbracket^\alpha$  is the union of all sets  $\llbracket Q'(x) \rrbracket^{\alpha, x/u}$  where  $u$  runs over all elements of  $\llbracket Q \rrbracket^\alpha$
- $\llbracket \{x \in Q \mid F(x)\} \rrbracket^\alpha$  is the subset of  $\llbracket Q \rrbracket^\alpha$  containing all elements  $u$  for which  $\llbracket F(x) \rrbracket^{\alpha, x/u} = 1$

*Remark 2.* It is easy to prove that if all concept and relation symbols are interpreted as finite sets and if all function symbols with result type  $\text{set}(t)$  always return finite sets, then all well-formed queries of type  $\text{set}(t)$  denote a *finite* subset of  $\llbracket t \rrbracket$ . Moreover, if the interpretations of the function and predicate symbols are computable functions, then the interpretation of queries is computable as well. This holds even if base types are interpreted as infinite sets.

### 2.3 Predefined Symbols

We use a number of predefined function and predicate symbols as given in Fig. 7. These are assumed to be implicitly declared in every signature, and their semantics is fixed. All of these symbols are overloaded for all simple types  $t$ . Moreover, we use special notations for them.

All of this is completely analogous to the usual treatment of equality as a predefined predicate symbol in first-order logic. The only difference is that our slightly richer type system calls for a few additional predefined symbols.

It is easy to add further predefined symbols, in particular equality of sets (which, however, may be inefficient to decide) and binary union of queries. We omit these here for simplicity.

| Symbol        | Type   | Semantics         |
|---------------|--|-------------------|
| $\{\cdot\}$   | $: t \rightarrow \text{set}(t)$              | the singleton set |
| $\_\doteq\_\$ | $: t, t \rightarrow \text{prop}$             | equality          |
| $\_\in\_\$    | $: t, \text{set}(t) \rightarrow \text{prop}$ | elementhood       |

Fig. 7. Predefined Symbols

## 2.4 Definable Queries

Using the predefined symbols, we can define a number of further useful query formation operators:

*Example 3.* Using the singleton symbol  $\{\cdot\}$ , we can define for  $\Gamma \vdash_{\Sigma} Q : set(t)$  and  $\Gamma, x : t \vdash_{\Sigma} q(x) : t'$

$$\{q(x) : x \in Q\} := \bigcup_{x \in Q} \{q(x)\} \quad \text{of type } set(t').$$

It is easy to show that, semantically, this is the replacement operator, i.e.,  $\llbracket \{q(x) : x \in Q\} \rrbracket^{\alpha}$  is the set containing exactly the elements  $\llbracket q(x) \rrbracket^{\alpha, x/u}$  for any  $u \in \llbracket Q \rrbracket^{\alpha}$ .

*Example 4 (SQL-style Queries).* For a query  $\vdash_{\Sigma} Q : set(a_1 \times \dots \times a_N)$ , natural numbers  $n_1, \dots, n_k \in \{1, \dots, N\}$ , and a proposition  $x_1 : a_1, \dots, x_N : a_N \vdash_{\Sigma} F(x_1, \dots, x_n) : prop$ , we write

**select**  $n_1, \dots, n_k$  **from**  $Q$  **where**  $F(1, \dots, N)$

for the query

$$\{x_{n_1} * \dots * x_{n_k} : x \in \{y \in Q \mid F(y_1, \dots, y_N)\}\}$$

of type  $set(a_{n_1} \times \dots \times a_{n_k})$ .

*Example 5 (XQuery-style Queries).* For queries  $\vdash_{\Sigma} Q : set(a)$  and  $x : a \vdash_{\Sigma} q'(x) : a'$  and  $x : a, y : a' \vdash_{\Sigma} Q''(x, y) : set(a'')$ , and a proposition  $x : a, y : a' \vdash_{\Sigma} F(x, y) : prop$ , we write

**for**  $x$  **in**  $Q$  **let**  $y = q'(x)$  **where**  $F(x, y)$  **return**  $Q''(x, y)$

for the query

$$\bigcup_{z \in P} Q''(z_1, z_2) \quad \text{with} \quad P := \{z \in \{x * q'(x) : x \in Q\} \mid F(z_1, z_2)\}$$

of type  $set(a'')$ .

*Example 6 (DL-style Queries).* For a relation  $\vdash_{\Sigma} R < a, a'$ , a concept  $c < a$ , and a query  $\vdash_{\Sigma} Q : set(a')$ , we write  $\square^c R.Q$  for the query  $\{x \in c \mid \forall y \in R(x).y \in Q\}$  of type  $set(a)$ .

Note that, contrary to the universal restriction  $\square R.Q$  in description logic, we have to restrict the query to all  $x$  of concept  $c$  instead of querying for all  $x$  of type  $a$ . This makes sense in our setting because we assume that we can only iterate efficiently over concepts but not over (possibly infinite!) base types.

However, this is not a loss of generality: individual signatures may always couple a base type  $a$  with a concept  $is_a$  such that  $\llbracket is_a \rrbracket = \llbracket a \rrbracket$ .

| Declaration   | Intuition  |
|---|--|
| Base types  |  |
| $id : type$   | URIs of MMT declarations                         |
| $obj : type$  | MMT (OPENMATH) OBJECTS                           |
| $xml : type$  | XML elements                                     |
| Concepts  |  |
| $theory < id$   | theories   |
| $view < id$   | views  |
| $constant < id$   | constants  |
| $style < id$  | styles   |
| Relations   |  |
| $includes < id, id$                                     | inclusion between theories                       |
| $declares < id, id$                                     | declarations in a theory                         |
| $domain < id, id$                                       | domain of structure/view                         |
| $codomain < id, id$                                     | codomain of structure/view                       |
| Functions   |  |
| $type : id \rightarrow obj$                             | type of a constant                               |
| $def : id \rightarrow obj$                              | definiens of a constant                          |
| $infer : id, obj \rightarrow obj$                       | type inference relative to a theory              |
| $arg_p : obj \rightarrow obj$                           | argument at position $p$                         |
| $subobj : obj, id \rightarrow set(obj)$                 | subobjects with a certain head                   |
| $unify : obj \rightarrow set(id \times obj \times obj)$ | all objects that unify with a given one          |
| $render : id, id \rightarrow xml$                       | rendering of a declaration using a certain style |
| $render : obj, id \rightarrow xml$                      | rendering of an object using a certain style     |
| $u : id$  | literals for MMT URIs $u$                        |
| $o : obj$   | literals for MMT objects $o$                     |
| Predicates  |  |
| $occurs : id, obj \rightarrow prop$                     | occurs in  |

**Fig. 8.** The QMT Signature for MMT

### 3 Querying MMT Libraries

We will now fix an MMT-specific signature  $\Sigma$  that customizes QMT with the MMT ontology as well as with several functions and predicates based on the MMT specification. The declarations of  $\Sigma$  are listed in Fig. 8.

For simplicity, we avoid presenting any details of MMT and refer to [RK11] for a comprehensive description. For our purposes, it is sufficient to know that MMT organizes mathematical knowledge in a simple ontology that can be seen as the fragment of OMDoc pertaining to formal theories. We will explain the necessary details below when explaining the respective  $\Sigma$ -symbols.

An MMT **library** is any set of MMT declarations (not necessarily well-typed or closed under dependency). We will assume a fixed library  $L$  in the following. Based on  $L$ , we will define a model  $M$  by giving the interpretation  $s^M$  for every symbol  $s$  listed in Fig. 8.

*Base Types* We use three base types. Firstly, every MMT declaration has a globally unique **canonical identifier**, its MMT URI. We use this to define  $id^M$  as the set of all MMT URIs declared in  $L$ .

$obj^M$  the set of all OPENMATH objects that can be formed from the symbols in  $id^M$ . In order to handle objects with free variables conveniently, we use the following convention: All objects in  $obj^M$  are technically closed; but we permit the use of a special binder symbol **free**, which can be used to formally bind the free variables. This has the advantage that the context of an object, which may carry, e.g., type attributions, is made explicit. Using general OPENMATH objects means that the type  $obj$  is subject to exactly  $\alpha$ -equality and attribution flattening, the only equalities defined in the OPENMATH standard. The much more difficult problem of queries relative to a stronger equality relation remains future work.

The remaining base type  $xml$  is a generic container for any non-MMT **XML data** such as HTML or presentation MathML. Thus,  $xml^M$  is the set of all XML elements. This is useful because the MMT API contains several functions that return XML.

*Ontology* For simplicity, we restrict attention to the most important notions of the MMT ontology; adding the remaining notions is straightforward. The ontology only covers the MMT declarations, all of which have canonical identifiers. Thus, all concepts refine the type  $id$ , and all relations are between identifiers.

Among the MMT **concepts**, **theories** are used to represent logics, theories of a logic, ontologies, type theories, etc. They contain **constants**, which represent function symbols, type operators, inference rules, theorems, etc. Constants may have OPENMATH **objects** [BCC<sup>+</sup>04] as their type or definiens. Theories are related via theory morphisms called **views**. These are truth-preserving translations from one theory to another and represent translations and models. Theories and views together form a multi-graph of theories across which theorems can be shared. Finally, **styles** contain notations that govern the translation from content to presentation markup.

MMT theories, views, and styles can be structured by a strong module system. The most important modular construct is the *includes* relation for explicit imports. The *declares* relation relates every theory to the constants it declares; this includes the constants that are not explicitly declared in  $L$  but induced by the module system. Finally, two further relations connect each view to its *domain* and *codomain*.

All concepts and relations are interpreted in the obvious way. For example, the set  $theory^M$  contains the MMT URIs of all theories in  $L$ .

*Function and Predicate Symbols* Regarding the function and predicate symbols, we are very flexible because a wide range of operations can be defined for MMT libraries. In particular, every function implemented in the MMT API can be easily exposed as a  $\Sigma$ -symbol. Therefore, we only show a selection of symbols that showcase the potential.

In Sect. 2, we have deliberately omitted **partial function symbols** in order to simplify the presentation of our language. However, in practice, it is often necessary to add them. For example,  $\text{def}^M$  must be a partial function because (i) the argument might not be the MMT URI of a constant declaration in  $L$ , or (ii) even if it is, that constant may be declared without a definiens. The best solution for an elegant treatment of partial functions is to use option types  $\text{opt}(t)$  akin to set types  $\text{set}(t)$ . However, for simplicity, we make  $\llbracket - \rrbracket$  a partial function that is undefined whenever the interpretation of its argument runs into an undefined function application. This corresponds to the common concept of queries returning an error value.

The partial **functions**  $\text{type}^M$  and  $\text{def}^M$  take the identifier of a constant declaration and return its type or definiens, respectively. They are undefined for other identifiers.

The partial function  $\text{infer}^M(u, o)$  takes an object  $o$  and returns its dynamically inferred type. It is undefined if  $o$  is ill-typed. Since MMT does not commit to a type system, the argument  $u$  must identify the type system (which is represented as an MMT theory itself). If  $O$  is a binding object of the form  $\text{OMBIND}(\text{OMS}(\text{free}), \Gamma, o')$ , the type of  $o'$  is inferred in context  $\Gamma$ .

$\text{arg}_p$  is a family of function symbols indexed by a natural number  $p$ .  $p$  indicates the position of a direct subobject (usually an argument), and  $\text{arg}_p^M(o)$  is the subobject of  $o$  at position  $p$ . In particular,  $\text{arg}_i^M(\text{OMA}(f, a_1, \dots, a_n)) = a_i$ . Note that arbitrary subobjects can be retrieved by iterating  $\text{arg}_p$ . Similarly,  $\text{subobj}^M(o, h)$  is the set of all subobjects of  $o$  whose head is the symbol with identifier  $h$ . In particular, the head of  $\text{OMA}(\text{OMS}(h), a_1, \dots, a_n)$  is  $h$ . In both cases, we keep track of the free variables, e.g.,  $\text{arg}_2^M(\text{OMBIND}(b, \Gamma, o)) = \text{OMBIND}(\text{OMS}(\text{free}), \Gamma, o)$  for  $b \neq \text{OMS}(\text{free})$ .

$\text{unify}^M(O)$  performs an object query: It returns the set of all tuples  $u * o * s$  where  $u$  is the MMT URI of a declaration in  $L$  that contains an object  $o$  that unifies with  $O$  using the substitution  $s$ . Here we use a purely syntactic definition for unifiability of OPENMATH objects.

$\text{render}^M(o, u)$  and  $\text{render}^M(d, u)$  return the presentation markup dynamically computed by the MMT rendering engine. This is useful because the query and the rendering engine are often implemented on the same remote server. Therefore, it is reasonable to compute the rendering of the query results, if desired, as part of the query evaluation. Moreover, larger signatures might provide additional functions to further operate on the presentation markup.  $\text{render}$  is overloaded because we can present both MMT declarations and MMT objects. In both cases,  $u$  is the MMT URI of the style providing the notations for the rendering.

The **predicate** symbol  $\text{occurs}$  takes an object  $O$  and an identifier  $u$ , and returns true if  $u$  occurs in  $O$ .

Finally, we permit **literals**, i.e., arbitrary URIs and arbitrary OPENMATH objects may be used as nullary constants, which are interpreted as themselves (or as undefined if they are not in the universe). This is somewhat inelegant but necessary in practice to form interesting queries. A more sophisticated QMT

signature could use one function symbol for every OPENMATH object constructor instead of using OPENMATH literals.

*Example 7.* An MMT theory graph is the multigraph formed by using the theories as nodes and all theory morphisms between them as edges. The components of the theory graph can be retrieved with a few simple queries.

Firstly, the set of theories is retrieved simply using the query *theory*. Secondly, the theory morphisms are obtained by two different queries:

$$\begin{aligned} \text{views} & \quad \{v * x * y : v \in \text{view}, x \in \text{domain}(v), y \in \text{domain}(v)\} \\ \text{inclusions} & \quad \bigcup_{y \in \text{theory}} \{x * y : x \in \text{includes}^*(y)\} \end{aligned}$$

The first one returns all view identifiers with their domain and codomain. Here we use an extension of the replacement operator  $\{\_ : \_\}$  from Ex. 3 to multiple variables. It is straightforward to define in terms of the unary one. The second query returns all pairs of theories between which there is an inclusion morphism.

*Example 8.* Consider a constant identifier  $\exists I$  for the introduction rule of the existential quantifier from the natural deduction calculus. It produces a constructive existence proof of  $\exists x.P(x)$ ; it takes two arguments: a witness  $w$ , and a proof of  $P(w)$ . Moreover, consider a theorem with identifier  $u$ . Recall that using the Curry-Howard representation of proofs-as-objects, a theorem  $u$  is a constant, whose type is the asserted formula and whose definiens is the proof.

Then the following query retrieves all existential witnesses that come up in the proof of  $u$ :

$$\{\text{arg}_1(x) : x \in \text{subobj}(\text{def}(u), \exists I)\}$$

Here we have used the replacement operator introduced in Ex. 3.

*Example 9 (Continuing Ex. 8).* Note that when using  $\exists I$ , the proved formula  $P$  is present only implicitly as the type of the second argument of  $\exists I$ . If the type system is given by, for example,  $LF$  and type inference for  $LF$  is available, we can extend the query from Ex. 8 as follows:

$$\{\text{arg}_1(x) * \text{infer}(LF, \text{arg}_2(x)) : x \in \text{subobj}(\text{def}(u), \exists I)\}$$

This will retrieve all pairs  $(w, P)$  of witnesses and proved formulas that come up in the proof of  $u$ .

## 4 Implementation

We have implemented QMT as a part of the MMT API. The implementation includes a concrete XML syntax for queries and an integration with the MMT web server, via which the query engine is exposed to users. The server can run as a background service on a local machine as well as a dedicated remote server. Sources, binaries, and documentation are available at the project web site [Rab08].

The MMT API already implements the MMT ontology so that appropriate indices for the semantics of all concept and relation symbols are available. Indices scale well because they are written to the hard drive and cached to memory on demand. With two exceptions, the semantics of all function and predicate symbols is implemented by standard MMT API functions.

The semantics of *unify* is computed differently: A substitution tree index of the queries library is maintained separately by an installation of MathWebSearch [KS06]. Thus, QMT automatically inherits some heuristics of MathWebSearch, such as unification up to symmetry of certain relation symbols. MathWebSearch and query engine run on the same machine and communicate via HTTP.

Another subtlety is the semantics of *infer*. The MMT API provides a plugin interface, through which individual type systems can be registered; the first argument to  $\text{infer}^M$  is used to choose an applicable plugin. In particular, we provide a plugin for the logical framework LF [HHP93], which handles type inference for any type system that is formalized in LF; this covers all type systems defined in the LATIN library [CHK<sup>+</sup>11] and thus also applies to our imports of the Mizar [TB85] and TPTP libraries [SS98].

Query servers for individual libraries can be set up easily. In fact, because the MMT API abstracts from different backends, queries automatically return results from all libraries that are registered with a particular instance of the MMT API. This permits queries across libraries, which is particularly interesting if libraries share symbols. Shared symbols arise, for example, if both libraries use the standard OpenMath CDs where possible or if overlap between the libraries' underlying meta-languages is explicated in an integrating framework like the LATIN atlas [CHK<sup>+</sup>11].

*Example 10.* The LATIN library [CHK<sup>+</sup>11] consists of over 1000 highly modularized LF signatures and views between them, formalizing a variety of logics, type theories, set theories, and related formal systems. Validating the library and producing the index for the MMT ontology takes a few minutes with typical desktop hardware; reading the index into memory takes a few seconds. Typical queries as given in this paper are evaluated within seconds.

As an extreme example, consider the query  $Q = \text{Declares}(\text{theory})$ . It returns in less than a second the about 2000 identifiers that are declared in any theory. The query  $\bigcup_{x \in Q} \{x * \text{type}(x)\}$  returns the same number of results but pairs every declaration with its type. This requires the query engine to read the types of all declarations (as opposed to only their identifiers). If none of these are cached in memory yet, the evaluation takes about 4 minutes.

## 5 Conclusion and Future Work

We have introduced a simple, expressive query language for mathematical theories (QMT) that combines features of compositional, property, and object query languages. QMT is implemented on top of the MMT API; that provides any library that is serialized as MMT content markup with a scalable, versatile querying engine out of the box. As both MMT and its implementation are designed to

admit natural representations of any declarative language, QMT can be readily applied to many libraries including, e.g., those written in Twelf, Mizar, or TPTP.

Our presentation focused on querying *formal* mathematical libraries. This matches our primary motivation but is neither a theoretical nor a practical restriction. For example, it is straightforward to add a base type for presentation MathML and some functions for it. MathWebSearch can be easily generalized to permit unification queries on presentation markup. This also permits queries that mix content and presentation markup, or content queries that find presentation results. Moreover, for presentation markup that is generated from content markup, it is easy to add a function that returns the corresponding content item so that queries can jump back and forth between them.

Similarly, we can give a QMT signature with base types for authors and documents (papers, book chapters, etc.) as well as relations like `author-of` and `cites`. It is easy to generate the necessary indices from existing databases and to reuse our implementation for them. Moreover, with a relation `mentions` between papers and the type *id* of mathematical concepts, we can combine content and narrative aspects in queries. An index for the `mentions` relation is of course harder to obtain, which underscores the desirability of mathematical documents that are annotated with content URIs.

## References

- ADL12. D. Aspinall, E. Denney, and C. Lüth. Querying Proofs. In *Proceedings of LPAR*, 2012. To appear.
- ANS03. ANSI/ISO/IEC. 9075:2003, Database Language SQL, 2003.
- AS04. Andrea Asperti and Matteo Selmi. Efficient Retrieval of Mathematical Statements. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Mathematical Knowledge Management*, pages 17–31. Springer, 2004.
- AY08. M. Altamimi and A. Youssef. A Math Query Language with an Expanded Set of Wildcards. *Mathematics in Computer Science*, 2:305–331, 2008.
- BCC<sup>+</sup>04. S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See <http://www.openmath.org/standard/om20>.
- BR03. G. Bancerek and P. Rudnicki. Information Retrieval in MML. In A. Asperti, B. Buchberger, and J. Davenport, editors, *Mathematical Knowledge Management*, pages 119–132. Springer, 2003.
- CHK<sup>+</sup>11. M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. Project Abstract: Logic Atlas and Integrator (LATIN). In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 287–289. Springer, 2011.
- GC03. F. Guidi and C. Sacerdoti Coen. Querying Distributed Digital Libraries of Mathematics. In T. Hardin and R. Rioboo, editors, *Proceedings of Calculemus*, pages 17–30, 2003.
- HHP93. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

- Koh06. M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*. Number 4180 in Lecture Notes in Artificial Intelligence. Springer, 2006.
- KRZ10. M. Kohlhase, F. Rabe, and V. Zholudev. Towards MKM in the Large: Modular Representation and Scalable Software Architecture. In S. Autexier, J. Calmet, D. Delahaye, P. Ion, L. Rideau, R. Rioboo, and A. Sexton, editors, *Intelligent Computer Mathematics*, volume 6167 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2010.
- KŞ06. M. Kohlhase and I. Şucan. A Search Engine for Mathematical Formulae. In T. Ida, J. Calmet, and D. Wang, editors, *Artificial Intelligence and Symbolic Computation*, pages 241–253. Springer, 2006.
- LM06. P. Libbrecht and E. Melis. Methods to Access and Retrieve Mathematical Content in ActiveMath. In A. Iglesias and N. Takayama, editors, *International Congress on Mathematical Software*, pages 331–342. Springer, 2006.
- MG08. J. Mišutka and L. Galamboš. Extending full text search engine for mathematical content. In P. Sojka, editor, *Towards a Digital Mathematics Library*, pages 55–67, 2008.
- MM06. R. Munavalli and R. Miner. MathFind: a math-aware search engine. In E. Efthimiadis, S. Dumais, D. Hawking, and K. Järvelin, editors, *International ACM SIGIR Conference on Research and Development in Information Retrieval*, page 735. ACM, 2006.
- MY03. B. Miller and A. Youssef. Technical Aspects of the Digital Library of Mathematical Functions. *Annals of Mathematics and Artificial Intelligence*, 38(1–3):121–136, 2003.
- Rab08. F. Rabe. The MMT System, 2008. see <https://trac.kwarc.info/MMT/>.
- RK11. F. Rabe and M. Kohlhase. A Scalable Module System. see <http://arxiv.org/abs/1105.0548>, 2011.
- SL11. P. Sojka and M. Líška. Indexing and Searching Mathematics in Digital Libraries - Architecture, Design and Scalability Issues. In J. Davenport, W. Farmer, J. Urban, and F. Rabe, editors, *Intelligent Computer Mathematics*, pages 228–243. Springer, 2011.
- SS98. G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
- TB85. A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985.
- Urb06. J. Urban. MOMM - Fast Interreduction and Retrieval in Large Libraries of Formalized Mathematics. *International Journal on Artificial Intelligence Tools*, 15(1):109–130, 2006.
- W3C03. W3C. Mathematical Markup Language (MathML) Version 2.0 (second edition), 2003. See <http://www.w3.org/TR/MathML2>.
- W3C07. W3C. XQuery 1.0: An XML Query Language, 2007. <http://www.w3.org/TR/xquery/>.
- W3C08. W3C. SPARQL Query Language for RDF, 2008. <http://www.w3.org/TR/rdf-sparql-query/>.
- ZK09. V. Zholudev and M. Kohlhase. TNTBase: a Versioned Storage for XML. In *Proceedings of Balisage: The Markup Conference 2009*, volume 3 of *Balisage Series on Markup Technologies*. Mulberry Technologies, Inc., 2009.

# Extending MKM Formats at the Statement Level

Fulya Horozal, Michael Kohlhase, and Florian Rabe

Computer Science, Jacobs University Bremen, Germany <http://kwarc.info>

**Abstract.** Successful representation and markup languages find a good balance between giving the user freedom of expression, enforcing the fundamental semantic invariants of the modeling framework, and allowing machine support for the underlying semantic structures. MKM formats maintain strong invariants while trying to be foundationally unconstrained, which makes the induced design problem particularly challenging.

In this situation, it is standard practice to define a minimal core language together with a scripting/macro facility for syntactic extensions that map into the core language. In practice, such extension facilities are either fully unconstrained (making invariants and machine support difficult) or limited to the object level (keeping the statement and theory levels fixed).

In this paper we develop a general methodology for extending MKM representation formats at the statement level. We show the utility (and indeed necessity) of statement-level extension by redesigning the OMDoc format into a minimal, regular core language (strict OMDoc) and an extension (pragmatic OMDoc) that maps into strict OMDoc.

## 1 Introduction

The development of representation languages for mathematical knowledge is one of the central concerns of the MKM community. After all, practical mathematical knowledge management consists in the manipulation of expressions in such languages. To be successful, MKM representation formats must balance multiple concerns. A format should be expressive and flexible (for depth and ease of modeling), foundationally unconstrained (for coverage), regular and minimal (for ease of implementation), and modular and web-transparent (for scalability). Finally, the format should be elegant, feel natural to mathematicians, and be easy to read and write. Needless to say that this set of requirements is over-constrained so that the design problem for MKM representation formats lies in relaxing some of the constraints to achieve a global optimum.

In languages for formalized mathematics, it is standard practice to define a minimal core language that is extended by macros, functions, or notations. For example, Isabelle [Pau94] provides a rich language of notations, abbreviations, syntax and printing translations, and a number of definitional forms. In narrative formats for mathematics, for instance, the  $\text{\TeX}/\text{\LaTeX}$  format – arguably the most commonly used format for representing mathematical knowledge – goes a

similar way, only that the core language is given by the  $\text{\TeX}$  layout primitives and the translation is realized by macro expansion and is fully under user control. This extensibility led to the profusion of user-defined  $\text{\LaTeX}$  document classes and packages that has made  $\text{\TeX}/\text{\LaTeX}$  so successful.

However, the fully unconstrained nature of the extensibility makes ensuring invariants and machine support very difficult, and thus this approach is not immediately applicable to content markup formats. There, MathML3 [Aus+10] is a good example of the state of the art. It specifies a core language called “strict content MathML” that is equivalent to OpenMath [Bus+04b] and “full content MathML”. The first subset uses a minimal set of elements representing the meaning of a mathematical expression in a uniform, regular structure, while the second one tries to strike a pragmatic balance between verbosity and formality. The meaning of non-strict expressions is given by a fixed translation: the “strict content MathML translation” specified in section 4.6 of the MathML3 recommendation [Aus+10].

This language design has the advantage that only a small, regular sublanguage has to be given a mathematical meaning, but a larger vocabulary that is more intuitive to practitioners of the field can be used for actual representation. Moreover, semantic services like validation only need to be implemented for the strict subset and can be extended to the pragmatic language by translation. Ultimately, a representation format might even have multiple pragmatic front-ends geared towards different audiences. These are semantically interoperable by construction.

The work reported in this paper comes from an ongoing language design effort, where we want to redesign our OMDoc format [Koh06] into a minimal, regular core language (*strict OMDoc 2*) and an extension layer (*pragmatic OMDoc 2*) whose semantics is given by a “pragmatic-to-strict” ( $\mathcal{P}2\mathcal{S}$ ) translation. While this problem is well-understood for mathematical *objects*, extension frameworks *at the statement level* seem to be restricted to the non-semantic case, e.g. the `amsthm` package for  $\text{\LaTeX}$ .

Languages for mathematics commonly permit a variety of pragmatic statements, e.g., implicit or case-based definitions, type definitions, theorems, or proof schemata. But representation frameworks for such languages do not include a generic mechanism that permits introducing arbitrary pragmatic statements — instead, a fixed set is built into the format. Among logical frameworks, Twelf/LF [PS99; HHP93] permits two statements: defined and undefined constants. Isabelle [Pau94] and Coq [BC04] permit much larger, but still fixed sets that include, for example, recursive case-based function definitions. Content markup formats like OMDoc permit similar fixed sets.

A large set of statements is desirable in a representation format in order to model the flexibility of individual languages. A large *fixed* set on the other hand is unsatisfactory because it is difficult to give a theoretical justification for fixing any specific set of statements. Moreover, it is often difficult to define the semantics of a built-in statement in a foundationally unconstrained representation

format because many pragmatic statement are only meaningful under certain foundational assumptions.

In this paper we present a general formalism for adding new pragmatic statement forms to our OMDoc format; we have picked OMDoc for familiarity and foundation-independence; any other foundational format can be extended similarly. Consider for instance the pragmatic statement of an “implicit definition”, which defines a mathematical object by describing it so accurately, that there is only one object that fits this description. For instance, the exponential function  $\exp$  is defined as the (unique) solution of the differential equation  $f = f'$  with  $f(0) = 1$ . This form of definition is extensively used in practical mathematics, so pragmatic OMDoc should offer an infrastructure for it, whereas strict OMDoc only offers “simple definitions” of the form  $c := d$ , where  $c$  is a new symbol and  $d$  any object. In our extension framework, the  $\mathcal{P}2S$  translation provides the semantics of the implicit definition in terms of the strict definition  $\exp := \iota f. (f' = f \wedge f(0) = 1)$ , where  $\iota$  is a “definite description operator”: Given an expression  $A$  with free variable  $x$ , such that there is a unique  $x$  that makes  $A$  valid,  $\iota x.A$  returns that  $x$ , otherwise  $\iota x.A$  is undefined.

Note that the semantics of an implicit definition requires a definite description operator. While most areas of mathematics at least implicitly assume its existence, it should not be required in general because that would prevent the representation of systems without one. Therefore, we make these requirements explicit in a special theory that defines the new pragmatic statement and its strict semantics. This theory must be imported in order for implicit definitions to become available. Using our extension language, we can recover a large number of existing pragmatic statements as definable special cases, including many existing ones of OMDoc. Thus, when representing formal languages in OMDoc, authors have full control what pragmatic statements to permit and can define new ones in terms of existing ones.

In the next section, we will recap those parts of OMDoc that are needed in this paper. In Section 3, we define our extension language, and in Section 4, we look at particular extensions that are motivated by mathematical practice. Finally, in Section 5, we will address the question of extending the concrete syntax with pragmatic features as well.

## 2 MMT/OMDoc

OMDoc is a comprehensive content-based format for representing mathematical knowledge and documents. It represents mathematical knowledge at three levels: mathematical formulae at the *object level*, symbol declarations, definitions, notation definitions, axioms, theorems, and proofs at the *statement level*, and finally modular scopes at the *theory level*. Moreover, it adds an infrastructure for representing functional aspects of *mathematical documents* at the content markup level. OMDoc

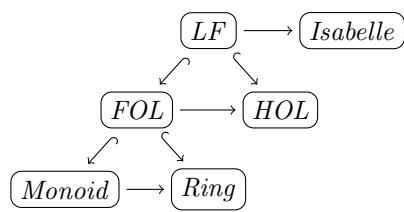
|            |           |
|------------|-----------|
| Theories   | Documents |
| Statements |           |
| Objects    |           |

1.2 has been successfully used as a representational basis in applications ranging from theorem prover interfaces, via knowledge based up to eLearning systems. To allow this diversity of applications, the format has acquired a large, interconnected set of language constructs motivated by coverage and user familiarity (i.e., by pragmatic concerns) and not by minimality and orthogonality of language primitives (strict concerns).

To reconcile these language design issues for OMDoc 2, we want to separate the format into a *strict* core language and a *pragmatic* extension layer that is elaborated into strict OMDoc via a “pragmatic-to-strict” ( $\mathcal{P}2\mathcal{S}$ ) translation.

For strict OMDoc we employ the foundation-independent, syntactically minimal MMT framework (see below). For pragmatic OMDoc, we aim at a language that is feature-complete with respect to OMDoc 1.2 [Koh06], but incorporates language features from other MKM formats, most notably from Isabelle/Isar [Wen99], PVS [ORS92], and Mizar [TB85].

The MMT language was emerged from a complete redesign of the formal core<sup>1</sup> of OMDoc focusing on foundation-independence, scalability, modularity, while maintaining coverage of formal systems. The MMT language is described in [RK11] and implemented in [Rab08].



**Fig. 1.** An MMT Theory Graph

and *HOL* for higher-order logic. In general, we describe the theories with meta-theory  $M$  as  **$M$ -theories**. The importance of meta-theories in MMT is that the syntax and semantics of  $M$  induces the syntax and semantics of all  $M$ -theories. For example, if the syntax and semantics are fixed for *LF*, they determine those of *FOL* and *Monoid*.

At the statement level, MMT uses **constant** declarations as a single primitive to represent all OMDoc statement declarations. These are differentiated by the type system of the respective meta-theory. In particular, the Curry-Howard correspondence is used to represent axioms and theorems as plain constants (with special types).

In Figure 2, we show a small fragment of the MMT grammar that we need in the remainder of this paper. Meta-symbols of the BNF format are given in color.

---

<sup>1</sup> We are currently working on adding an informal (natural language) representation and a non-trivial (strict) document level to MMT, their lack does not restrict the results reported in this paper.

MMT uses **theories** as a single primitive to represent formal systems such as logical frameworks, logics, or theories. These form theory graphs such as the one on the left, where single arrows  $\rightarrow$  denote theory translations and hooked arrows  $\hookrightarrow$  denote the meta-theory relation between two theories. The theory *FOL* for first-order logic is the meta-theory for *Monoid* and *Ring*. And the theory *LF* for the logical framework LF [HHP93] is the meta-theory of *FOL*.

|             |   |
|-------------|---|
| Modules     | $G ::= (\text{theory } T = \{\Sigma\})^*$                     |
| Theories    | $\Sigma ::= \cdot \mid \Sigma, c[:E][=E] \mid \text{meta } T$ |
| Contexts    | $\Gamma ::= \cdot \mid \Gamma, x[:E]$                         |
| Expressions | $E ::= x \mid c \mid E E^+ \mid E \Gamma. E$                  |

**Fig. 2.** MMT Grammar

The module level of MMT introduces *theory declarations*  $\text{theory } T = \{\Sigma\}$ . Theories  $\Sigma$  contain *constant declarations*  $c[:E_1][=E_2]$  that introduce named atomic expressions  $c$  with optional type  $E_1$  or definition  $E_2$ . Moreover, each theory may declare its meta-theory  $T$  via *meta*  $T$ .

MMT expressions are a fragment of OpenMath [Bus+04a] objects, for which we introduce a short syntax. They are formed from variables  $x$ , constants  $c$ , applications  $E E_1 \dots E_n$  of functions  $E$  to a sequence of arguments  $E_i$ , and bindings  $E_1 \Gamma. E_2$  that use a binder  $E_1$ , a context  $\Gamma$  of bound variables, and a scope  $E_2$ . Contexts  $\Gamma$  consist of variables  $x[:E]$  that can optionally attribute a type  $E$ .

The semantics of MMT is given in terms of *foundations* for the upper-most meta-theories. Foundations define in particular the typing relation between expressions, in which MMT is parametric. For example, the foundation for *LF* induces the type-checking relation for all theories with meta-theory *LF*.

*Example 1 (MMT-Theories).* Below we give an MMT theory *Propositions*, which will serve as the meta-theory of several logics introduced in this paper. It introduces all symbols needed to declare logical connectives and inference rules of a logic. The syntax and semantics of this theory are defined in terms of type theory, e.g., the logical framework LF [HHP93].

*type*,  $\rightarrow$ , and *lam* are untyped constants representing the primitives of type theory. *type* represents the universe of all types,  $\rightarrow$  constructs function types  $\alpha \rightarrow \beta$ , and *lam* represents the  $\lambda$ -binder.  $o$  is the type of logical formulas and *proof* is a constant that assigns to each logical formula  $F : o$  the type *proof*  $F$  of its proof.

```
theory Propositions = {
  type
  →
  lam
  o    : type
  proof : o → type
}
```

### 3 A Framework for Language Extensions

We will now define our extension language (EL). It provides a syntactic means to define pragmatic language features and their semantics in terms of strict OMDoc.

*Syntax* EL adds two primitive declarations to MMT theories: *extension declarations* and *pragmatic declarations*:

|   |
|---|
| $\Sigma ::= \Sigma, \text{extension } e = \Phi$       |
| $\quad   \quad \Sigma, \text{pragmatic } c : \varphi$ |

Extension declarations `extension e =  $\Phi$`  introduce a new declaration schema  $e$  that is described by  $\Phi$ . Intuitively,  $\Phi$  is a function that takes some arguments and returns a list of declarations, which define the strict semantics of the declaration scheme.

Pragmatic declarations `pragmatic c :  $\varphi$`  introduce new declarations that make use of a previously declared extension. Intuitively,  $\varphi$  applies an extension  $e$  a sequence of arguments and evaluates to the returned list of declarations. Thus,  $c : \varphi$  serves as a pragmatic abbreviation of a list of strict declarations.

The key notion in both cases is that of *theory families*. They represent collections of theories by specifying their common syntactic shape. Intuitively, theory families arise by putting a  $\lambda$ -calculus on top of theory fragments  $\Sigma$ :

|                 |  |
|-----------------|--|
| Theory Families | $\Phi ::= \{\Sigma\} \mid \lambda x : E. \Phi$ |
|                 | $\varphi ::= e \mid \Phi E$                    |

We group theory families into two non-terminal symbols as shown above:  $\Phi$  is formed from theory fragments  $\{\Sigma\}$  and  $\lambda$ -abstraction  $\lambda x : E. \Phi$ . And  $\varphi$  is formed from references to previously declared extension  $e$  and applications of parametric theory families to arguments  $E$ . This has the advantage that both  $\Phi$  and  $\varphi$  have a very simple shape.

*Example 2 (Extension Declarations).* In Figure 3 we give the theory `Assertion`, which declares extensions for axiom and theorem declarations. Their semantics is defined in terms of the Curry-Howard representation of strict OMDoc.

Both extensions take a logical formula  $F : o$  as a parameter. The extension `axiom` permits pragmatic declarations of the form  $c : \text{axiom } F$ . These abbreviate MMT constant declarations of the form  $c : \text{proof } F$ .

The extension `theorem` additionally takes a parameter  $D : \text{proof } F$ , which is a proof of  $F$ . It permits pragmatic declarations of the form  $c : \text{theorem } F D$ . These abbreviate MMT constant declarations of the form  $c : \text{proof } F = D$ .

```

theory Assertion = {
  meta Propositions
  extension axiom =  $\lambda F : o. \{$ 
    c : proof F
  }
  extension theorem =  $\lambda F : o. \lambda D : \text{proof } F. \{$ 
    c : proof F = D
  }
}

```

**Fig. 3.** An MMT Theory with Extension Declarations

Any MMT theory may introduce extension declarations. However, pragmatic declarations are only legal if the extension that is used has been declared in the meta-theory:

**Definition 1 (Legal Extension Declarations).** *We say that an extension declaration  $\text{extension } e = \lambda x_1 : E_1. \dots \lambda x_n : E_n. \{\Sigma\}$  is **legal** in an MMT theory  $T$ , if the declarations  $x_1 : E_1, \dots, x_n : E_n$  and  $\Sigma$  are well-formed in  $T$ .*

*This includes the case where  $\Sigma$  contains pragmatic declarations.*

**Definition 2 (Legal Pragmatic Declarations).** *We say that a pragmatic declaration  $\text{pragmatic } c : e E_1 \dots E_n$  is **legal** in an MMT theory  $T$  if there is a declaration extension  $e = \lambda x_1 : E'_1. \dots \lambda x_n : E'_n. \{\Sigma\}$  in the meta-theory of  $T$  and each  $E_i$  has type  $E'_i$ .*

Here the typing relation is the one provided by the MMT foundation.

*Semantics* Extension declarations do not have a semantics as such because the extension declared in  $M$  only govern what pragmatic declarations are legal in  $M$ -theories. In particular, contrary to the constant declarations in  $M$ , a model of  $M$  does not interpret the extension declarations.

The semantics of pragmatic declarations is given by elaborating them into strict declarations:

**Definition 3 (Pragmatic-to-Strict Translation  $\mathcal{P}2S$ ).** *A legal pragmatic declaration  $\text{pragmatic } c : e E_1 \dots E_n$  is translated to a list of strict constant declarations*

$$c.d_1 : \gamma(F_1) = \gamma(D_1), \dots, c.d_m : \gamma(F_m) = \gamma(D_m)$$

where  $\gamma$  substitutes every  $x_i$  with  $E_i$  and every  $d_j$  with  $c.d_j$  if we have

$$\text{extension } e = \lambda x_1 : E'_1. \dots \lambda x_n : E'_n. \{d_1 : F_1 = D_1, \dots, d_m : F_m = D_m\}$$

and every expression  $E_i$  has type  $E'_i$ .

*Example 3.* Consider the following MMT theories in Figure 4: *HOL* includes the MMT theory *Propositions* and declares a constant  $i$  as the type of individuals. It adds the usual logical connectives and quantifiers – here we only present truth (*true*) and the universal quantifier ( $\forall$ ) – and introduces equality ( $\equiv$ ) on expressions of type  $\alpha$ . Then it includes *Assertion*. This gives *HOL* access to the extensions *axiom* and *theorem*.

*Commutativity* uses *HOL* as its meta-theory and declares a constant  $\circ$  that takes two individuals as arguments and returns an individual. It adds a pragmatic declaration named *comm* that declares the commutativity axiom for  $\circ$  using the axiom extension from *HOL*.

*Commutativity'* is obtained by elaborating *Commutativity* according to Definition 3.

```

theory HOL = {
    meta Propositions
    i      : type
    true   : o
    :
    :
    ∃      : (α → o) → o
    ≡      : α → α → o
    include Assertion
}

theory Commutativity = {
    meta HOL
    ◦           : i → i → i
    pragmatic comm : axiom ∀x : i. ∀y : i. x ◦ y ≡ y ◦ x
}

theory Commutativity' = {
    meta HOL
    ◦           : i → i → i
    comm.c : proof ∀x : i. ∀y : i. x ◦ y ≡ y ◦ x
}

```

**Fig. 4.** A  $\mathcal{P}2\mathcal{S}$  Translation Example

## 4 Representing Extension Principles

Formal mathematical developments can be classified based on whether they follow the axiomatic or the definitional method. The former is common for logics where theories declare primitive constants and axioms. The latter is common for foundations of mathematics where a fixed theory (the foundation) is extended only by defined constants and theorems. In MMT, both the logic and the foundation are represented as a meta-theory  $M$ , and the main difference is that the definitional method does not permit undefined constants in  $M$ -theories.

However, this treatment does not capture conservative extension principles: These are meta-theorems that establish that certain extensions are acceptable even if they are not definitional. We can understand them as intermediates between axiomatic and definitional extensions: They may be axiomatic but are essentially as safe as definitional ones.

To make this argument precise, we use the following definition:

**Definition 4.** *We call the theory family  $\Phi = \lambda x_1 : E'_1. \dots \lambda x_n : E'_n. \{\Sigma\}$  **conservative** for  $M$  if for every  $M$ -theory  $T$  and all  $E_1 : E'_1, \dots, E_n : E'_n$ , every model of  $T$  can be extended to a model of  $T, \gamma(\Sigma)$ , where  $\gamma$  substitutes every  $x_i$  with  $E_i$ .*

*An extension declaration `extension e = Φ` is called **derived** if all constant declarations in  $\Sigma$  have a definiens; otherwise, it is called **primitive**.*

Primitive extension declarations correspond to axiom declarations because they postulate that certain extensions of  $M$  are legal. The proof that they are indeed conservative is a meta-argument that must be carried out as a part of the proof that  $M$  is an adequate MMT representation of the represented formalism. Similarly, derived extension declarations correspond to theorem declarations because their conservativity follows from that of the primitive ones. More precisely: If all primitive extension principles in  $M$  are conservative, then so are all derived ones.

In the following, we will recover built-in extension statements of common representation formats as special cases of our extension declarations. We will follow a *little foundations* paradigm and state every extensions in the smallest theory in which it is meaningful. Using the MMT module system, this permits maximal reuse of extension definitions. Moreover, it documents the (often implicit) foundational assumptions of each extension.

*Implicit Definitions in OMDoc* Implicit definitions of OMDoc 1.2 are captured using the following derived extension declaration. If the theory *ImplicitDefinitions* in Figure 5 is included into a meta-theory  $M$ , then  $M$ -theories may use implicit definitions.

```
theory ImplicitDefinitions = {
  meta Propositions
   $\exists^! : (\alpha \rightarrow o) \rightarrow o$ 
   $\iota : (\alpha \rightarrow o) \rightarrow \alpha$ 
   $\iota_{ax} : proof \exists^!x P x \rightarrow proof P(\iota P)$ 
  extension impldef =  $\lambda\alpha : type. \lambda P : \alpha \rightarrow o. \lambda m : proof \exists^!x : \alpha. P x. \{$ 
     $c : \alpha = \iota P$ 
     $c_{ax} : proof \exists^!x : \alpha. P x$ 
  }
}
```

**Fig. 5.** An Extension for Implicit Definitions

Note that *ImplicitDefinitions* requires two other connectives: A description operator ( $\iota$ ) and a unique existential ( $\exists^!$ ) are needed to express the meaning of an implicit definition. We deliberately assume only those two operators in order to maximize the re-usability of this theory: Using the MMT module system, any logic  $M$  in which these two operators are definable can import the theory *ImplicitDefinitions*.

More specifically, *ImplicitDefinitions* introduces the definite description operator as a new binding operator ( $\iota$ ), and describes its meaning by the axiom  $\exists^!x P(x) \Rightarrow P(\iota P)$  formulated in  $\iota_{ax}$  for any predicate  $P$  on  $\alpha$ . The extension *impldef* permits pragmatic declarations of the form  $f : impldef \alpha P m$ , which defines  $f$  as the unique object which makes the property  $P$  valid. This leads to the well-defined condition that there is indeed such a unique object, which is discharged by the proof  $m$ . The pragmatic-to-strict translation from Section 3 translates the pragmatic declaration  $f : impldef \alpha P m$  to the strict constant declarations  $f.c : \alpha = \iota P$  and  $f.c_{ax} : proof \exists^!x : \alpha. P x$ .

*Mizar-Style Functor Definitions* The Mizar language [TB85] provides a wide (but fixed) variety of special statements, most of which can be understood as conservative extension principles for first-order logic. A comprehensive list of the corresponding extension declarations can be found in [IKR11]. We will only consider one example in Figure 6.

```

theory FunctorDefinitions = {
  meta Propositions
   $\wedge : o \rightarrow o \rightarrow o$ 
   $\Rightarrow : o \rightarrow o \rightarrow o$ 
   $\forall : (\alpha \rightarrow o) \rightarrow \alpha$ 
   $\exists : (\alpha \rightarrow o) \rightarrow \alpha$ 
   $\doteq : \alpha \rightarrow \alpha \rightarrow o$ 
  extension functor =  $\lambda\alpha : \text{type}. \lambda\beta : \text{type}. \lambda means : \alpha \rightarrow \beta \rightarrow o.$ 
     $\lambda existence : proof \forall x : \alpha. \exists y : \beta. means x y.$ 
     $\lambda uniqueness : proof \forall x : \alpha. \forall y : \beta. \forall y' : \beta.$ 
       $means x y \wedge means x y' \Rightarrow y \doteq y'.$ 
    f :  $\alpha \rightarrow \beta$ 
    definitional_theorem : proof  $\forall x : \alpha. means x (f x)$ 
}
}

```

**Fig. 6.** An Extension for Mizar-Style Functor Definitions

The theory *FunctorDefinitions* describes Mizar-style implicit definition of a unary function symbol (called a *functor* in Mizar). This is different from the one above because it uses a primitive extension declaration that is well-known to be conservative. In Mizar, the axiom *definitional\_theorem* is called the *definitional theorem* induced by the implicit definition. Using the extension *functor*, one can introduce pragmatic declarations of the form **pragmatic**  $c : functor A B P E U$  that declare functors  $c$  from  $A$  to  $B$  that are defined by the property  $P$  where  $E$  and  $U$  discharge the induced proof obligations.

*Flexary Extensions* The above two examples become substantially more powerful if they are extended to implicit definitions of functions of arbitrary arity. This is supported by our extension language by using an LF-based logical framework with term sequences and type sequences. We omit the formal details of this framework here for simplicity and refer to [Hor12] instead. We only give one example in Figure 7 that demonstrates the potential.

```

theory CaseBasedDefinitions = {
  meta Propositions
   $\wedge : o^n \rightarrow o$ 
   $\vee^! : o^n \rightarrow o$ 
   $\Rightarrow : o \rightarrow o \rightarrow o$ 
   $\forall : (\alpha \rightarrow o) \rightarrow o$ 
  extension casedef =  $\lambda n : \mathbb{N}. \lambda\alpha : \text{type}. \lambda\beta : \text{type}. \lambda c : (\alpha \rightarrow o)^n.$ 
     $\lambda d : (\alpha \rightarrow \beta)^n. \lambda\rho : proof \forall x : \alpha. \vee^! \left[ c_i x \right]_{i=1}^n. \{$ 
    f :  $\alpha \rightarrow \beta$ 
    ax : proof  $\forall x : \alpha. \wedge \left[ c_i x \Rightarrow (f x) = (d_i x) \right]_{i=1}^n$ 
}
}

```

**Fig. 7.** An Extension for Case-Based Definitions

The theory *CaseBasedDefinitions* introduces an extension that describes the case-based definition of a unary function  $f$  from  $\alpha$  to  $\beta$  that is defined using  $n$  different cases where each case is guarded by the predicate  $c_i$  together with the respective definiens  $d_i$ . Such a definition is well-defined if for all  $x \in \alpha$  exactly one out of the  $c_i x$  is true. Note that these declarations use a special sequence constructor: for example,  $[c_i x]_{i=1}^n$  simplifies to the sequence  $c_1 x, \dots, c_n x$ . Moreover,  $\wedge$  and  $\vee^!$  are flexary connectives, i.e., they take a flexible number of arguments. In particular,  $\vee^!(F_1, \dots, F_n)$  holds if exactly one of its arguments holds.

The pragmatic declaration `pragmatic f : casedef n α β c1 ... cn d1 ... dn ρ` corresponds to the following function definition:

$$f(x) = \begin{cases} d_1(x) & \text{if } c_1(x) \\ \vdots & \vdots \\ d_n(x) & \text{if } c_n(x) \end{cases}$$

*HOL-Style Type Definitions* Due to the presence of  $\lambda$ -abstraction and a description operator in HOL [Chu40], a lot of common extension principles become derivable in HOL, in particular, implicit definitions.

But there is one primitive definition principle that is commonly accepted in HOL-based formalizations of the definitional method: A Gordon/HOL type definition [Gor88] introduces a new type that is axiomatized to be isomorphic to a subtype of an existing type. This cannot be expressed as a derivable extension because HOL does not use subtyping.

```
theory Types = {
  meta Propositions
  ∀ : (α → o) → o
  ∃ : (α → o) → o
  ≡ : (α → α) → o
  extension typedef = λα : type. λA : α → o. λP : proof ∃x : α. A x. {
    T           : type
    Rep         : T → α
    Abs         : α → T
    Rep'        : proof ∀x : T. A (Rep x)
    Rep-inverse : proof ∀x : T. Abs (Rep x) ≡ x
    Abs-inverse : proof ∀x : α. A x ⇒ Rep (Abs x) ≡ x
  }
}
```

**Fig. 8.** An Extension for HOL-Style Type Definitions

The theory *Types* in Figure 8, formalizes this extension principle. Our symbol names follow the implementation of this definition principle in Isabelle/HOL [NPW02]. Pragmatic declarations of the form `pragmatic t : typedef α A P` introduce a new non-empty type  $t$  isomorphic to the predicate  $A$  over  $\alpha$ . Since all HOL-types must be non-empty, a proof  $P$  of the non-emptiness of  $A$  must be

supplied. More precisely, it is translated to the following strict constant declarations:

- $t.T : \text{type}$  is the new type that is being defined,
- $t.\text{Rep} : t.T \rightarrow \alpha$  is an injection from the new type  $t.T$  to  $\alpha$ ,
- $t.\text{Abs} : \alpha \rightarrow t.T$  is the inverse of  $t.\text{Rep}$  from  $\alpha$  to the new type  $t.T$ ,
- $t.\text{Rep}'$  states that the property  $A$  holds for any term of type  $t.T$ ,
- $t.\text{Rep\_inverse}$  states that the injection of any element of type  $t.T$  to  $\alpha$  and back is equal to itself,
- $t.\text{Abs\_inverse}$  states that if an element satisfies  $A$ , then injecting it to  $t.T$  and back is equal to itself.

HOL-based proof assistants implement the type definition principle as a built-in statement. They also often provide further built-in statements for other definition principles that become derivable in the presence of type definitions, e.g., a definition principle for record types. For example, in Isabelle/HOL [NPW02], HOL is formalized in the Pure logic underlying the logical framework Isabelle [Pau94]. But because the type definition principle is not expressible in Pure, it is implemented as a primitive Isabelle feature that is only active in Isabelle/HOL.

## 5 Syntax Extensions and Surface Languages

Our definitions from Section 3 permit pragmatic *abstract* syntax, which is elaborated into strict abstract syntax. For human-oriented representations, it is desirable to complement this with similar extensions of pragmatic *concrete* syntax. While the pragmatic-to-strict translation at the abstract syntax level is usually non-trivial and therefore not invertible, the corresponding translation at the concrete syntax level should be compositional and bidirectional.

### 5.1 OMDoc Concrete Syntax for EL Declarations

First we extend OMDoc with concrete syntax that exactly mirrors the abstract syntax from Section 3. The declaration `extension e = λx1 : E1. . . λxn : En. {Σ}` is written as

```
<extension name="e">
  <parameter name="x1">[E1]</parameter>
  .
  <parameter name="xn">[En]</parameter>
  <theory>
    [Σ]
  </theory>
</extension>
```

Here we use the box notation  $\boxed{A}$  to gloss the XML representation of an entity  $A$  given in abstract syntax.

Similarly, the pragmatic declaration `pragmatic c : e E1 ... En` is written as

```
<pragmatic name="c" extension="⟨M⟩?e">
   $\boxed{E_1} \dots \boxed{E_n}$ 
</pragmatic>
```

Here  $⟨M⟩$  is the meta-theory in which  $e$  is declared so that  $⟨M⟩?e$  is the MMT URI of the extension.

*Example 4.* For the implicit definitions discussed in Section 3, we use the extension *impldef* from Figure 5, which we assume has namespace URI  $⟨U⟩$ . If  $\rho$  is a proof of unique existence for an  $f$  such that  $f' = f \wedge f(0) = 1$ , then the exponential function is defined in XML by

```
<pragmatic name="exp" extension="⟨U⟩?ImplicitDefinitions?impldef">
   $\boxed{\lambda f. f' = f \wedge f(0) = 1} \boxed{\rho}$ 
</pragmatic>
```

## 5.2 Pragmatic Surface Syntax

OMDoc is mainly a machine-oriented interoperability format, which is not intended for human consumption. Therefore, the EL-isomorphic syntax introduced is sufficient in principle – at least for the formal subset of OMDoc we have discussed so far.

OMDoc is largely written in the form of “surface languages” – domain-specific languages that can be written effectively and transformed to OMDoc in an automated process. For the formal subset of OMDoc, we use a MMT-inspired superset of the Twelf/LF [PS99; HHP93] syntax, and for informal OMDoc we use *STEX* [Koh08], a semantic extension of *TEX/LATEX*.

For many purposes like learning the surface language or styling OMDoc documents, **pragmatic surface syntax**, i.e., a surface syntax that is closer to the notational conventions of the respective domain, has great practical advantages. It is possible to support, i.e., generate and parse, pragmatic surface syntax by using the macro/scripting framework associated with most representation formats.

For instance, we can regain the XML syntax familiar from OMDoc 1.2 via notation definitions that transform between pragmatic elements and the corresponding OMDoc 1.2 syntax. For Twelf/LF, we would extend the module system preprocessor, and for Isabelle we would extend the SML-based syntax/parsing subsystem. We have also extended *STEX* as an example of a semi-formal surface language. Here we used the macro facility of *TEX* as the computational engine. We conjecture that most practical surface languages for MKM can be extended similarly.

These translations proceed in two step. Firstly, pragmatic surface syntax is translated into our pragmatic MMT syntax. Our language is designed to make this step trivial: in particular, it does not have to look into the parameters used in a pragmatic surface declaration. Secondly, pragmatic MMT syntax is type-checked and, if desired, translated into strict MMT syntax. All potentially difficult semantic analysis is part of this second step. This design makes it very easy for users to introduce their own pragmatic surface syntax.

## 6 Conclusion & Future Work

In this paper, we proposed a general statement-level extension mechanism for MKM formats powered by the notion of theory families. Starting with MMT as a core language, we are able to express most of the pragmatic language features of OMDoc 1.2 as instances of our new extension primitive. Moreover, we can recover extension principles employed in languages for formalized mathematics including the statements employed for conservative extensions in Isabelle/HOL and Mizar. We have also described a principle how to introduce corresponding pragmatic concrete syntax.

The elegance and utility of the extension language is enhanced by the modularity of the OMDoc 2 framework, whose meta-theories provide the natural place to declare extensions: the scoping rules of the MMT module system supply the justification and intended visibility of statement-level extensions. In our examples, the Isabelle/HOL and Mizar extensions come from their meta-logics, which are formalized in MMT.

We also expect our pragmatic syntax to be beneficial in system integration because it permit interchanging documents at the pragmatic MMT level. For example, we can translate implicit definitions of one system to those of another system even if – as is typical – the respective strict implementations are very different.

For full coverage of OMDoc 1.2, we still need to capture abstract data types and proofs; the difficulties in this endeavor lie not in the extension framework but in the design of suitable meta-logics that justify them. For OMDoc-style proofs, the  $\bar{\lambda}\mu\tilde{\nu}$ -calculus has been identified as suitable [ASC06], but remains to be encoded in MMT. For abstract data types we need a  $\lambda$ -calculus that can reflect signatures into (inductive) data types; the third author is currently working on this.

The fact that pragmatic extensions are declared in meta-theories points towards the idea that OMDoc metadata and the corresponding metadata ontologies [LK09] are actually meta-theories as well (albeit at a somewhat different level); we plan to work out this correspondence for OMDoc 2.

Finally, we observe that we can go even further and interpret the feature of definitions that is primitive in MMT as pragmatic extensions of an even more foundational system. Then definitions  $c : E = E'$  become pragmatic notations for a declaration  $c : E$  and an axiom  $c = E'$ , where  $=$  is an extension symbol introduced in a meta-theory for equality. Typing can be handled similarly. This

would also permit introducing other modifiers in declarations such as `<` for subtype declarations.

## References

- [ASC06] Serge Autexier and Claudio Sacerdoti Coen. “A Formal Correspondence Between OMDoc with Alternative Proofs and the  $\bar{\lambda}\mu\tilde{\mu}$ -calculus”. In: *Mathematical Knowledge Management (MKM)*. Ed. by Jon Borwein and William M. Farmer. LNAI 4108. Springer Verlag, 2006, pp. 67–81.
- [Aus+10] Ron Ausbrooks et al. *Mathematical Markup Language (MathML) Version 3.0*. W3C Recommendation. World Wide Web Consortium (W3C), 2010. URL: <http://www.w3.org/TR/MathML3>.
- [BC04] Y. Bertot and P. Castéran. *Coq’Art: The Calculus of Inductive Constructions*. Springer, 2004.
- [Bus+04a] S. Buswell et al. *The Open Math Standard, Version 2.0*. Tech. rep. See <http://www.openmath.org/standard/om20>. The Open Math Society, 2004.
- [Bus+04b] Stephen Buswell et al. *The Open Math Standard, Version 2.0*. Tech. rep. The OpenMath Society, 2004. URL: <http://www.openmath.org/standard/om20>.
- [Chu40] A. Church. “A Formulation of the Simple Theory of Types”. In: *Journal of Symbolic Logic* 5.1 (1940), pp. 56–68.
- [Gor88] M. Gordon. “HOL: A Proof Generating System for Higher-Order Logic”. In: *VLSI Specification, Verification and Synthesis*. Ed. by G. Birtwistle and P. Subrahmanyam. Kluwer-Academic Publishers, 1988, pp. 73–128.
- [HHP93] R. Harper, F. Honsell, and G. Plotkin. “A framework for defining logics”. In: *Journal of the Association for Computing Machinery* 40.1 (1993), pp. 143–184.
- [Hor12] F. Horozal. “Logic Translations with Declaration Patterns”. <https://svn.kwarc.info/repos/fhorozal/pubs/patterns.pdf>. 2012.
- [IKR11] M. Iancu, M. Kohlhase, and F. Rabe. *Translating the Mizar Mathematical Library into OMDoc format*. Tech. rep. KWARC Report 01/11. Jacobs University Bremen, 2011.
- [Koh06] Michael Kohlhase. *OMDOC – An open markup format for mathematical documents [Version 1.2]*. LNAI 4180. Springer Verlag, Aug. 2006. URL: <http://omdoc.org/pubs/omdoc1.2.pdf>.
- [Koh08] Michael Kohlhase. “Using LATEX as a Semantic Markup Format”. In: *Mathematics in Computer Science* 2.2 (2008), pp. 279–304. URL: <https://svn.kwarc.info/repos/stex/doc/mcs08/stex.pdf>.
- [LK09] Christoph Lange and Michael Kohlhase. “A Mathematical Approach to Ontology Authoring and Documentation”. In: *MKM/Calculemus Proceedings*. Ed. by Jacques Carette et al. LNAI 5625. Springer Verlag, July 2009, pp. 389–404. ISBN: 978-3-642-02613-3.

- URL: <http://kwarc.info/kohlhase/papers/mkm09-omdoc4onto.pdf>.
- [NPW02] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.
- [ORS92] S. Owre, J. Rushby, and N. Shankar. “PVS: A Prototype Verification System”. In: *11th International Conference on Automated Deduction (CADE)*. Ed. by D. Kapur. Springer, 1992, pp. 748–752.
- [Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*. Vol. 828. Lecture Notes in Computer Science. Springer, 1994.
- [PS99] F. Pfenning and C. Schürmann. “System Description: Twelf - A Meta-Logical Framework for Deductive Systems”. In: *Lecture Notes in Computer Science* 1632 (1999), pp. 202–206.
- [Rab08] F. Rabe. *The MMT System*. see <https://trac.kwarc.info/MMT/>. 2008.
- [RK11] F. Rabe and M. Kohlhase. “A Scalable Module System”. see <http://arxiv.org/abs/1105.0548>. 2011.
- [TB85] A. Trybulec and H. Blair. “Computer Assisted Reasoning with MIZAR”. In: *Proceedings of the 9th International Joint Conference on Artificial Intelligence*. Ed. by A. Joshi. 1985, pp. 26–28.
- [Wen99] M. Wenzel. “Isar - A Generic Interpretative Approach to Readable Formal Proof Documents”. In: *Theorem Proving in Higher Order Logics*. Ed. by Y. Bertot et al. Vol. 1690. Springer, 1999, pp. 167–184.

# Representing Isabelle in LF

Florian Rabe

Jacobs University Bremen

f.rabe@jacobs-university.de

LF has been designed and successfully used as a meta-logical framework to represent and reason about object logics. Here we give a representation of the Isabelle logical framework in LF using the recently introduced module system for LF. The major novelty of our approach is that we can naturally represent the advanced Isabelle features of type classes and locales.

Our representation of type classes relies on a feature so far lacking in the LF module system: morphism variables and abstraction over them. While conservative over the present system in terms of expressivity, this feature is needed for a representation of type classes that preserves the modular structure. Therefore, we also design the necessary extension of the LF module system.

## 1 Introduction

Both Isabelle and LF were developed at roughly the same time to provide formal proof theoretic frameworks in which object logics can be defined and studied. Both use the Curry-Howard correspondence to represent the proofs of the object logic as terms of the meta-logic.

Isabelle ([13], [14]) is based on intuitionistic higher-order logic ([2]) with shallow polymorphism and was designed as a generic LCF-style interactive theorem prover. LF ([6]) is the corner of the  $\lambda$ -cube ([1]) that extends simple type theory with dependent function types and is inspired by the judgments-as-types methodology ([10]). We will work with the Twelf implementation of LF ([16]).

It is straightforward to represent Isabelle’s underlying logic as an object logic of LF (see, e.g., [6]). However, Isabelle provides a number of advanced features that go beyond the base logic and that cannot be easily represented in other systems. These include in particular a module system ([9], [5]) and a structured proof language ([11]).

Recently, we gave a module system for LF in [18]. We wanted to choose primitive notions that are so simple that they admit a completely formal semantics. While such formal semantics are commonplace for type theories – in the form of inference systems – they quickly get very complex for module systems on top of type theories. At the same time these primitives should be expressive enough to admit natural representations of modular design patterns. Here by “natural”, we mean that we are willing to accept lossy (in the sense of being non-invertible) encodings of modular specifications as long as their modular structure of sharing and reuse is preserved.

In this paper we give such a representation of the Isabelle module system in the LF module system. The main idea of the encoding is that all modules of Isabelle (theories, locales, type classes) are represented as LF signatures, and that all relations between Isabelle modules (imports, sublocales, interpretations, subclasses, instantiations) are represented as LF signature morphisms.

Thus, our contribution is two-fold. Firstly, we validate the design of the LF module system by showing that it provides just the right primitives needed to represent the Isabelle module system. Actually, before arriving at that conclusion we identify one feature that we have to add to the LF module system: abstraction over morphisms. And secondly, the encoding of Isabelle in LF is useful to integrate Isabelle

and LF specifications. Moreover, for researchers familiar with LF but not with Isabelle, this paper can complement the Isabelle documentation with an LF-based perspective on the foundations of Isabelle.

In Sect. 2, we will repeat the basics of Isabelle and LF to make the paper self-contained. In Sect. 3, we extend the LF module system with abstraction over morphisms. Then we give our representation in Sect. 4.

## 2 Preliminaries

### 2.1 Isabelle

Isabelle is a grown and widely used system, which has led to a rich ontology of Isabelle declarations. We will only consider the core and module system declarations in this paper. And even among those, we will restrict attention to a proper subset of Isabelle’s power.

For the purposes of this paper, we make some minor adjustments for simplicity and consider Isabelle’s language to be generated by the grammar in Fig. 1. Here | and \* denote alternative and repetition, and we use special fonts for *nonterminals* and keywords.

|                       |   |
|-----------------------|---|
| <i>theory</i>         | ::= theory <i>name</i> imports <i>name</i> * begin <i>thycont</i> end   |
| <i>thycont</i>        | ::= (locale   sublocale   interpretation  <br>  class   instantiation   thysymbol)*   |
| <i>locale</i>         | ::= locale <i>name</i> = ( <i>name</i> : <i>instance</i> )* for <i>locsymbol</i> * + <i>locsymbol</i> *                                     |
| <i>sublocale</i>      | ::= sublocale <i>name</i> < <i>instance proof</i> *   |
| <i>interpretation</i> | ::= interpretation <i>instance proof</i> *  |
| <i>instance</i>       | ::= <i>name</i> where <i>namedinst</i> *  |
| <i>class</i>          | ::= class <i>name</i> = <i>name</i> * + <i>locsymbol</i> *  |
| <i>instantiation</i>  | ::= instantiation <i>type</i> :: ( <i>name</i> *) <i>name</i> begin <i>locsymbol</i> * <i>proof</i> * end                                   |
| <i>thysymbol</i>      | ::= consts <i>con</i>   defs <i>def</i>   axioms <i>ax</i>   lemma <i>lem</i><br>  typedecl   types <i>types</i>                            |
| <i>locsymbol</i>      | ::= fixes <i>con</i>   defines <i>def</i>   assumes <i>ax</i>   lemma <i>lem</i>  |
| <i>con</i>            | ::= <i>name</i> :: type   |
| <i>def</i>            | ::= <i>name</i> : <i>name</i> <i>var</i> * $\equiv$ <i>term</i>   |
| <i>ax</i>             | ::= <i>name</i> : Prop  |
| <i>lem</i>            | ::= <i>name</i> : Prop <i>proof</i>   |
| <i>typedecl</i>       | ::= ( <i>var</i> *) <i>name</i>   |
| <i>types</i>          | ::= ( <i>var</i> *) <i>name</i> = type  |
| <i>namedinst</i>      | ::= <i>name</i> = <i>term</i>   |
| <i>type</i>           | ::= <i>var</i> :: <i>name</i>   <i>name</i>   ( <i>type</i> , ..., <i>type</i> ) <i>name</i>   <i>type</i> $\Rightarrow$ <i>type</i>   prop |
| <i>term</i>           | ::= <i>var</i>   <i>name</i>   <i>name term</i> *   $\lambda$ ( <i>var</i> :: type)*. <i>term</i>   |
| <i>Prop</i>           | ::= Prop $\Longrightarrow$ Prop   $\wedge$ ( <i>var</i> :: type)*. Prop   <i>term</i> $\equiv$ <i>term</i>                                  |
| <i>proof</i>          | ::= a proof term  |
| <i>name, var</i>      | ::= identifier  |

Figure 1: Simplified Isabelle Grammar

A *theory* is a named group of declarations. Theories may use imports to import other theories, which yields a simple module system. Within theories, *locale* and type *class* declarations provide further

sources of modularity. Theories, locales, and type classes may be related using a number of declarations as described below.

The *core declarations* occurring in theories (*thysymbol*) and locales (*locsymbol*) are quite similar. *consts* and *fixes* declare typed constants  $c :: \tau$ . *defs* and *defines* declare definitions for a constant  $f$  taking  $n$  arguments as  $f\_def : f x_1 \dots x_n \equiv t$  where  $t$  is a term in the variables  $x_i$ . *axioms* and *assumes* declare named axioms  $a$  asserting a proposition  $\varphi$  as  $a : \varphi$ . *lemma* declares a named lemma  $l$  asserting  $\varphi$  with proof  $P$  as  $l : \varphi P$ .

Furthermore, in theories, *typeddecl* declares  $n$ -ary type operators  $t$  as  $(\alpha_1, \dots, \alpha_n) t$ , and similarly *types* declares an abbreviation  $t$  for a type  $\tau$  in the variables  $\alpha_i$  as  $(\alpha_1, \dots, \alpha_n) t = \tau$ . Locales do not contain type declarations. However, they may declare new types indirectly by declaring constants whose types have free type variables, e.g.,  $\circ : \alpha \Rightarrow \alpha \Rightarrow \alpha$  in a locale for groups. References to these types are made indirectly using type inference, e.g., if there is another constant  $e : \beta$ , then an axiom  $x \circ e = x$  enforces that  $\alpha$  and  $\beta$  refer to the same type.

The constant declarations within a locale serve as parameters that can be instantiated. The intuition is that a locale *instance*  $loc$  where  $\sigma$  takes the locale with name  $loc$  and translates it into a new context (which can be a theory or another locale). Here  $\sigma$  is a list of parameter instantiations (*namedinst*) of the form  $c = t$  instantiating the parameter  $c$  of  $loc$  with the term  $t$  in that new context.

Locale instances are used in two places. Firstly, locale declarations may contain a list of instances used to inherit from other locales. In a locale declaration

```
locale loc = ins1 : loc1 where σ1 ... insn : locn where σn for Σ + Σ'
```

the new locale  $loc$  inherits via  $n$  named instances: Instance  $ins_i$  inherits from the locale  $loc_i$  via the list of parameter instantiations  $\sigma_i$ .  $\Sigma$  and  $\Sigma'$  declare the core declarations of the locale.

The set of constant declarations of the locale is defined as follows: (i) The declarations in  $\Sigma$  logically precede the instances, i.e., are available in  $\sigma_i$  and  $\Sigma'$ . (ii) A copy of the declarations of each  $loc_i$  translated by  $\sigma_i$  is available in each  $\sigma_j$  for  $j > i$  and in  $\Sigma'$ ; the names  $ins_i$  serve as qualifiers to resolve name clashes if two declarations of the same name are present. (iii) The declarations in  $\Sigma'$  are only available in  $\Sigma'$ .

The  $\sigma_i$  do not have to instantiate all parameters of  $loc_i$  – parameters that are not instantiated become parameters of  $loc$ . Thus, the parameters of  $loc$  consist of the not-instantiated parameters of the  $loc_i$  and the constants declared in  $\Sigma$  and  $\Sigma'$ .

Secondly, a declaration *sublocale*  $loc' < loc$  where  $\sigma \pi$  postulates a translation from  $loc$  to  $loc'$ , which maps the parameters of  $loc$  according to  $\sigma$ . The axioms and definitions of  $loc$  induce proof obligations over  $loc'$  that must be discharged by giving a list  $\pi$  of proofs. If all proof obligations are discharged, all theorems about  $loc$  can be translated to yield theorems about  $loc'$ , and Isabelle does that automatically. A locale *interpretation* is very similar to a *sublocale*. The difference is that all  $loc$  expressions are translated into the current theory rather than into a second locale.

The concepts of locales and *type classes* have recently been aligned ([5]) and in particular type classes are also locales. But the syntax still reflects their different use cases. A type class is a locale inheriting only from other type classes and only without parameter instantiations. Thus, the locale syntax can be simplified to *class*  $C = C_1 \dots C_n + \Sigma$  where  $C$  inherits from the  $C_i$ . All declarations in  $\Sigma$  may refer to at most one type variable, which can be assumed to be of the form  $\alpha :: C$ . The intuition is that  $\Sigma$  provides operations  $c_1, \dots, c_n$  that are polymorphic in the parametric type  $\alpha$  and axioms about them.

An instance of a type class is a tuple  $(\tau, c_1\_def, \dots, c_n\_def)$  where  $\tau$  is a type and  $c_i\_def$  is a definition for  $c_i$  at the type  $\tau$ . Because every  $c_i$  can only have one definition per type, the definitions can

be inferred from the context and be dropped from the notation; then a type class can be seen as a unary predicate on types  $\tau$ . Type class instantiations are of the form

```
instantiation t :: (C1, ..., Cn)C begin Σ π end
```

where  $t$  is an  $n$ -ary type operator, i.e., a type with  $n$  free type variables  $\alpha_i$ .  $\Sigma$  contains the definitions for the operations of  $C$  at the type  $(\alpha_1, \dots, \alpha_n)t$  in terms of the operations of the instances  $\alpha_i :: C_i$ . This creates proof obligations for the axioms of  $C$ , and we assume that all the needed proofs are provided as a list  $\pi$ . The semantics is that if  $\tau_i :: C_i$  are type class instances, then so is  $(\tau_1, \dots, \tau_n)t :: C$ . Note that this includes base types for  $n = 0$ .

*Example 1.* The following sketches type class for ordering ands semilattices with universe  $\alpha$ , ordering  $\leq$ , and infimum  $\sqcap$  (where we omit inferable types and write  $\cdot$  for empty lists):

```
class order = · + ≤:: α ⇒ α ⇒ prop
class semlat = order + ⊓:: α ⇒ α ⇒ α
locale lat = inf : semlat where ·
                     sup : semlat where ≤ = λxλy. y inf. ≤ x for · + .
```

Here the omitted axioms in *semlat* would enforce that the type variables  $\alpha$  in the types of  $\leq$  and  $\sqcap$  refer to the same type. Then a locale for lattices is obtained by using two named instances of a semilattice where the second one flips the ordering. The parameters of *lat* are *inf*,  $\leq$  (the ordering), *inf*. $\sqcap$  (the infimum), and *sup*. $\sqcap$  (the supremum), but not *sup*. $\leq$  which is instantiated.

Finally the *inner syntax* for *terms*, *types*, *propositions*, and *proof* terms – also called the Pure language – is given by an intuitionistic higher-order logic with shallow polymorphism. Types are formed from type variables  $\alpha :: C$  for type classes  $C$ , base types, type operator applications, function types, and the base type *prop* of propositions. Type class instances of the form  $\tau :: C$  are formed from type variables  $\alpha :: C$  and type operator applications  $(\tau_1, \dots, \tau_n)t$  for a corresponding instantiation  $t :: (C_1, \dots, C_n)C$  and type class instances  $\tau_i :: C_i$ . We will assume every type to be a type class instance by using the special type class *Type* of all types.

Terms are formed from variables, typed constants, application, and lambda abstraction. Constants may be polymorphic in the sense that their types may contain free type variables. When a polymorphic constant is used, Isabelle automatically infers the type class instances for which the constant is used. Propositions are formed from implication, universal quantification over any type, and equality on any type.

We always assume that all types are fully reconstructed. Similarly, we cover neither the Isar proof language nor tactic invocations and simply assume proof terms from Pure’s natural deduction calculus.

## 2.2 LF

The non-modular declarations in an LF signature are *kinded type family* symbols  $a : K$  and *typed constants*  $c : A$ . Both may carry definitions, e.g.,  $c : A = t$  introduces  $c$  as an abbreviations for  $t$ . The objects of Twelf are *kinds*  $K$ , *kinded type families*  $A : K$ , and *typed terms*  $t : A$ . *type* is the kind of types, and  $A \rightarrow \text{type}$  is the kind of type families indexed by terms of type  $A$ . We use Twelf notation for binding and application: The type  $\Pi_{x:A}B(x)$  of dependent functions taking  $x : A$  to an element of  $B(x)$  is written  $\{x : A\}B$ , and the function term  $\lambda_{x:A}t(x)$  taking  $x : A$  to  $t(x)$  is written  $[x : A]t$ . We write  $A \rightarrow B$  instead of  $\{x : A\}B$  if  $x$  does not occur in  $B$ , and we will also omit the types of bound variables if they can be inferred.

The Twelf *module system* ([19]) is based on the notions of signatures and signature morphisms ([8]). Given two signatures  $\text{sig } S = \{\Sigma\}$  and  $\text{sig } T = \{\Sigma'\}$ , a signature morphism from  $S$  to  $T$  is a type/kind-preserving map  $\mu$  of  $\Sigma$ -symbols to  $\Sigma'$ -expressions. Thus,  $\mu$  maps every constant  $c : A$  of  $\Sigma$  to a term  $\mu(c) : \bar{\mu}(A)$  and every type family symbol  $a : K$  to a type family  $\mu(a) : \mu(K)$ . Here,  $\mu(-)$  doubles as the homomorphic extension of  $\mu$ , which maps closed  $\Sigma$ -expressions to closed  $\Sigma'$  expressions. Signature morphisms preserve typing and kinding, i.e., if  $\vdash_{\Sigma} E : F$ , then  $\vdash_{\Sigma'} \mu(E) : \mu(F)$ .

*Signature* declarations are straightforward:  $\text{sig } T = \{\Sigma\}$ . Signatures may be nested and may include other signatures. Basic *morphisms* are given explicitly as  $\{\sigma : S \rightarrow T\}$ , and composed morphisms are formed from basic morphisms, identity, composition, and two kinds of named morphisms: views and structures.<sup>1</sup>

We will use the following grammar where the structure identifiers  $T.s$  and the symbol identifiers  $S.c^{\mu}$  and  $S.a^{\mu}$  are described below:

|                  |  |
|------------------|--|
| Signature graphs | $G ::= \cdot   G, \text{sig } T = \{\Sigma\}   G, \text{view } v : S \rightarrow T = \mu$  |
| Signatures       | $\Sigma ::= \cdot   \Sigma, \text{sig } T = \{\Sigma\}   \Sigma, \text{include } S$<br>$  \Sigma, \text{struct } s : S = \{\sigma\}   \Sigma, c : A [= t]   \Sigma, a : K [= A]$ |
| Morphisms        | $\sigma ::= \cdot   \sigma, \text{struct } s := \mu   \sigma, c := t   \sigma, a := A$   |
| Compositions     | $\mu ::= T.s   \{\sigma : S \rightarrow T\}   v   id   incl   \mu \mu$   |
| Contexts         | $\Gamma ::= \cdot   \Gamma, x : A$   |
| Kinds            | $K ::= \text{type}   A \rightarrow K$  |
| Type families    | $A ::= S.a^{\mu}   A t   \{x : A\} A$  |
| Terms            | $t ::= S.c^{\mu}   x   [x : A] t   t t$  |

Modular LF uses the following judgments for well-formed syntax:

|  |   |
|--|---|
| $\triangleright G$                       | well-formed signature graphs                            |
| $G \triangleright \mu : S \rightarrow T$ | morphism between signatures $S$ and $T$ declared in $G$ |
| $G \triangleright_T \Gamma \text{ Ctx}$  | contexts for signature $T$                              |
| $G; \Gamma \triangleright_T E : E'$      | typing in signature $T$                                 |

The judgment for signature graphs mainly formalizes uniqueness of identifiers and type-preservation of morphisms based on the typing judgment for expressions. The judgments for contexts and typing are essentially the same as for non-modular LF except that the identifiers available in signature  $T$  and their types are determined by the module system. Therefore, we only describe the judgments for identifiers and morphisms and refer to [18] for details.

**Morphisms** First of all, to simplify the language, we will employ the following condition on all morphisms from  $S$  to  $T$ :  $T$  must include all signatures that  $S$  includes, and if  $S$  includes  $R$ , the application of  $\mu$  to symbols of  $R$  is the identity. In particular, views and structures may only be declared if this condition holds.

Firstly, the semantics of a *structure* declaration  $\text{struct } s : S = \{\sigma\}$  in  $T$  is that it induces (i) for every constant  $c : A$  of  $S$  a constant  $s.c : T.s(A)$  in  $T$ , and (ii) a morphism  $T.s$  from  $S$  to  $T$  that maps every symbol  $c$  of  $S$  to  $s.c$ . Here  $\sigma$  is a partial morphism from  $S$  to  $T$ , and if  $\sigma$  contains  $c := t$ , the constant  $s.c$  is defined as  $t$ . In particular,  $t$  must have type  $T.s(A)$  over  $T$ . The same holds for type family symbols  $a$ . Thus, structures instantiate parametric signatures.

<sup>1</sup>Anonymous morphisms are actually not present in [19]. They are easy to add conceptually, but are a bit harder to add to Twelf as they violate the phase distinction between modular and non-modular syntax kept by the other declarations. We will need them later on.

Because structures are named, a signature may have multiple structures of the same signature, which are all distinct. For example, if  $S$  already contains a structure  $r$  instantiating a third signature  $R$ , then `struct r' : R` in  $T$  leads to the two morphisms  $r'$  and the composition  $S.r.T.s$  from  $R$  to  $T$  and two copies of the constants of  $R$ . Structures may instantiate whole structures at once: If  $T$  declares instead `struct r' : R = {struct r := T.r'}`, then the two copies of  $R$  are shared. More generally,  $\sigma$  may contain instantiations `struct r := μ` for a morphism  $\mu$  from  $R$  to  $T$ , which is equivalent to instantiating every symbol  $c$  of  $R$  with  $\mu(c)$ . Another way to say this is that the diagram on the right commutes.

$$\begin{array}{ccc} R & \xrightarrow{\mu} & T \\ S.r & \searrow & T.s \\ S & \downarrow & \end{array}$$

Secondly, the semantics of *anonymous morphisms*  $\{\sigma : S \rightarrow T\}$  is straightforward. They are well-formed if  $\sigma$  is total and map all constants according to  $\sigma$ . Thirdly, *views*  $v$  are just names given to existing morphisms.

Fourthly, *inclusion*, *identity* and *composition* are defined by

$$\begin{array}{c} \text{sig } T = \{\Sigma\} \text{ in } G \quad \text{include } S \text{ in } \Sigma \qquad \text{incl} \\ \hline G \triangleright \text{incl} : S \rightarrow T \\ \\ \text{sig } T = \{\Sigma\} \text{ in } G \qquad \text{id} \qquad \frac{G \triangleright \mu : R \rightarrow S \quad G \triangleright \mu' : S \rightarrow T}{G \triangleright \mu \mu' : R \rightarrow T} \text{ comp} \end{array}$$

**Identifiers** Defining which symbol identifiers are available in a signature is intuitively easy, but a formal definition can be cumbersome because all included symbols and those induced by structures have to be computed along with their translated types and definitions. Using morphisms and the novel notation  $S.c^\mu$  for symbol identifiers, we can give a very elegant definition:

$$\text{sig } S = \{\Sigma\} \text{ in } G \quad c : E \text{ in } \Sigma \quad G \triangleright \mu : S \rightarrow T \qquad \text{tp} \\ \hline G \triangleright_T S.c^\mu : \mu(E)$$

and similarly for defined symbols and type family constants. The prize to pay is an awkward notation, but we can recover the usual notations as follows:

- *id* yields local symbols, and we write  $c$  instead of  $T.c^{id}$ .
- *incl* yields included symbols, and we write  $S.c$  instead of  $S.c^{incl}$ .
- If  $T$  contains a structure from  $S$ , we have  $G \triangleright T.s : S \rightarrow T$ , and we write  $s.c$  instead of  $S.c^{T.s}$ . Accordingly, we introduce constants  $s.r.c$  for composed morphisms  $S.r.T.s$  from  $R$  to  $T$ , and so on.
- All other identifiers  $T.c^\mu$ , e.g., those where  $\mu$  contains views or anonymous morphisms, are reduced to one of the other cases by applying the morphism.

**Functors** While views are well-established in logical frameworks based on model theory (see, e.g., [4, 20]), they are an unusual feature in proof theoretical frameworks. (In fact, the LF module system has been criticized for using views instead of functors or even – in light of [7] – for using either one rather than only structures.) Therefore, we quickly describe how functors are a derived notion in the presence of views and anonymous morphisms.

Assume a functor  $F$  from  $S$  to  $T$ . Its input is a structure  $s$  instantiating  $S$ , and its output is a list  $\tau$  of instantiations for the symbols of  $T$ . Here  $\tau$  may refer to the symbols induced by  $s$ . We can write this in LF as

$$\text{sig } F_0 = \{\text{struct } s : S\} \quad \text{view } F : T \rightarrow F_0 = \{\tau\}$$

Now given a theory  $D$ , we can understand instances of a signature  $S$  over  $D$  as morphisms from  $S$  to  $D$ . This is justified because a morphism from  $S$  to  $D$  realizes every declaration of  $S$  in terms of  $D$ . More generally we can think of morphisms from  $S$  to  $D$  as implementations or models of  $S$  in terms of  $D$ . The application of  $F$  should map instances of  $S$  to instances of  $T$ . Thus, given a morphism  $\mu$  from  $S$  to  $D$ , we can write the application  $F(\mu)$  as the composed morphism

$$F \{\text{struct } s := \mu : F_0 \rightarrow D\}$$

which is indeed a morphism from  $T$  to  $D$ .

### 3 Morphism Variables in LF

We add a feature to the LF module system that permits morphism variables and abstraction over them. For morphism variables, the grammar is extended as follows:

$$\text{Contexts } \Gamma ::= \Gamma, X : S \quad \text{Morphisms } \mu ::= \mu X$$

Due to the presence of morphism variables, the judgment for well-formed morphisms must be amended to depend on the context. Then we can give the typing rules as:

$$\frac{G \triangleright_T \Gamma \text{ Ctx} \quad \text{sig } S = \{\Sigma\} \text{ in } G}{G \triangleright_T \Gamma, X : S \text{ Ctx}} \text{ contmor} \quad \frac{G \triangleright_T \Gamma, X : S \text{ Ctx}}{G; \Gamma, X : S \triangleright X : S \rightarrow T} \text{ morvar}$$

where we retain the restriction on signature inclusions: All signatures included into  $S$  must also be included into  $T$ .

Note that we can understand the signature  $S$  as a (dependent) record type, a morphism  $\mu : S \rightarrow T$  as a record value of type  $S$  visible in the signature  $T$ , and an identifier  $S.c^\mu$  as the projection out of the record type  $S$  at the field  $c$  applied to  $\mu$ . Then  $X$  is simply a variable of record type, and abstraction over morphism variables is straightforward:

$$\text{Type families } A ::= \{X : S\} A \quad \text{Terms } t ::= [X : S] t \mid t \mu$$

and (omitting the obvious  $\Pi$ -rule and the rules for  $\beta$  and  $\eta$ -conversion)

$$\frac{G; \Gamma, X : S \triangleright_T t : A}{G; \Gamma \triangleright_T [X : S] t : \{X : S\} A} \text{ morlam} \quad \frac{G; \Gamma \triangleright_T f : \{X : S\} A \quad G; \Gamma \triangleright \mu : S \rightarrow T}{G; \Gamma \triangleright_T f \mu : A[X/\mu]} \text{ morapp}$$

Here  $t$  and  $A$  may contain occurrences of the morphism variable  $X$ . In particular  $X$  may occur as a morphism argument to some expression, e.g.,  $g X$ , or in an identifier  $S.c^X$ , which we write as  $X.c$  in accordance with our notation for structures.

A crucial feature of the LF module system is that it is conservative: Modular signatures can be elaborated into non-modular ones (essentially by replacing every structure declaration with the induced constant declarations). We want to elaborate morphism variables similarly.

To elaborate  $X : S$ , we can assume that all structures in  $S$  have already been elaborated and that all defined symbols have been removed by expanding definitions, i.e., (up to reordering)  $S$  is of the form  $\text{sig } S = \{\text{include } R_1, \dots, \text{include } R_m, c_1 : B_1, \dots, c_n : B_n\}$ . Then  $[X : S]t$  in a signature  $T$  is elaborated to  $[x_1 : B'_1] \dots [x_n : B'_n]t'$  where for expressions  $E$  over  $S$ , we obtain  $E'$  by replacing every occurrence of  $X$  with the morphism  $\{c_1 := x_1, \dots, c_n := x_n : S \rightarrow T\}$ . (In particular, after using morphism application, the identifiers  $S.c_i^X$  simply become  $x_i$ .)  $\{X : S\}A$  is elaborated accordingly. Finally,  $t\mu$  is elaborated to  $t\mu(S.c_1) \dots \mu(S.c_n)$ .

This extended module system is not conservative over LF:  $S$  may contain type declarations, but LF does not permit abstraction over type variables. But we obtain conservativity if we make the following additional restriction: Contexts  $\Gamma, X : S$  are only well-formed if all type family symbols  $R.a^\mu$  available in  $S$  are included from other signatures, i.e.,  $\mu = \text{incl} \dots \text{incl}$ . Conversely, neither  $S$  nor any signature that  $S$  instantiates may contain type family declarations.

This restriction seems ad hoc but is in fact quite natural. Assume we have LF signatures  $\lceil L \rceil$  and  $\lceil T \rceil$  that represent an object logic  $L$  and a theory  $T$  of  $L$ . Then, typically,  $\lceil L \rceil$  contains type declarations for the syntactic categories and judgments of  $L$  and constant declarations for the logical symbols and inference rules;  $\lceil T \rceil$  includes  $\lceil L \rceil$  and adds constant declarations for the non-logical symbols (sorts, functions, predicates, etc.) and axioms. Thus, our extension lets us abstract over morphisms out of theories but not over morphisms out of object logics. And the former are exactly the morphisms that we are interested in because morphisms out of  $\lceil T \rceil$  can be used to represent models or implementations of  $T$ . In particular, below,  $T$  will be an axiomatic type class and morphisms out of  $\lceil T \rceil$  will be type class instances.

## 4 Representing Isabelle in LF

The representation of Isabelle in LF proceeds in two steps. In a first step, we declare an LF signature *Pure* for the inner syntax of Isabelle. This syntax declares symbols for all primitives that can occur (explicitly or implicitly) in *Pure* expressions. In a second step, every Isabelle expression  $E$  is represented as an LF expression  $\lceil E \rceil$ . Finally we have to justify the adequacy of the encoding.

For the *inner syntax*, the LF signature *Pure* is given in Fig. 4. This is a straightforward intrinsically typed encoding of higher-order logic in LF. Pure types  $\tau$  are encoded as LF-terms  $\lceil \tau \rceil : tp$  and Pure terms  $t :: \tau$  as LF-terms  $\lceil t \rceil : tm \lceil \tau \rceil$ . Using higher-order abstract syntax, the LF function space  $A \rightarrow B$  with  $\lambda$ -abstraction  $[x : A]t$  and application  $f t$  is distinguished from the encoding  $tm(\lceil \sigma \rceil \Rightarrow \lceil \tau \rceil)$  of the Isabelle function space with application  $\lceil f \rceil @ \lceil t \rceil$  and  $\lambda$ -abstraction  $\lambda([x : tm \lceil \tau \rceil] \lceil t \rceil)$ . Pure propositions  $\varphi$  are encoded as LF-terms  $\lceil \varphi \rceil : tm \text{ prop}$ , and derivations of  $\varphi$  as LF-terms of type  $\vdash \lceil \varphi \rceil$ . Where possible, we use the same symbol names in LF as in Isabelle, and we can also mimic most of the Isabelle operator fixities and precedences.

The signature *Pure* only encodes how composed Pure expressions are formed from the atomic ones. The atomic expressions – variables and constants etc. – are added when encoding the outer syntax as LF declarations. For the *non-modular declarations*, this is straightforward, an overview is given in the following table:

```

sig Pure = {
  tp      : type.
  ⇒      : tp → tp → tp.           infix right 0 ⇒.
  tm      : tp → type.           prefix 0 tm.
  λ       : (tm A → tm B) → tm (A ⇒ B).
  @       : tm (A ⇒ B) → tm A → tm B.   infix left 1000 @.

  prop    : tp.
  ∧      : (tm A → tm prop) → tm prop.
  ⇒⇒    : tm prop → tm prop → tm prop.   infix right 1 ⇒⇒.
  ≡      : tm A → tm A → tm prop.   infix none 2 ≡.

  ⊢      : tm prop → type.           prefix 0 ⊢.
  ∧I     : (x : tm A ⊢ (B x)) → ⊢ ∧([x]B x).
  ∧E     : ⊢ ∧([x]B x) → {x : tm A} ⊢ (B x).
  ⇒⇒I   : (⊢ A → ⊢ B) → ⊢ A ⇒⇒ B.
  ⇒⇒E   : ⊢ A ⇒⇒ B → ⊢ A → ⊢ B.
  refl   : ⊢ X ≡ X.
  subs   : {F : tm A → tm B} ⊢ X ≡ Y → ⊢ F X ≡ F Y.
  exten  : {x : tm A} ⊢ (F x) ≡ (G x) → ⊢ λF ≡ λG.
  beta   : ⊢ (λ[x : tm A]F x) @ X ≡ F X.
  eta    : ⊢ λ ([x : tm A]F @ x) ≡ F.

  sig Type = {this : tp.}.
}.

```

Figure 2: LF Signature for Isabelle

| Expression                  | Isabelle                        | LF   |
|-----------------------------|---------------------------------|--|
| base type, type operator    | $(\alpha_1, \dots, \alpha_n) t$ | $t : tp \rightarrow \dots \rightarrow tp \rightarrow tp$ |
| type variable               | $\alpha$                        | $\alpha : tp$  |
| constant                    | $c :: \tau$                     | $c : tm \vdash \tau \dashv$                              |
| variable                    | $x :: \tau$                     | $x : tm \vdash \tau \dashv$                              |
| assumption/axiom/definition | $a : \varphi$                   | $a : \vdash \varphi \dashv$                              |
| theorem                     | $a : \varphi P$                 | $a : \vdash \varphi \dashv = \vdash P \dashv$            |

The main novelty of our encoding is to also cover the *modular declarations*. The basic idea is to represent all high-level scoping concepts as signatures and all relations between them as signature morphisms as in the following table:

| Isabelle  | LF                       |
|---|--------------------------|
| theory, locale, type class                          | signature                |
| theory import                                       | morphism (inclusion)     |
| locale import, type class import                    | morphism (structure)     |
| sublocale, interpretation, type class instantiation | morphism (view)          |
| instance of type class $C$                          | morphism with domain $C$ |

In the following, we give the important cases of the mapping  $\vdash - \dashv$  from Isabelle to LF by induction on

the Isabelle syntax. We occasionally use `color` to distinguish the meta-level symbols (such as `=`) from Isabelle and Twelf syntax such as `=`.

**Theories** Isabelle theories and theory imports are encoded directly as LF-signatures and signature inclusions. The only subtlety is that the LF encodings additionally include our *Pure* signature.

```
theory T imports T1,...,Tn begin Σ end =  
sig T = { include Pure. include T1. ... include Tn. Σ }.
```

where the body  $\Sigma$  of the theory is translated component-wise as described by the respective cases below.

**Type Classes** The basic idea of the representation of Isabelle type classes in LF is as follows: An Isabelle type class  $C$  is represented as an LF signature  $C$  that contains all the declarations of  $C$  and a field  $this : tp$ . All occurrences in  $C$  of the single permitted type variable  $\alpha :: C$  are translated to  $this$  such that  $this$  represents the type that is an instance of  $C$ .

This means that  $\alpha$  is not considered as a type variable but as a type declaration that is present in the type class. This change of perspective is essential to obtain an elegant encoding of type classes.

In particular, the subsignature *Type* of *Pure* represents the type class of all types. Morphisms with domain *Type* are simply terms of type *tp*, i.e., types.

The central invariant of the representation is this: An Isabelle type class instance  $\tau :: C$  is represented as an LF morphism  $\tau :: C$  from  $C$  into the current LF signature that maps the field  $this$  to  $\tau$  and all operations of  $C$  to the encoding of their definitions at  $\tau$ . Thus, in particular,  $\tau :: C(C.this) = \tau$ .

*Example 2* (Continued). The first type class from Ex. I is represented in LF as follows:

```
sig order = {this : tp. ≤: tm(this ⇒ this ⇒ prop)}
```

In general, we represent type classes as follows:

```
class C = C1 ... Cn + Σ = sig C = {this : tp. I1. ... In. Σ }.
```

where  $I_i$  abbreviates `struct insi : Ci = {this := this ρi}` for some fresh names  $ins_i$ . Since one  $this$  is imported from each superclass  $C_i$ , they must be shared using the instantiations  $this := this$ .  $ρ_i$  contains one structure sharing declaration for each type class imported by  $I_i$  that has already been imported by  $I_1, \dots, I_{i-1}$ .

*Example 3* (Continued). The second type class from Ex. I is represented in LF as follows:

```
sig semlat = {this : tp. struct o : order = {this := this}. □ : tm(this ⇒ this ⇒ this)}
```

A type class *instantiation*

```
instantiation t :: (C1, ..., Cn)C begin Σ π end
```

is represented as an LF functor taking instances of the  $C_i$  and returning an instance of  $C$ . We represent such a functor as a signature

```
sig v = {struct α1 : C1 ... struct αn : Cn}.
```

collecting the input and a view

```
view v' : C → v = {this := t α1.this ... αn.this Σ π}.
```

describing the output.  $v'$  must map the field  $tp$  of  $C$  to the type that is an instance of  $C$ . This type is obtained by applying  $t$  to the argument types that are instances of the  $C_i$ . In Isabelle, this is  $t \alpha_1 \dots \alpha_n$ ; in LF, each  $\alpha_i$  is a structure of  $C_i$ , thus we use the induced constants  $\alpha_i.this$ .

Here  $\lceil \Sigma \rceil$  gives instantiations that map every constant of  $C$  to its definition in terms of the  $\alpha_i$ . Similarly,  $\lceil \pi \rceil$  maps every axiom of  $C$  to its proof. Note how – in accordance with the Curry-Howard representation of proofs as terms – the discharging of proof obligations is just a special case of instantiating a constant.

Now assume type class instances  $\tau_i :: C_i$  encoded as morphisms  $\lceil \tau_i :: C_i \rceil : C_i \rightarrow S$  (where  $S$  is the current signature). The encoding  $\lceil (\tau_1, \dots, \tau_n) t :: C \rceil : C \rightarrow S$  is obtained as the composition

$$v' \{ \text{struct } \alpha_1 := \lceil \tau_1 :: C_1 \rceil \dots \text{struct } \alpha_n := \lceil \tau_n :: C_n \rceil : v \rightarrow S \}.$$

Clearly this is a morphism from  $C$  to  $S$ ; we need to show that indeed

$$\lceil (\tau_1, \dots, \tau_n) t :: C \rceil (C.this) = \lceil t \tau_1 \dots \tau_n \rceil.$$

This holds because

$$\begin{aligned} \lceil (\tau_1, \dots, \tau_n) t :: C \rceil (C.this) &= \{ \dots \text{struct } \alpha_i := \lceil \tau_i :: C_i \rceil \dots \} (v'(C.this)) \\ &= \{ \dots \text{struct } \alpha_i := \lceil \tau_i :: C_i \rceil \dots \} (t \alpha_1.this \dots \alpha_n.this) = \\ t \lceil \tau_1 :: C_1 \rceil (C_1.this) \dots \lceil \tau_n :: C_n \rceil (C_n.this) &= t \lceil \tau_1 \rceil \dots \lceil \tau_n \rceil = \lceil t \tau_1 \dots \tau_n \rceil \end{aligned}$$

We have the general result that the Isabelle subclass relation  $C \subseteq D$  holds iff there is an LF morphism  $i : D \rightarrow C$ . Then if the type class instance  $\tau :: C$  (occurring in some theory or locale  $S$ ) is represented as a morphism  $\lceil \tau :: C \rceil : C \rightarrow S$ , the type class instance  $\tau :: D$  is represented as  $\lceil \tau :: D \rceil = i \lceil \tau :: C \rceil$ . Isabelle has the limitation that there can be at most one way how  $C$  is a subclass of  $D$ , which has the advantage that  $i$  is unique and can be dropped from the notation. In LF, we have to make it explicit.

*Example 4 (Continued).* The trivial subclass relation  $order \subseteq Type$  is represented by the morphism  $i = \{this := this : Type \rightarrow order\}$ . The subclass relation  $semlat \subseteq order$  is represented by the morphism  $semlat.o$ . Finally, the morphism  $i semlat.o = \{this := this : Type \rightarrow semlat\}$  represents  $semlat \subseteq Type$ .

**Locales** Similarly to type classes, Isabelle *locales* are encoded as subsignatures: For example,

$$\text{locale } loc = ins_1 : loc_1 \text{ where } \sigma_1 \text{ for } \Sigma + \Sigma'$$

is encoded as the LF signature

$$\text{sig } loc = \{\Theta \lceil \Sigma \rceil \text{ struct } ins_1 : loc_1 = \{\lceil \sigma_1 \rceil\}. \lceil \Sigma' \rceil\}.$$

Here  $\Theta$  contains type declarations  $\alpha : tp$  for the free type variables of the locale. Those are the free type variables that occur in the declarations of  $\Sigma$  and  $\Sigma'$ . These correspond to the single declaration  $this : tp$  in type classes. This encoding of type variables may be surprising because free type variables correspond to universal types whereas the declarations in  $\Theta$  correspond to existential types. We hold that our LF-encoding precisely captures the intended meaning of locales, whereas the definition of locales within Isabelle prefers universal type variables in order to be compatible with the underlying type theory.

If a locale inherits from more than one locale, the encoding is defined correspondingly using one structure  $\text{struct } ins_i : loc_i = \{\vartheta_i \lceil \sigma_i \rceil\}$  for each locale instance  $ins_i : loc_i$  where  $\sigma_i$ . Here  $\vartheta_i$  contains the

instantiations for free type variables of  $loc_i$  that are induced by  $\sigma_i$  and inferred by Isabelle. Furthermore, some additional sharing declarations become necessary due to a subtlety in the semantics of Isabelle locales: If a locale inherits two equal instances (same locale, same instantiations), they are implicitly identified. But in LF different structures are always distinguished unless shared explicitly. Therefore, we have to add to  $\lceil \sigma_i \rceil$  one sharing declaration  $\text{struct } ins := ins'$  for each instance  $ins$  present in  $loc_i$  that is equal to one already imported by one of  $ins_1, \dots, ins_{i-1}$ .

*Example 5* (Continued). The locale from Ex. 1 is represented in LF as follows:

```
sig lat = {
    struct inf : semlat.
    struct sup : semlat = {this := inf.this.  $\leq := \lambda[x]\lambda[y]$  inf.  $\leq @y@x$ }.
}
```

Note how the instantiation for  $\leq$  induces an instantiation for the type  $this$ . In other words, the  $\vartheta$  mentioned above is  $this := inf.this$ .

*Sublocale* declarations are encoded as views from the super- to the sublocale. Thus, the declaration

$$\text{sublocale } loc' < loc \text{ where } \sigma \pi$$

is encoded as (for some fresh name  $v$ ):

$$\text{view } v : loc \rightarrow loc' = \{\vartheta \lceil \sigma \rceil \lceil \pi \rceil\}.$$

Here  $\vartheta$  contains the instantiations of the free type variables (see  $\Theta$  above), which are inferred by Isabelle based on the instantiations in  $\sigma$ .

Locale *interpretations* are interpreted in the same way except that the codomain is the current LF signature (which encodes the Isabelle theory containing the locale interpretation) instead of the sublocale.

As for type classes, we have the general result that  $loc$  is a sublocale of  $loc'$  iff there is an LF signature morphism from  $loc$  to  $loc'$ . Accordingly,  $loc$  can be interpreted in the theory  $T$  iff there is a morphism from  $loc$  to  $T$ . For example,  $loc$  is a sublocale of  $loc_1$  from above via the composed morphism  $v loc.ins_1$ . Contrary to type classes, there may be several different sublocale relationships between two locales. In LF these are distinguished elegantly as different morphisms between the locales.

*Example 6* (Continued).  $lat$  is a sublocale of  $semlat$  in two different ways represented by the LF morphisms  $lat.inf$  and  $lat.sup$ . These are trivial sublocale relations induced by inheritance.

**Constant Declarations** Finally we have to represent those aspects of the non-modular declarations that are affected by type classes. We will only consider the case of constants. Definitions, axioms, and theorems are represented accordingly. The central idea is that free type variables constrained by type classes are represented using  $\lambda$  abstraction for morphism variables.

An Isabelle constant  $c :: \tau$  with free type variables  $\alpha_i :: C_i$  is represented as the LF-constant taking morphism arguments:

$$c : \{\alpha_1 : C_1\} \dots \{\alpha_n : C_n\} tm \lceil \tau \rceil.$$

Here in  $\lceil \tau \rceil$  every occurrence of the morphism variable  $\alpha_i$  is represented as  $\alpha_i.this$ .

Whenever  $c$  is used with inferred type arguments  $\tau_i :: C_i$  in a composed expression, it is represented by application of  $c$  to morphisms:

$$\lceil c \rceil = c \lceil \tau_1 :: C_1 \rceil \dots \lceil \tau_n :: C_n \rceil.$$

Actually, we cannot use the same identifier  $c$  in LF as in Isabelle: Instead, we must keep track how  $c$  came into scope. For example, if  $c$  was imported from some theory  $S$ , we must use  $S.c$  in LF; if the current scope is a locale and  $c$  was imported from some other locale via an instance  $ins$ , we must use  $ins.c$  in LF; if  $c$  was moved into the current theory from a locale  $loc$  via an interpretation declaration which was encoded using the fresh name  $v$ , we must use  $loc.c^v$  in LF, and so on.

**Types** The representation of types was already indicated above, but we summarize it here for clarity. Type operator declarations  $(\alpha_1, \dots, \alpha_n)t$  are encoded as constants  $t : tp \rightarrow \dots \rightarrow tp \rightarrow tp$ . And types occurring in expressions are encoded as

$$\begin{array}{lll} \ulcorner \alpha :: C \urcorner & = & \alpha.this \\ \ulcorner t \urcorner & = & t \\ \ulcorner (\tau_1, \dots, \tau_n)t \urcorner & = & t \ulcorner \tau_1 \urcorner \dots \ulcorner \tau_n \urcorner \\ \ulcorner \tau_1 \Rightarrow \tau_n \urcorner & = & \ulcorner \tau_1 \urcorner \Rightarrow \ulcorner \tau_2 \urcorner \\ \ulcorner prop \urcorner & = & prop. \end{array}$$

**Adequacy** Before we state the adequacy, we need to clarify in what sense our representation is adequate. In Isabelle, locales and type classes are not primitive notions. Instead, they are internally elaborated into the underlying type theory. For example, all declarations in a locale or a type class are relativized and lifted to the top level. Thus, they are available elsewhere and not only within the locale.

While there are certainly situations when this is useful, here we care about the modular structure and the underlying type theory, but not about the elaboration of the former into the latter. Therefore, we do not want a representation in LF that adequately preserves the elaboration. In fact, if we wanted to preserve the elaboration, we could simply use Isabelle to eliminate all modular structure and represent the non-modular result using well-known representations of higher-order logic in LF.

Therefore, we have to forbid all Isabelle theories where names are used outside their scope. Let us call an Isabelle theory *simple* if all declared names are only used in their respective declaration scope – theory, locale, or type class – unless they were explicitly moved into a new scope using imports, sublocale, interpretation, or instantiation declarations, or using inheritance between type classes and locales.

Then we can summarize our representation with the following theorem:

**Theorem 7.** *If a simple sequence of Isabelle theories  $T_1 \dots T_n$  is well-formed (in the sense of Isabelle), then the LF signature graph  $\text{Pure } \ulcorner T_1 \urcorner \dots \ulcorner T_n \urcorner$  is well-formed (in the sense of LF extended with morphism variables).*

*Proof.* To show the adequacy for the encoding of the inner syntax is straightforward. A similar proof was given in [6].

The major lemmas for the outer syntax were already indicated in the text:

- For an Isabelle type class instance  $\tau :: C$  used in theory or locale  $S$  and context  $\Gamma$ , we have  $G; \Gamma \triangleright \ulcorner \tau :: C \urcorner : C \rightarrow S$  and  $\ulcorner \tau :: C \urcorner(C.this) = \ulcorner \tau \urcorner$ .
- There is an Isabelle sublocale relation  $loc' < loc$  via instantiations  $\sigma$  whenever the incomplete LF morphism  $\{\ulcorner \sigma \urcorner \dots : loc \rightarrow loc'\}$  can be completed (by instantiating the axioms of  $loc$  with proof terms over  $loc'$ ).

The main difficulty in the proofs is to show that at any point in the translated LF signatures exactly the right atomic expressions are in scope. This has to be verified by a difficult and tedious comparison of the Isabelle documentation with the semantics of the LF module system. In particular, in our simplified grammar for Isabelle, we have omitted the features that would break this result. These include in particular the features whose translation requires inventing and keeping track of fresh names, such as overloading and unqualified locale instantiation.  $\square$

## 5 Conclusion

We have presented a representation of Isabelle’s module system in the LF module system. Previous logic encodings in LF have only covered non-modular languages (e.g., [6] [8] [15]), and ours is the first encoding of a modular logic. We also believe ours to be the first encoding of type classes or locale-like features in any logical framework.

The details of the translation are quite difficult, and a full formalization requires intricate knowledge of both systems. However, guided by the use of signatures and signature morphisms as the main primitives in the LF module system, we could give a relatively intuitive account of Isabelle’s structuring mechanisms.

Our translation preserves modular structure; in particular the translation is compositional and the size of the output is linear in the size of the input. We are confident that our approach scales to other systems such as the type classes of Haskell or the functors of SML, and thus lets us study the modular properties of programming languages in logical frameworks. Moreover, we hold that the trade-off made in the LF module system between expressivity and simplicity makes it a promising starting point to investigate the movement of modular developments between systems.

In order to formulate the representation, we had to add abstraction over morphisms to the LF module system. This effectively gives LF a restricted version of dependent record types. This is similar to the use of contexts as dependent records as, e.g., in [17]. Contrary to, e.g., [3] and [12], the LF records may only occur in contravariant positions, which makes them a relatively simple conservative addition.

An integration of this feature into the Twelf implementation of LF remains future work. Similarly, the use of anonymous morphisms has not been implemented in Twelf yet. In both cases, the implementation is conceptually straightforward. However, since it would permit the use of morphisms in terms, types, and kinds, it would require a closer integration of modular and core syntax in Twelf, which has so far been avoided deliberately. We will undertake the Twelf side of the implementation soon.

In any case, Twelf will hardly be a bottleneck. Any implementation of a translation from Isabelle to LF would have to be implemented from within Isabelle as it requires Isabelle’s reconstruction of types and instantiations (let alone proof terms). However, Isabelle currently eliminates most aspects of modularity when checking a theory. For example, it is already difficult to export the local constants of a theory because the methods provided by Isabelle can only return all local, imported, or internally generated constants at once. The most promising albeit still very difficult approach seems to be to use a standalone parser for the Isabelle outer syntax and then fill in the gaps by calling the methods provided by Isabelle. Thus, even though this paper solves the logical questions how to translate from Isabelle to LF, the corresponding software engineering questions are non-trivial and remain open.

## References

- [1] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.
- [2] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.
- [3] R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.
- [4] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.
- [5] F. Haftmann and M. Wenzel. Constructive Type Classes in Isabelle. In T. Altenkirch and C. McBride, editors, *TYPES conference*, pages 160–174. Springer, 2006.
- [6] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
- [7] R. Harper and B. Pierce. Design Issues in Advanced Module Systems. In B. Pierce, editor, *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.
- [8] R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.
- [9] F. Kammüller, M. Wenzel, and L. Paulson. Locales – a Sectioning Concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics*, pages 149–166. Springer, 1999.
- [10] P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the '73 Logic Colloquium*, pages 73–118. North-Holland, 1974.
- [11] T. Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *TYPES conference*, pages 259–278. Springer, 2002.
- [12] U. Norell. The Agda WiKi, 2005. <http://wiki.portal.chalmers.se/agda>
- [13] L. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.
- [14] L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
- [15] F. Pfenning. Structural cut elimination: I. intuitionistic and classical logic. *Information and Computation*, 157(1-2):84–141, 2000.
- [16] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.
- [17] B. Pientka and J. Dunfield. A Framework for Programming and Reasoning with Deductive Systems (System description). In *International Joint Conference on Automated Reasoning*, 2010. To appear.
- [18] F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, pages 40–48. ACM Press, 2009.
- [19] F. Rabe and C. Schürmann. A Practical Module System for LF. In *Proceedings of the Workshop on Logical Frameworks Meta-Theory and Practice (LFMTP)*, 2009.
- [20] D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. In M. Karpinski, editor, *Fundamentals of Computation Theory*, pages 413–427. Springer, 1983.

# A Proof Theoretic Interpretation of Model Theoretic Hiding

Mihai Codescu, Fulya Horozal, Michael Kohlhase, Till Mossakowski, Florian Rabe

Jacobs University Bremen, DFKI Bremen

**Abstract.** Logical frameworks like LF are used for formal representations of logics in order to make them amenable to formal machine-assisted meta-reasoning. While the focus has originally been on logics with a proof theoretic semantics, we have recently shown how to define model theoretic logics in LF as well. We have used this to define new institutions in the Heterogeneous Tool Set in a purely declarative way.

It is desirable to extend this model theoretic representation of logics to the level of structured specifications. Here a particular challenge among structured specification building operations is hiding, which restricts a specification to some export interface. Specification languages like ASL and CASL support hiding, using an institution-independent model theoretic semantics abstracting from the details of the underlying logical system.

Logical frameworks like LF have also been equipped with structuring languages. However, their proof theoretic nature leads them to a theory-level semantics without support for hiding. In the present work, we show how to resolve this difficulty.

## 1 Introduction

This work is about reconciling the model theoretic approach of algebraic specifications and institutions [AKKB99,ST10,GB92] with the proof theoretic approach of logical frameworks [HHP93,Pau94].

In [Rab10,CHK<sup>+</sup>10], we show how to represent institutions in logical frameworks, notably LF [HHP93], and extend the Heterogeneous Tool Set [MML07] with a mechanism to add new logics that are specified declaratively in a logical framework.

In the present work, we extend this to the level of structured specifications, including hiding. In particular, we will translate the ASL-style structured specifications with institutional semantics [SW83,Wir86,ST88] (also used in CASL [Mos04]) into the module system MMT [Rab08] that has been developed in the logical frameworks community.

Like ASL, MMT is a generic structuring language that is parametric in the underlying language. But where ASL assumes a model theoretic base language – given as an institution – MMT assumes a proof theoretic base language given in terms of typing judgments. If we instantiate MMT with LF (as done in [RS09]), we can represent both logics and theories as MMT-structured LF signatures. This is used in the LATIN project [KMR09] to obtain a large body of structured representations of logics and logic translations. An important practical benefit of MMT is that it is integrated with a scalable knowledge management infrastructure based on OMDoc [Koh06].

However, contrary to model theoretic structuring languages like ASL, structuring languages like MMT for logical frameworks have a proof theoretic semantics and do not support hiding, which makes them less expressive than ASL. Therefore, we proceed in two steps. Firstly, we

extend LF+MMT with primitives that support hiding while preserving its proof theoretic flavor. Here we follow and extend the theory-level semantics for hiding given in [GR04]. Secondly, we assume an institution that has been represented in LF, and give a translation of ASL-structured specifications over it into the extended LF+MMT language.

The paper is organised as follows. In Sect. 2, we recall ASL- or CASL-style structured specifications with their institution-independent semantics; and in Sect. 3 we recall LF and MMT with its proof theoretic semantics. In Sect. 4, we extend MMT with hiding, and in Sect. 5, we define a translation of ASL style specifications into MMT and prove its correctness. Sect. 6 concludes the paper.

## 2 Structured specifications

The notion of institution [GB92] has been introduced as a formalisation of the notion of logical system. It abstracts away from the details of signatures, sentences and models. Moreover, it assumes that signatures can be related via signature morphisms (and this carries over to sentences and models). This will be of importance for structuring languages.

**Definition 1.** An institution is a quadruple  $I = (\text{Sig}, \text{Sen}, \text{Mod}, \models)$  where:

- $\text{Sig}$  is a category of signatures;
- $\text{Sen} : \text{Sig} \rightarrow \mathcal{SET}$  is a functor to the category  $\mathcal{SET}$  of small sets and functions, giving for each signature  $\Sigma$  its set of sentences  $\text{Sen}(\Sigma)$  and for any signature morphism  $\varphi : \Sigma \rightarrow \Sigma'$  the sentence translation function  $\text{Sen}(\varphi) : \text{Sen}(\Sigma) \rightarrow \text{Sen}(\Sigma')$  (denoted also  $\varphi$ );
- $\text{Mod} : \text{Sig}^{\text{op}} \rightarrow \text{Cat}$  is a functor to the category of categories and functors  $\text{Cat}$ <sup>1</sup> giving for any signature  $\Sigma$  its category of models  $\text{Mod}(\Sigma)$  and for any signature morphism  $\varphi : \Sigma \rightarrow \Sigma'$  the model reduct functor  $\text{Mod}(\varphi) : \text{Mod}(\Sigma') \rightarrow \text{Mod}(\Sigma)$  (denoted  $\_|_{\varphi}$ );
- a satisfaction relation  $\models_{\Sigma} \subseteq |\text{Mod}(\Sigma)| \times \text{Sen}(\Sigma)$  for each signature  $\Sigma$

such that the following satisfaction condition holds:

$$M'|_{\varphi} \models_{\Sigma'} e \Leftrightarrow M' \models_{\Sigma} \varphi(e)$$

for each  $M' \in |\text{Mod}(\Sigma')|$  and  $e \in \text{Sen}(\Sigma)$ , expressing that truth is invariant under change of notation and context.

For an institution  $I$ , a *theory* is a pair  $(\Sigma, E)$  where  $\Sigma$  is a signature and  $E$  is a set of sentences. For a class  $E$  of  $\Sigma$ -sentences, let us denote  $\text{Mod}_{\Sigma}(E)$  the class of all  $\Sigma$ -models satisfying  $E$  and  $\text{Cl}_{\Sigma}(E)$  the logical consequences of  $E$ .

Working with monolithic specifications is only suitable for specifications of fairly small size. For practical situations, in the case of large systems, a flat specification would become impossible to understand and use efficiently. Moreover, a modular design allows for reuse of specifications. Therefore, algebraic specification languages provide support for structuring specifications.

---

<sup>1</sup> We disregard here the foundational issues, but notice however that  $\text{Cat}$  is actually a so-called quasi-category.

The semantics of (structured) specifications can be given as a signature and either (i) a class of models of that signature (*model-level semantics*) or (ii) a set of sentences over that signature (*theory-level semantics*). In the presence of structuring, the two semantics may be different in a sense that will be made precise below. The first algebraic specification language, Clear [BG80], used a theory-level semantics; the first algebraic specification language using model-level semantics for structured specifications was ASL [SW83,Wir86], whose structuring mechanisms were extended to an institution-independent level in [ST88].

In Fig. 1, we present a kernel of specification-building operations and their semantics over an arbitrary institution, similar to the one introduced in [ST88]. The third and fourth columns of the table contain the model-level and the theory-level semantics for the corresponding **structured specification**  $SP$ , denoted  $Mod[SP]$  and  $Thm[SP]$  respectively, while the signature of  $SP$ , denoted  $Sig[SP]$  is in the second column of the table. Note that we restrict attention to hiding against inclusion morphisms. Moreover, we will only consider basic specifications that are finite.

| $SP$   | $Sig[SP]$   | $Mod[SP]$  | $Thm[SP]$                                       |
|--|-------------|--|---|
| $(\Sigma, E)$  | $\Sigma$    | $Mod_{\Sigma}(E)$                                  | $Cl_{\Sigma}(E)$                                |
| $SP_1 \cup SP_2$<br>$Sig[SP_1] = Sig[SP_2]$                    | $Sig[SP_1]$ | $Mod(SP_1) \cap Mod(SP_2)$                         | $Cl_{\Sigma}(Thm[SP_1] \cup Thm[SP_2])$         |
| $\sigma(SP)$<br>$\sigma : Sig[SP] \rightarrow \Sigma'$         | $\Sigma'$   | $\{M \in Mod(\Sigma')   M _{\sigma} \in Mod[SP]\}$ | $Cl_{\Sigma}(\{\sigma(e)   e \in Thm[SP]\})$    |
| $\sigma^{-1}(SP)$<br>$\sigma : \Sigma \hookrightarrow Sig[SP]$ | $\Sigma$    | $\{M _{\sigma}   M \in Mod[SP]\}$                  | $\{e \in Sen(\Sigma)   \sigma(e) \in Thm[SP]\}$ |

**Fig. 1.** Semantics of Structured Specifications

Without hiding, the two semantics can be regarded as dual because we have  $Thm[SP] = Thm(Mod[SP])$ , which is called *soundness* and *completeness* in [ST10]. But completeness does not hold in general in the presence of hiding [Bor02]. Moreover, in [ST10] it is proved that this choice for defining the theory level semantics is the strongest possible choice with good structural properties (e.g. compositionality). This shows that the mismatch between theory-level semantics and model-level semantics cannot be bridged in this way. We will argue below that this is not a failure of formalist methods in general; instead, we will pursue a different approach that takes model-level aspects into account while staying mechanizable.

The mismatch between model and theory-level semantics is particularly apparent when looking at **refinements**. For two  $\Sigma$ -specifications  $SP$  and  $SP'$ , we write  $SP \rightsquigarrow_{\Sigma} SP'$  if  $Mod[SP'] \subseteq Mod[SP]$ . Without hiding, this is equivalent to  $Thm[SP] \subseteq Thm[SP']$ , which can be seen as soundness and completeness properties for refinements. But in the presence of hiding, both soundness (if  $SP$  has hiding) and completeness (if  $SP'$  has hiding) for refinements may fail.

### 3 LF and MMT

The Edinburgh Logical Framework LF [HHP93] is a proof theoretic logical framework based on a dependent type theory related to Martin-Löf type theory [ML74]. Precisely, it is the corner of the  $\lambda$ -cube [Bar92] that extends simple type theory with dependent function types. In [RS09], LF was extended with a module system called MMT. MMT [Rab08] is a generic module system which structures signatures using named imports and signature morphisms. The expressivity of MMT is similar to that of ASL or development graphs [AHMS99] except for hiding. In [Rab10], LF is used as a logical framework to represent both proof and model theory of object logics.

We give a brief summary of basic LF signatures, MMT-structured LF signatures, and the representation of model theory in LF in Sect. 3.1, 3.2, and 3.3, respectively. Our approach is not restricted to LF and can be easily generalized to other frameworks such as Isabelle or Maude along the lines of [CHK<sup>+</sup>10].

#### 3.1 LF

LF expressions  $E$  are grouped into kinds  $K$ , kinded type-families  $A : K$ , and typed terms  $t : A$ . The kinds are the base kind **type** and the dependent function kinds  $\Pi x : A. K$ . The type families are the constants  $a$ , applications  $a t$ , and the dependent function type  $\Pi x : A. B$ ; type families of kind **type** are called types. The terms are constants  $c$ , applications  $t t'$ , and abstractions  $\lambda x : A. t$ . We write  $A \rightarrow B$  instead of  $\Pi x : A. B$  if  $x$  does not occur in  $B$ . An LF **signature**  $\Sigma$  is a list of kinded type family declarations  $a : K$  and typed constant declarations  $c : A$ . Optionally, declarations may carry definitions. A grammar is given in Sect. 3 below.

Given two signatures  $\Sigma$  and  $\Sigma'$ , an LF **signature morphism**  $\sigma : \Sigma \rightarrow \Sigma'$  is a typing- and kinding-preserving map of  $\Sigma$ -symbols to  $\Sigma'$ -expressions. Thus,  $\sigma$  maps every constant  $c : A$  of  $\Sigma$  to a term  $\sigma(c) : \bar{\sigma}(A)$  and every type family symbol  $a : K$  to a type family  $\sigma(a) : \bar{\sigma}(K)$ . Here,  $\bar{\sigma}$  is the homomorphic extension of  $\sigma$  to  $\Sigma$ -expressions, and we will write  $\sigma$  instead of  $\bar{\sigma}$  from now on. Signature morphisms preserve typing and kinding: if  $\vdash_{\Sigma} E : E'$ , then  $\vdash_{\Sigma'} \sigma(E) : \sigma(E')$ .

Composition and identity of signature morphisms are straightforward, and we obtain a category  $\mathbb{LF}$  of LF signatures and morphisms. This category has **inclusion** morphisms by taking inclusions between sets of declarations. Moreover, it has **pushouts** along inclusions [HST94]. Finally, a **partial morphism** from  $\Sigma$  to  $\Sigma'$  is a signature morphism from a subsignature of  $\Sigma$  to  $\Sigma'$ .

LF uses the Curry-Howard correspondence to represent axioms as constants and theorem as defined constants (whose definiens is the proof). Then the typing-preservation of signature morphisms corresponds to the theorem preservation of theory morphisms.

#### 3.2 LF+MMT

The motivation behind the MMT structuring operations is to give a flattenable, concrete syntax for a module system on top of a declarative language. Signature morphisms are used as the main concept to relate and form modular signatures, and signature morphisms can themselves be given in a structured way. Moreover, signature morphisms are always named and can be composed into morphism expressions.

The grammar for the LF+MMT language is given below where  $[-]$  denotes optional parts. Object level expressions  $E$  unify LF terms, type families, and kinds, and morphism level expressions are composed morphisms:

|                            |  |
|----------------------------|--|
| Signature graph            | $G ::= \cdot \mid G, \%sig T = \{\Sigma\} \mid \%view v : S \rightarrow T = \{\sigma\}$          |
| Signatures                 | $\Sigma ::= \cdot \mid \Sigma, \%struct s : S = \{\sigma\} \mid \Sigma, c : E [= E']$            |
| Morphisms                  | $\sigma ::= \cdot \mid \sigma, \%struct s := \mu \mid \sigma, c := E$                            |
| Object level expressions   | $E ::= type \mid c \mid x \mid E E \mid \lambda x : E. E \mid \Pi x : E. E \mid E \rightarrow E$ |
| Morphism level expressions | $\mu ::= \cdot \mid T.s \mid v \mid \mu \mu'$  |

The LF signatures and signature morphisms are those without the keyword `%struct`. Those are called **flat**.

*Syntax* The module level declarations consist of named signatures  $R, S, T$  and two kinds of signature morphism declarations. Firstly, `views %view v : S → T = {σ}` occur on toplevel and declare an explicit morphism from  $S$  to  $T$  given by  $σ$ . Secondly, `structures %struct s : S = {σ}` occur in the body of a signature  $T$  and declare an import from  $S$  into  $T$ . Structures carry a partial morphism  $σ$  from  $S$  to  $T$ , i.e.,  $σ$  maps some symbols of  $S$  to expressions over  $T$ . Views and structures correspond to refinements and inclusion of subspecifications in unions in ASL and CASL.

MMT differs from ASL-like structuring languages in that it uses named imports. Consequently, the syntax of MMT can refer to all paths in the signature graph using composed morphisms; these morphism level expressions  $μ$  are formed from structure names  $T.s$ , view names  $v$ , and diagram-order composition  $μ μ'$ .

MMT considers morphisms  $μ$  from  $S$  to  $T$  as expressions on the module level. Such a morphism  $μ$  has type  $S$  and is valid over  $T$ . Most importantly, MMT permits structured morphisms: The morphisms  $σ$  occurring in views and structures from  $S$  to  $T$  may map a structure `%struct r : R = {σ}` declared in  $S$  (i.e., a morphism level constant of type  $R$  over  $S$ ) to a morphism  $μ : R \rightarrow T$  (i.e., a morphism level expression of type  $R$  over  $T$ ). These are called structure maps `%struct r := μ`.

*Semantics* The semantics of LF+MMT is given by **flattening**. Every well-formed LF+MMT signature graph  $G$  is flattened into a diagram  $\overline{G}$  over LF. Every signature  $S$  in  $G$  produces a node  $\overline{S}$  in  $\overline{G}$ ; every structure `%struct s : S = {σ}` occurring in  $T$  produces an edge  $\overline{T.s}$  from  $\overline{S}$  to  $\overline{T}$ ; and every view `%view v : S → T = {σ}` produces an edge  $\overline{v}$  from  $\overline{S}$  to  $\overline{T}$ . Accordingly, every morphism expression  $μ$  yields a morphism  $\overline{μ}$ . These results can be found in [RS09], and we will only sketch the central aspects here.

The flattening is defined by recursively replacing all structure declarations and structure maps with lists of flat declarations. To flatten a structure declaration `%struct s : S = {σ}` in a signature  $T$ , assume that  $S$  and  $σ$  have been flattened already. For every declaration  $c : E [= E']$  in  $\overline{S}$ , we have in  $\overline{T}$

- a declaration  $s.c : \overline{T.s}(E) = E''$  in  $\overline{S}$  if  $σ$  contains  $c := E''$ ,
- a declaration  $s.c : \overline{T.s}(E) [= \overline{T.s}(E')]$  in  $\overline{S}$  otherwise.

The morphism  $\overline{T.s}$  from  $\overline{S}$  to  $\overline{T}$  maps every  $\overline{S}$ -symbol  $c$  to the  $\overline{T}$  symbol  $s.c$ .

For a view  $\%view v : S \rightarrow T = \{\sigma\}$ , the morphism  $\overline{v}$  from  $\overline{S}$  to  $\overline{T}$  is given by the flattening of  $\sigma$ .  $\overline{\cdot}$  is the identity morphism in  $\text{LF}$ , and  $\overline{\mu} \mu'$  is the composition  $\mu' \circ \mu$ .

Finally, morphisms  $\sigma$  from  $S$  to  $T$  are flattened as follows. To flatten a structure map  $\%struct r := \mu$  where  $r$  is a structure from  $R$  to  $S$ , assume that  $R$  has been flattened already. Then the flattening of  $\sigma$  contains  $s.c := \overline{\mu}(c)$  for every constant  $c$  in  $\overline{R}$ .

In particular, if  $\%sig T = \{\Sigma, \%struct s : S = \{\sigma\}\}$  the semantics of signature graphs is such that the left diagram below is a pushout. Here  $S_0$  is a subsignature of  $\overline{S}$  such that  $\overline{\sigma} : S_0 \rightarrow \overline{\Sigma}$ . Moreover, if  $S$  declares a structure  $r$  of type  $R$ , then the semantics of a structure map  $\%struct r := \mu$  occurring in  $\sigma$  is that the diagram on the right commutes.

$$\begin{array}{ccc} S_0 & \longrightarrow & \overline{S} \\ \downarrow \overline{\sigma} & & \downarrow \overline{T.s} \\ \overline{\Sigma} & \longrightarrow & \overline{T} \end{array} \quad \begin{array}{ccc} \overline{R} & \xrightarrow{\overline{S.r}} & \overline{S} \\ & \searrow \overline{\mu} & \downarrow \overline{T.s} \\ & & \overline{T} \end{array}$$

### 3.3 Representing Logics in LF

The main idea behind the representation of models in LF is to represent models of  $\Sigma$  as LF morphisms  $\lceil \Sigma \rceil \rightarrow \mathcal{F}$  where  $\lceil \Sigma \rceil$  encodes  $\Sigma$  and  $\mathcal{F}$  is some LF signature. Usually,  $\mathcal{F}$  is a fixed signature representing the foundation of mathematics such as an encoding of set theory. The feasibility of this approach has been demonstrated in [HR10,IR10], which in particular give encodings of ZFC set theory, Mizar's set theory, and Isabelle's higher-order logic.

We use a simplified variant and represent the underlying logic  $L$  as tuples  $(L^{Syn}, L^{Mod}, L^{mod}, \mathcal{F})$  as in the commuting LF diagram on the right.  $L^{Syn}$  represents the syntax of the logic,  $\mathcal{F}$  represents the foundation of mathematics, and  $L^{Mod}$  represents individual models as an extension of  $\mathcal{F}$ . Finally,  $L^{mod}$  interprets the syntax in the model.

Individual **signatures** are represented as inclusion morphisms out of  $L$ , from which we obtain  $\Sigma^{Mod}$  and  $\Sigma^{mod}$  as a pushout. Now we can see  $\Sigma^{Mod}$  as a theory in the meta-language  $\mathcal{F}$  axiomatizing  $\Sigma$  models. Thus **models**  $M$  can finally be represented as morphisms from  $\Sigma^{Mod}$  into the foundation that are the identity on  $\mathcal{F}$ .

Due to the Curry-Howard representation of proofs as terms, there is no conceptual difference between representing signatures and theories of the underlying logic. If  $\Sigma^{Syn}$  contains axioms, then so does  $\Sigma^{Mod}$ , and  $M$  must map these axioms to proofs in  $\mathcal{F}$ .

We assume that  $L^{Syn}$  contains two distinguished declarations  $o : \text{type}$  and  $\text{ded} : o \rightarrow \text{type}$ . Then  $\Sigma$ -sentences are represented as closed  $\beta\eta$ -normal terms of type  $o$  over  $\Sigma^{Syn}$ . The satisfaction of a sentence  $F$  by a model  $M$  is represented as the inhabitation of the type  $M(\Sigma^{mod}(\text{ded } F))$  over  $\mathcal{F}$ . Theorems are represented as sentences  $F$  for which the type  $\text{ded } F$  is

$$\begin{array}{ccccc} \mathcal{F} & \xrightarrow{id_{\mathcal{F}}} & \mathcal{F} & & \\ \downarrow & & \uparrow M & & \\ L^{Mod} & \longrightarrow & \Sigma^{Mod} & & \\ \uparrow L^{mod} & & \uparrow \Sigma^{mod} & & \\ L^{Syn} & \longrightarrow & \Sigma^{Syn} & & \end{array}$$

inhabited over  $\Sigma^{Syn}$ . In [Rab10,CHK<sup>+</sup>10], the proof theory of the logic is represented parallel to the model theory as a morphism  $L^{pf} : L^{Syn} \rightarrow L^{Pf}$  where  $L^{Pf}$  adds the proof rules that populate the types  $\text{ded } F$ . Here, we will assume for simplicity that  $L^{Syn} = L^{Pf}$ , and our results easily extend to the general case.

## 4 Hiding in LF and MMT

In a proof theoretic setting, flattening is not a theorem but rather the way to assign meaning to a modular signature. Therefore, hiding is a particularly difficult operation to add to systems like LF+MMT because hiding precludes flattening. We follow the approach taken in [GR04] and represent signatures with hidden information as inclusions  $\Sigma_v \hookrightarrow \Sigma_h$  where  $\Sigma_v$  represents the visible interface  $\Sigma_h \setminus \Sigma_v$  the hidden information. We will abstractly introduce LF signatures with hidden declarations in and morphisms between such signatures in Sect. 4.1. Then we will give concrete syntax for them in and generalize the MMT structuring operations in Sect. 4.2.

### 4.1 LF with Hiding

We will not only introduce LF signatures with hidden declarations but also LF morphisms that hide constants. The latter is similar to partial morphisms but has to be distinguished from the partial morphisms that already occur in MMT-structures. Therefore, we need two kinds of partiality and use the following definition.

Given two LF-signatures  $\Sigma$  and  $\Sigma'$ , an **H-morphism** from  $\Sigma$  to  $\Sigma'$  consists of two subsignatures  $\Sigma_0 \hookrightarrow \Sigma_1 \hookrightarrow \Sigma$  and an LF signature morphism  $\sigma : \Sigma_0 \rightarrow \Sigma'$ . The intuition is that  $\sigma$  maps all constants in  $\Sigma_0$  to  $\Sigma'$ -expressions and hides all constants in  $\Sigma \setminus \Sigma_1$ ; for the intermediate declarations in  $\Sigma_1 \setminus \Sigma_0$ ,  $\sigma$  is left undefined, i.e., partial. We call  $\Sigma_0$  the **revealed domain** and  $\Sigma_1$  the **non-hidden domain** of  $\sigma$ . We call  $\sigma$  **total** if  $\Sigma_1 = \Sigma_0$  and otherwise **partial**; and we call  $\sigma$  **revealing** if  $\Sigma = \Sigma_1$  and otherwise **hiding**. Then the (partial) revealing morphisms are exactly the (partial) LF-morphisms.

For a  $\Sigma$ -expression  $E$ , we say that  $\sigma$  **maps**  $E$  if  $E$  is a  $\Sigma_0$ -expression and that  $\sigma$  **hides**  $E$  if  $E$  is not a  $\Sigma_1$ -expression. Then we can define a **composition** of total H-morphisms as follows: The revealed domain of  $\sigma' \circ \sigma$  is the largest subsignature of the revealed domain of  $\sigma$  comprising only constants  $c$  such that  $\sigma'$  maps  $\sigma(c)$ ; then we can put  $(\sigma' \circ \sigma)(c) = \sigma'(\sigma(c))$ .

An **H-signature** is a pair  $\Sigma = (\Sigma_v, \Sigma_h)$  such that  $\Sigma_v$  is a subsignature of  $\Sigma_h$ . We call  $\Sigma_h$  the **domain** and  $\Sigma_v$  the **visible domain** of  $\Sigma$ .

Finally, we define the category  $\text{LFH}$  whose objects are H-signatures and whose morphisms  $(\Sigma_v, \Sigma_h) \rightarrow (\Sigma'_v, \Sigma'_h)$  are total H-morphisms from  $\Sigma_v$  to  $\Sigma'_v$ . Note that these morphisms are exactly the total morphisms from  $\Sigma_h$  to  $\Sigma'_v$  whose revealed domain is at most  $\Sigma_v$ .

**Lemma 2.**  *$\text{LFH}$  is indeed a category.*

*Proof.* The  $\text{LFH}$  identity of  $(\Sigma_v, \Sigma_h)$  is the LF identity of  $\Sigma_v$ . Neutrality is clear. Associativity follows after observing that  $\sigma'' \circ (\sigma' \circ \sigma)$  hides  $c$  iff  $(\sigma'' \circ \sigma') \circ \sigma$  hides  $c$ .

$\text{LFH}$ -morphisms only translate between the visible domains and may even use hiding in doing so. We are often interested in whether the hidden information could also be translated. Therefore, we define:

**Definition 3.** For an  $\text{LFH}$ -morphism  $\sigma_0 : \Sigma \rightarrow \Sigma'$  with revealed domain  $\Sigma_0$ , we write  $\sigma_0 : \Sigma \xrightarrow{\downarrow} \Sigma'$  if  $\sigma_0$  can be extended to a total revealing morphism  $\sigma : \Sigma_h \rightarrow \Sigma'_h$ , i.e., if there is an  $\text{LF}$  morphism  $\sigma : \Sigma_h \rightarrow \Sigma'_h$  that agrees with  $\sigma_0$  on  $\Sigma_0$ .

## 4.2 LF+MMT with Hiding

We can now extend the MMT structuring to  $\text{LFH}$ , i.e., to a base language with hiding. The flattening of signature graphs with hiding will produce  $\text{LFH}$ -diagrams.

We avoid using pairs  $(\Sigma_v, \Sigma_h)$  in the concrete syntax for H-signatures and instead extend the grammar of LF+MMT as follows:

```
Signatures  $\Sigma ::= \cdot \mid \Sigma, [\%hide] \%struct s : S = \{\sigma\} \mid \Sigma, [\%hide] c : E [= E]$ 
Morphisms  $\sigma ::= \cdot \mid \sigma, \%struct s := \mu \mid \sigma, c := E \mid \%hide c \mid \%hide \%struct s$ 
```

If a declaration in  $\Sigma$  has the `%hide` modifier, we call it **hidden**, otherwise **visible**. Hidden declarations are necessary to keep track of the hidden information. From a proof theoretical perspective, it may appear more natural to delete them, but this would not be adequate to represent ASL specifications with hiding.

If  $\sigma$  contains `%struct c := E` (or `%hide c`), we say that  $\sigma$  **maps** (or **hides**)  $c$ , and accordingly for structures. As before, we call signatures or morphisms **flat** if they do not contain the `%struct` keyword.

The semantics of a well-formed signature graph  $G$  is given in two steps: first  $G$  is flattened into a flat signature graph  $\tilde{G}$ , second the semantics of a flat signature graph  $G$  is given by an  $\text{LFH}$ -diagram  $\bar{G}$ . In particular, every composite  $\mu$  from  $S$  to  $T$  occurring in  $G$  induces a total H-morphism  $\bar{\mu} : \bar{S}_v \rightarrow \bar{T}_v$ .

Well-formedness and semantics are defined in a joint induction on the structure of  $G$ , and only minor adjustments to the definition of  $\bar{G}$  for LF+MMT are needed. We begin with the flat syntax.

Firstly, a flat signature  $\%sig T = \{\Sigma\}$  induces a hiding signature  $\bar{T} = (\bar{T}_v, \bar{T}_h)$  as follows:  $\bar{T}_h$  contains all declarations in  $\Sigma$ , and  $\bar{T}_v$  is the largest subsignature of  $\bar{T}_h$  that contains only visible declarations.  $\Sigma$  is well-formed if this is indeed a well-formed  $\text{LFH}$ -object.

Secondly, consider a flat morphism  $\sigma$  and two flat signatures  $S$  and  $T$  in  $G$ .  $\sigma$  induces an H-morphism from  $\bar{S}_v$  to  $\bar{T}_v$  as follows: Its revealed domain is the smallest subsignature of  $\bar{S}_v$  that contains all constants mapped by  $\sigma$ ; its non-hidden domain is the largest subsignature of  $\bar{S}_v$  that contains no constants hidden by  $\sigma$ .  $\sigma$  is well-formed if this is indeed a well-formed H-morphism from  $\bar{S}_v$  to  $\bar{T}_v$ .

Next we define the semantics of the full syntax by flattening an arbitrary signature graph  $G$  to  $\tilde{G}$ . We use the same definition as in [RS09] except for additionally keeping track of hidden declarations.

Firstly, consider a signature  $T$  with a structure  $\%structs : S = \{\sigma\}$ , and consider a declaration of  $c$  in  $\tilde{S}$ . Then  $\tilde{T}$  contains a constant  $s.c$  defined in the same way as for LF+MMT. Moreover,  $s.c$  is hidden in  $\tilde{T}$  if  $s$  is hidden in  $T$ ,  $c$  is hidden in  $S$ , or  $\tilde{\sigma}$  hides  $c$ .

Secondly, consider an occurrence of  $\%structs := \mu$  in  $\sigma$  in a structure or view declaration with domain  $S$ . Since the semantics  $\bar{\mu}$  of  $\mu$  is a total H-morphism, we must consider two cases

for every visible constant  $c$  in  $\tilde{S}$ : if  $c$  is in the revealed domain of  $\bar{\mu}$ , then  $\tilde{\sigma}$  contains  $s.c := \bar{\mu}(c)$  as for LF+MMT; otherwise,  $\tilde{\sigma}$  contains  $\%hide\ s.c$ .

Thirdly, consider an occurrence of  $\%hide\ %struct\ s$  in  $\sigma$  in a structure or view declaration with domain  $S$ . Then  $\tilde{\sigma}$  contains  $\%hide\ s.c$  for every visible constant  $c$  of  $\tilde{S}$ .

Finally, to define well-formedness of signature graphs, we use the same inference system as in [RS09] with the following straightforward restriction for morphisms: In a structure declaration  $\%struct\ s : S = \{\sigma\}$  within  $T$  or in a view declaration  $\%view\ v : S \rightarrow T = \{\sigma\}$ ,  $\tilde{\sigma}$  must be an H-morphism from  $\tilde{S}_v$  to  $\tilde{T}_v$ .  $\tilde{\sigma}$  must be total for views and may be partial for structures. Such structures and views induce edges  $\overline{T.s}$  and  $\overline{v}$  in  $\overline{G}$  in the obvious way.

It is easy to show that well-formedness of the flat syntax is decidable. Moreover, we have

**Theorem 4.**  $\overline{G}$  is a diagram over  $\text{LFH}$  for every well-formed signature graph  $G$ .

*Proof.* This is proved by a straightforward induction on the structure of  $G$ .

The morphisms  $\sigma$  in structures and views may only map symbols of the visible domain. Moreover, they may hide some of these symbols. However, if we inspect the definition of the flattening of a structure  $\%struct\ s : S = \{\sigma\}$ , we see that it imports all constants of  $\overline{S}$  including the hidden ones and including those hidden by  $\sigma$ . Therefore, we have:

**Lemma 5.** Assume a well-formed signature graph with hiding  $G$  containing a structure  $\%struct\ s : S = \{\sigma\}$  in  $T$ . Then  $\overline{T.s} : \overline{S} \xrightarrow{!} \overline{T}$ .

*Proof.* The extension of  $\overline{T.s}$  to  $\overline{S}_h$  maps every constant  $c$  to  $s.c$ .

## 5 Interpreting ASL in LF+MMT

We now introduce the translation from ASL-style structured specifications into LF+MMT. We assume that there is a representation of an institution  $I$  in LF (see Sect. 5.1), such that when translating an ASL-style specification over  $I$  (see Sect. 5.2), the resulting MMT specification is based on this representation. The subsequent subsections deal with proving adequacy of the translation.

### 5.1 Logics

Consider an encoding as in Sect. 3 for an institution  $I$ . We make the following assumptions about the adequacy of the encoding.

**Definition 6.** We say that a foundation  $\mathcal{F}$  is **adequate** if there is (i) a  $\mathcal{F}$ -type  $prop : type$  such that the formal statements of mathematics can be encoded as  $\mathcal{F}$ -terms  $F : prop$  and (ii) a type  $\text{ded } F$  of proofs of  $F$  for every  $F : prop$  such that  $\text{ded } F$  is inhabited iff  $F$  is a provable statement.

This definition is necessarily vague. To make it definite, we can assume that  $\mathcal{F}$  is the encoding of Zermelo-Fraenkel set theory given in [HR10], in which case the terms of type *prop* are first-order formulas over the binary predicate symbol  $\in$ .

**Definition 7.** Assume an adequate  $\mathcal{F}$ . We say that an institution  $I = (\text{Sig}, \text{Sen}, \text{Mod}, \models)$  is **adequately represented** as  $(L^{\text{Syn}}, \mathcal{F}, L^{\text{Mod}}, L^{\text{mod}})$  if there is a functor  $\Phi : \text{Sig} \rightarrow \text{LF}/L^{\text{Syn}}$  such that for every signature  $\Sigma$  (i)  $\Phi(\Sigma) = \Sigma^{\text{Syn}}$  is an extension of  $L^{\text{Syn}}$ , (ii) there is a bijection  $\lceil - \rceil$  mapping  $\Sigma$ -sentences to  $\beta\eta$ -normal  $\Sigma^{\text{Syn}}$ -terms of type  $\text{o}$ , and  $\lceil - \rceil$  is natural with respect to sentence translation  $\text{Sen}(\sigma)$  and morphism application  $\Phi(\sigma)$  (iii) there is a bijection  $\lceil - \rceil$  mapping  $\Sigma$ -models to LF-morphisms  $\Sigma^{\text{Mod}} \rightarrow \mathcal{F}$ , and  $\lceil - \rceil$  is natural with respect to model reduction  $\text{Mod}(\sigma)$  and precomposition with  $\Phi(\Sigma)^{\text{mod}}$ , (iv) satisfaction  $M \models_{\Sigma} F$  holds iff  $\lceil M \rceil$  maps  $\Sigma^{\text{mod}}(\lceil F \rceil)$  to an inhabited  $\mathcal{F}$ -type.

Using the definitions of [Rab10], this can be stated as an institution comorphism from  $I$  to an appropriate institution based on LF.

Our assumption of a bijection between  $I$ -models and LF-morphisms is quite strong. In most cases, not all models will be representable as morphisms. However, using canonical models constructed in completeness proofs, in many cases it will be possible to represent all models up to elementary equivalence.

## 5.2 Specifications

We define a translation from ASL specifications to signature graphs of LF+MMT with hiding. Since the ASL structuring is built over an arbitrary institution, we assume that the underlying institution has already been represented in LF and the representation is adequate.

The translation proceeds by induction on the structure of the specification  $SP$ . However, MMT does not use signature expressions in the way ASL uses specification-building operations; in particular, MMT structures may import only named signatures. Therefore, the translation introduces one MMT signature declaration for every specification-building operation used in  $SP$ . Note that this leads to an increase in size but not to the exponential blow-up incurred when flattening.

The cases of the translation are given in Fig. 2. Every specification-building operation yielding a specification  $SP$  over a signature  $\Sigma$  is translated to two MMT signatures of the form

$$\% \text{sig } N_{\Sigma} = \{ \% \text{struct } l : L^{\text{Syn}}, \lceil \Sigma \rceil \}, \quad \% \text{sig } N_{SP} = \{ \% \text{struct } s : N_{\Sigma}, \lceil SP \rceil \}$$

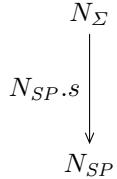
$\lceil \Sigma \rceil$  is a list of declarations representing the visible signature symbols, and similarly  $\lceil SP \rceil$  represents the hidden signature symbols and all axioms. These must refer to the logical symbols of the underlying logic, which is why  $N_{\Sigma}$  starts with an import from  $L^{\text{Syn}}$ .  $N_{\Sigma}$  and  $N_{SP}$  are fresh names generated during the translation.

We will describe the translation case by case visualizing the involved objects using diagrams in LF. First we introduce one simplification of the notation. Recall that technically, the semantics  $\overline{N}_{SP}$  of  $N_{SP}$  is an LFH object  $(\overline{N}_{SPv}, \overline{N}_{SPh})$  and similarly for  $\overline{N}_{\Sigma} = (\overline{N}_{\Sigma v}, \overline{N}_{\Sigma h})$ . A simple induction will show that  $N_{\Sigma}$  never contains hiding and that  $\overline{N}_{SP}.s : \overline{N}_{\Sigma v} = \overline{N}_{\Sigma h} \rightarrow \overline{N}_{SPv}$  is an isomorphism in LF. Therefore, we will always write  $N_{\Sigma}$  instead of  $\overline{N}_{\Sigma v}$ ,  $N_{SP}$  instead of  $\overline{N}_{SPh}$ , and  $N_{SP}.s$  instead of  $\overline{N}_{SP}.s$ .

|   |   |
|---|---|
| $SP := (\Sigma, \{F_1, \dots, F_n\})$   | <i>Basic</i>  |
| $\%sig N_\Sigma = \{\%struct l : L^{Syn}, \lceil \Sigma \rceil\}$   |   |
| $\%sig N_{SP} = \{\%struct s : N_\Sigma, \%hide a_1 : ded \lceil F_1 \rceil, \dots, \%hide a_n : ded \lceil F_n \rceil\}$ |   |
| $\Sigma = Sig[SP_1] = Sig[SP_2]$  | $\%sig N_\Sigma = \{\lceil \Sigma \rceil\}$   |
| $SP' := SP_1 \cup SP_2$   | $\%sig N_{SP_1} = \{\%struct s_1 : N_\Sigma, \lceil SP_1 \rceil\}$                                |
|   | $\%sig N_{SP_2} = \{\%struct s_2 : N_\Sigma, \lceil SP_2 \rceil\}$                                |
|   | <i>Union</i>  |
| $N_{SP'} = \{$  |   |
| $\%struct s : N_\Sigma,$  |   |
| $\%struct t_1 : N_{SP_1} = \{\%struct s_1 := s\}, \%struct t_2 : N_{SP_2} = \{\%struct s_2 := s\}$                        |   |
| $\}$  |   |
| $\sigma : \Sigma \rightarrow \Sigma'$   | $\%sig N_\Sigma = \{\%struct l : L^{Syn}, \lceil \Sigma \rceil\}$                                 |
| $SP' := \sigma(SP)$   | $\%sig N_{\Sigma'} = \{\%struct l' : L^{Syn}, \lceil \Sigma' \rceil\}$                            |
|   | $\%sig N_{SP} = \{\%struct s : N_\Sigma, \lceil SP \rceil\}$                                      |
|   | $\%view N_\sigma : N_\Sigma \rightarrow N_{\Sigma'} = \{\%struct l := l', \lceil \sigma \rceil\}$ |
| $\%sig N_{SP'} = \{\%struct s' : N_{\Sigma'}, \%struct t : N_{SP} = \{\%struct s := N_\sigma s'\}\}$                      | <i>Transl</i>   |
| $\sigma : \Sigma \hookrightarrow \Sigma'$   | $\%sig N_\Sigma = \{\%struct l : L^{Syn}, \lceil \Sigma \rceil\}$                                 |
| $dom(\Sigma) = \{c_1, \dots, c_m\}$   | $\%sig N_{\Sigma'} = \{\%struct l' : L^{Syn}, \lceil \Sigma' \rceil\}$                            |
| $dom(\Sigma') \setminus dom(\Sigma) = \{h_1, \dots, h_n\}$  | $\%sig N_{SP'} = \{\%struct s' : N_{\Sigma'}, \lceil SP' \rceil\}$                                |
| $SP := \sigma^{-1}(SP')$  | <i>Hide</i>   |
| $N_{SP} = \{\%struct s : N_\Sigma,$   |   |
| $\%struct t : N_{SP'} = \{\%struct s'.l' := s.l, s'.c_1 := s.c_1, \dots, s'.c_m := s.c_m,$                                |   |
| $\%hide s'.h_1, \dots, \%hide s'.h_n\}$   |   |
| $\}$  |   |

**Fig. 2.** Translation of ASL specifications to LF+MMT with Hiding

The rule *Basic* translates basic specification  $SP = (\Sigma, E)$  using the LF representation of the underlying institution.  $\lceil \Sigma \rceil$  contains one declaration for every non-logical symbol declared in  $\Sigma$ . For example, if  $L^{Syn}$  encodes first-order logic and has a declaration  $i : type$  for the universe, a binary predicate symbol  $p$  in  $\Sigma$  leads to a declaration  $p : l.i \rightarrow l.o$  in  $\lceil \Sigma \rceil$ . All axioms  $F \in E$ , lead to a declaration



$a : \text{ded} \lceil F \rceil$  where  $a$  is a fresh name. This has the effect that axioms are always hidden, which simplifies the notation significantly; it is not harmful because the semantics of ASL does not depend on whether an axiom is hidden or visible.

$$\begin{array}{ccc} N_\Sigma & \xrightarrow{N_{SP_1}.s_1} & N_{SP_1} \\ N_{SP_2}.s_2 \downarrow & \searrow N_{SP'}.s & \downarrow N_{SP'}.t_1 \\ N_{SP_2} & \xrightarrow[N_{SP'}.t_2]{} & N_{SP'} \end{array}$$

The rule *Union* assumes translations of  $\Sigma$ ,  $SP_1$ , and  $SP_2$  and creates the translation of  $SP' = SP_1 \cup SP_2$  by instantiating  $N_{SP_1}$  and  $N_{SP_2}$  in such a way that they share  $N_\Sigma$ . The semantics of LF+MMT guarantees that the resulting diagram on the left is a pushout in LF.

The rule *Transl* translates  $SP' = \sigma(SP)$  assuming that  $\sigma$  and  $SP$  have been translated already. The signature morphism  $\sigma$  is translated to a view in a straightforward way. Recall that  $N_\Sigma$  and  $N_{\Sigma'}$  contain no hidden declarations or axioms so that

$N_\sigma$  is a (total) morphism in LF. The resulting diagram is the left diagram below; it is again a pushout in LF.

Similarly, the rule *Hide* translates  $SP = \sigma^{-1}(SP')$  assuming that  $SP$  has been translated already. As  $\sigma : \Sigma \hookrightarrow \Sigma'$  is an inclusion, we only need to know the names  $c_i$  of the symbols in  $\Sigma$  and the names  $h_j$  of the symbols in  $\Sigma' \setminus \Sigma$ , which are to be hidden. Then we can form  $N_{SP}$  by importing from  $N_{SP'}$  and mapping all symbols that remain visible to their counterparts in  $N_{SP}$  and hiding the remaining symbols. The resulting diagram is the right diagram below. Note that by Lem. 5,  $N_{SP}.t$  extends to a total LF morphism  $N_{SP}.t^*$ ; moreover, it is easy to verify that  $N_{SP}.t^*$  is an isomorphism.

$$\begin{array}{ccc} N_\Sigma & \xrightarrow{N_\sigma} & N_{\Sigma'} \\ N_{SP}.s \downarrow & & \downarrow N_{SP'}.s' \\ N_{SP} & \xrightarrow[N_{SP'}.t]{} & N_{SP'} \\ \text{and} \\ N_\Sigma & \hookrightarrow & N_{\Sigma'} \\ N_{SP}.s \downarrow & & \downarrow N_{SP'}.s' \\ N_{SP} & \xleftarrow[N_{SP}.t^*]{} & N_{SP'} \end{array}$$

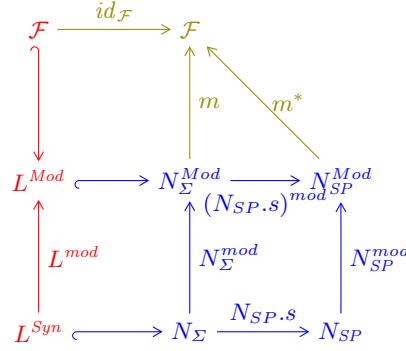
### 5.3 Adequacy for Specifications

The general idea of the encoding of models is given in Fig. 3. The diagram corresponds to the one from Sect. 3.3 except that both  $N_\Sigma$  and  $N_{SP}$  are drawn.  $(N_{SP}.s)^{\text{mod}}$  arises as the unique factorization through the pushout  $N_\Sigma^{\text{Mod}}$ .

Our result is that models  $M \in \text{Mod}^I[SP] \subseteq \text{Mod}^I(\Sigma)$  are encoded as LF morphisms  $m : N_\Sigma^{\text{Mod}} \rightarrow \mathcal{F}$  that factor through  $N_{SP}^{\text{Mod}}$ .

The translation of ASL to MMT yields pushouts between LF signatures extending  $L^{\text{Syn}}$ , but models are stated in terms of signatures extending  $L^{\text{Mod}}$ . Therefore, we use the following simple lemma:

**Lemma 8.** *If the left diagram below is a pushout in LF, then so is the right one.*



**Fig. 3.** Representation of Models in the Presence of Hiding

$$\begin{array}{ccc}
 \Sigma & \xrightarrow{\sigma_1} & \Sigma_1 \\
 \downarrow \sigma_2 & & \downarrow \tau_2 \\
 \Sigma_2 & \xrightarrow[\tau_1]{} & \Sigma' \\
 & & \\
 \Sigma^{Mod} & \xrightarrow{\sigma_1^{mod}} & \Sigma_1^{Mod} \\
 & \downarrow \sigma_2^{mod} & \downarrow \tau_2^{mod} \\
 \Sigma_2^{Mod} & \xrightarrow[\tau_1^{mod}]{} & \Sigma'^{Mod}
 \end{array}$$

*Proof.* This is shown with a straightforward diagram chase.

Then we are ready to state our main result:

**Theorem 9.** *Let  $I$  be an institution that is adequately represented in LF. Then for any signature  $\Sigma$ , any ASL-structured specification  $SP$  with  $\text{Sig}[SP] = \Sigma$ , and any  $\Sigma$ -model  $M$*

$$M \in \text{Mod}^I[SP] \quad \text{iff} \quad \text{exists } m^* : N_{SP}^{Mod} \rightarrow \mathcal{F} \text{ such that } (N_{SP}.s)^{mod}; m^* = \lceil M \rceil$$

*Proof.* The proof is done by induction on the structure of  $SP$ . All cases will refer to the corresponding diagrams in Sect. 5.2.

**Case  $SP = (\Sigma, E)$ :**

For the base case, the conclusion follows directly from the assumption that the representation of  $I$  in LF is adequate.

**Case  $SP = SP_1 \cup SP_2$ :**

Let  $M \in \text{Mod}[SP]$  and  $m := \lceil M \rceil : N_{\Sigma}^{Mod} \rightarrow \mathcal{F}$ . We want to factor  $m$  through  $N_{SP}^{Mod}$ . By definition, we have that  $M \in \text{Mod}[SP_1]$  and  $M \in \text{Mod}[SP_2]$ . By the induction hypothesis for  $SP_1$  and  $SP_2$ , we get that there are morphisms  $m_i : N_{SP_i}^{Mod} \rightarrow \mathcal{F}$  such that  $m = (N_{SP_i}.s)^{mod}; m_i$ . Using the pushout property we get a unique morphism  $m^* : N_{SP}^{Mod} \rightarrow \mathcal{F}$  such that  $(N_{SP_i}.s)^{mod}; (N_{SP'}.t_i)^{mod}, m^* = m$  which gives us the needed factorization.

For the reverse inclusion, let  $m := \lceil M \rceil : N_{\Sigma}^{Mod} \rightarrow \mathcal{F}$  represent a  $\Sigma$ -model  $M$  and factor as  $(N_{SP.s})^{mod}; m^*$ . Notice that by composing  $(N_{SP.t_i})^{mod}$  with  $m^*$  we get morphisms  $m_i : N_{SP_i}^{Mod} \rightarrow \mathcal{F}$ . By using the induction hypothesis,  $M$  is then a model of both  $SP_1$  and  $SP_2$  and by definition  $M$  is a model of  $SP$ .

**Case  $SP' = \sigma(SP)$ :**

Let  $M' \in Mod[SP']$  and  $m' := \lceil M' \rceil : N_{\Sigma'}^{Mod} \rightarrow \mathcal{F}$ . We want to prove that there is  $m'^* : N_{SP'}^{Mod} \rightarrow \mathcal{F}$  such that  $m' = (N_{SP'.s'})^{mod}; m'^*$ . By definition  $M'|_{\sigma} \in Mod[SP]$ . By induction hypothesis for  $SP'$  we get a morphism  $m := \lceil M' \rceil|_{\sigma} : N_{\Sigma}^{Mod} \rightarrow \mathcal{F}$  and a morphism  $m^* : N_{SP}^{Mod} \rightarrow \mathcal{F}$  such that  $(N_{SP.s})^{mod}; m^* = m = (N_{\sigma})^{mod}; m'$ , where the latter equality holds due to the definition of model reduct. Using the pushout property we get the desired  $m'^*$ .

For the reverse inclusion, assume  $m' := \lceil M' \rceil : N_{\Sigma'}^{Mod} \rightarrow \mathcal{F}$  that factors as  $(N_{SP'.s'})^{mod}; m'^*$ . Then  $(N_{SP.s})^{mod}; (N_{SP'.t})^{mod}; m'^*$  factors through  $N_{SP}^{Mod}$  and thus by induction hypothesis the reduct of  $M'$  is an  $SP$ -model, which by definition means that  $M'$  is an  $SP'$ -model.

**Case  $SP = \sigma^{-1}(SP')$ :**

Let  $M$  be an  $SP$ -model and let  $m := \lceil M \rceil : N_{\Sigma} \rightarrow \mathcal{F}$ . We want to prove that  $m$  factors through  $N_{SP}^{Mod}$ . By definition  $M$  has an expansion  $M'$  to an  $SP'$ -model. By induction hypothesis, there are morphisms  $m' := \lceil M' \rceil : N_{\Sigma'}^{Mod} \rightarrow \mathcal{F}$  and  $m'^* : N_{SP'}^{Mod} \rightarrow \mathcal{F}$  such that  $(N_{SP'.s'})^{mod}; m'^* = m'$ . Then  $m = (N_{SP.s})^{mod}; (N_{SP.t^{*-1}})^{mod}; m'^*$ .

For the reverse inclusion, let  $m := \lceil M \rceil$  be a morphism that factors as  $(N_{SP.s})^{mod}; m^*$ . We need to prove that  $M$  has an expansion to a  $SP'$ -model. We obtain it by applying the induction hypothesis to  $m' := (N_{SP'.s'})^{mod}; (N_{SP.t^*})^{mod}; m^*$ .

Corresponding to the adequacy for models, we prove the adequacy for theorems:

**Theorem 10.** *Let  $I$  be an institution and assume that  $I$  has been represented in LF in an adequate way. Then for any signature  $\Sigma$ , any ASL-structured specification  $SP$  with  $Sig[SP] = \Sigma$ , and any  $\Sigma$ -sentence  $F$*

$$F \in Thm^I[SP] \quad \text{iff} \quad N_{SP.s}(l.\text{ded} \lceil F \rceil) \text{ inhabited over } N_{SP}$$

*Proof.* This is proved by induction on  $SP$ . For the base case, this follows from the adequacy assumption. For the remaining cases, it follows easily after observing that due to Lem. 5 structures always translate (possibly hidden) theorems to (possibly hidden) theorems. We omit the details.

Finally we remark that  $N_{nf(SP)}$  is a flat H-signature that is isomorphic to the flattening of  $N_{SP}$  where  $nf(SP)$  is the normal form of  $SP$  as defined in [Bor02]. This can also be proved by induction and then used to prove the above results.

#### 5.4 Adequacy for Refinements

We want to give a syntactical criterion for refinement  $SP \rightsquigarrow_{\Sigma} SP'$ . Consider the diagram on the right.  $SP \rightsquigarrow_{\Sigma} SP'$  states that for all  $m$ , if  $m'^*$  exists, then some  $m^*$  exists such that the diagram commutes. Clearly, this holds if there is an LF morphism  $\rho : N_{SP}^{Mod} \rightarrow N_{SP'}^{Mod}$ .

```

    \begin{CD}
    N_{SP} @>N_{SP}^{Mod}>> N_{SP'}^{Mod} \\
    @V N_{SP.s} VV @VV m^* V \\
    N_{\Sigma} @>N_{\Sigma}^{Mod}>> N_{\Sigma}^{Mod} @> m >> \mathcal{F} \\
    @V N_{SP'.s} VV @V (N_{SP'.s})^{mod} VV @VV m'^* V \\
    N_{SP'} @>N_{SP'}^{Mod}>> N_{SP'}^{Mod}
    \end{CD}
    \rho
  
```

If  $\mathcal{F}$  has some additional technical properties, we can also prove the opposite implication:

**Theorem 11.** *Let  $I$  be an institution that is adequately encoded in LF. Moreover, assume that (i)  $\mathcal{F}$  can express  $I$ -models as tuples of their components, and (ii) whenever  $\mathcal{F}$  can prove the existence of a model of a finite  $I$ -theory,  $\mathcal{F}$  can also express some model of that theory.*

*Then for ASL-specifications  $SP$  and  $SP'$  over the signature  $\Sigma$ , we have that  $SP \rightsquigarrow_{\Sigma} SP'$  iff there is an LF morphism  $\rho : N_{SP}^{Mod} \rightarrow N_{SP'}^{Mod}$  such that  $(N_{SP}.s)^{mod}; \rho = (N_{SP'}.s)^{mod}$ .*

*Proof.* The right-to-left implication follows immediately using Thm. 9.

For the left-to-right implication, we assume  $SP \rightsquigarrow_{\Sigma} SP'$  and work within  $N_{SP'}^{Mod}$ . Due to the adequacy of  $\mathcal{F}$ ,  $\lceil SP \rightsquigarrow_{\Sigma} SP' \rceil$  is a provable statement of  $\mathcal{F}$  and thus of  $N_{SP'}^{Mod}$  (iii). Using (i), we can tuple the declarations in  $N_{SP'}^{Mod} \setminus \mathcal{F}$  (excluding the axioms) and obtain a  $\Sigma'$  model  $m$  as a term over  $\mathcal{F}$ ; using the axioms in  $N_{SP'}^{Mod} \setminus \mathcal{F}$ , we can prove that  $m'$  is an  $SP'$  model. Using (iii), we show that  $m$  is also an  $SP$  model. Then using (ii), we obtain an expression  $m^*$  over  $N_{SP'}^{Mod}$  that expresses a model of  $N_{SP'}$ . Finally, using (i), we can project out the components of  $m^*$ . The morphism  $\rho$  maps every symbol of  $N_{SP'}^{Mod} \setminus \mathcal{F}$  (excluding the axioms) to the corresponding components; it maps all axioms to proofs about  $m^*$ .

The assumption (i) of this theorem is mild. In fact, if (i) did not hold, it would be dubious how the foundation can express institutions and models at all. The assumption (ii) is more restricting. It holds for example in foundations that have a choice operator such as ZF with global choice function or higher-order logic. It is also possible to establish (ii) for individual institutions by giving a constructive model existence proof.

## 6 Conclusion

With the translation presented in this paper, it is possible to encode ASL- and CASL-style structured specifications with hiding in proof theoretic logical frameworks. This provides a new perspective on structured specifications that emphasizes constructive and mechanizable notions. Our translation is given for MMT-structured LF, but it easily generalizes to other MMT-structured logical frameworks.

Our encoding can be generalized to specifications given as development graphs. In this context, our representation theorem for refinements can be strengthened to represent the hiding theorem links of [MAH06]. Even heterogeneous specifications [MT09, MML07] can be covered. As LF+MMT uses the same structuring operations for logics as for theories, this requires only the representation of the involved logics and logic translations in LF.

A theorem very similar to our representation theorem for refinements can be obtained for conservative extensions. This permits the interpretation of the proof calculus for refinement given in [Bor02]. In particular, the rules using an oracle for conservative extensions can be represented elegantly as the composition of LF signature morphisms.

The translation to MMT also has the benefit that we can re-use the infrastructure provided by languages like OMDoc [Koh06] (an XML-based markup format for mathematical documents) and tools like TNTBase [ZK09] (a versioned XML database for OMDoc documents that supports complex searches and queries, e.g., via XQuery). Further tools developed along these lines are

the JOBAD framework (a JavaScript library for interactive mathematical documents), which will provide a web-based frontend for the Heterogeneous Tool Set, GMoc (a change management system), DocTip (a document and tool integration platform) and integration with the Eclipse framework (an integrated development environment).

## References

- AHMS99. S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an Evolutionary Formal Software-Development Using CASL. In D. Bert, C. Choppy, and P. Mosses, editors, *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1999.
- AKKB99. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner. *Algebraic Foundations of Systems Specification*. Springer, 1999.
- Bar92. H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.
- BG80. R. Burstall and J. Goguen. The semantics of Clear, a specification language. In Dines Bjørner, editor, *Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*, pages 292–332. Springer Berlin / Heidelberg, 1980.
- Bor02. Tomasz Borzyszkowski. Logical systems for structured specifications. *Theor. Comput. Sci.*, 286(2):197–245, 2002.
- CHK<sup>+</sup>10. M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, F. Rabe, and K. Sojakova. Logical Frameworks in Hets, 2010. Workshop on Abstract Development Techniques.
- GB92. J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.
- GR04. J. Goguen and G. Rosu. Composing Hidden Information Modules over Inclusive Institutions. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl*, pages 96–123. Springer, 2004.
- HHP93. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
- HR10. F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. Under review, see [http://kwarc.info/frabe/Research/HR\\_folsound\\_10.pdf](http://kwarc.info/frabe/Research/HR_folsound_10.pdf), 2010.
- HST94. R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.
- IR10. M. Iancu and F. Rabe. Formalizing Foundations of Mathematics. Under review, see [http://kwarc.info/frabe/Research/IR\\_foundations\\_10.pdf](http://kwarc.info/frabe/Research/IR_foundations_10.pdf), 2010.
- KMR09. M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN Project, 2009. See <https://trac.omdoc.org/LATIN/>.
- Koh06. M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*. Number 4180 in Lecture Notes in Artificial Intelligence. Springer, 2006.
- MAH06. T. Mossakowski, S. Autexier, and D. Hutter. Development graphs - Proof management for structured specifications. *J. Log. Algebr. Program.*, 67(1–2):114–145, 2006.
- ML74. P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the '73 Logic Colloquium*, pages 73–118. North-Holland, 1974.
- MML07. T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522, 2007.
- Mos04. P. D. Mosses, editor. *CASL Reference Manual*. Number 2960 (IFIP Series) in LNCS. Springer Verlag, 2004.

- MT09. Till Mossakowski and Andrzej Tarlecki. Heterogeneous logical environments for distributed specifications. In Andrea Corradini and Ugo Montanari, editors, *WADT 2008*, volume 5486 of *Lecture Notes in Computer Science*, pages 266–289. Springer, 2009.
- Pau94. L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
- Rab08. F. Rabe. *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen, 2008. Available at <http://kwarc.info/frabe/Research/phdthesis.pdf>.
- Rab10. F. Rabe. A Logical Framework Combining Model and Proof Theory. To be submitted, see [http://kwarc.info/frabe/Research/rabe\\_combining\\_09.pdf](http://kwarc.info/frabe/Research/rabe_combining_09.pdf), 2010.
- RS09. F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, pages 40–48. ACM Press, 2009.
- ST88. D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165–210, 1988.
- ST10. Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Specification and Formal Program Development*. To appear in Springer-Verlag, 2010.
- SW83. Donald Sannella and Martin Wirsing. A kernel language for algebraic specification and implementation. In *ADT*, 1983.
- Wir86. Martin Wirsing. Structured algebraic specifications: A kernel language. *Theor. Comput. Sci.*, 42:123–249, 1986.
- ZK09. V. Zheludev and M. Kohlhase. TNTBase: a Versioned Storage for XML. In *Proceedings of Balisage: The Markup Conference 2009*, volume 3. Mulberry Technologies, Inc., 2009.

# A Foundational View on Integration Problems

Florian Rabe<sup>1</sup> and Michael Kohlhase<sup>1</sup> and Claudio Sacerdoti Coen<sup>2</sup>

<sup>1</sup> Computer Science, Jacobs University, Bremen (DE)  
`initial.lastname@jacobs-university.de`

<sup>2</sup> Department of Computer Science, University of Bologna (IT)  
`sacerdot@cs.unibo.it`

**Abstract.** The integration of reasoning and computation services across system and language boundaries has been mostly treated from an engineering perspective. In this paper we take a foundational point of view. We identify the following form of integration problems: an informal (mathematical; i.e., logically underspecified) specification has multiple concrete formal implementations between which queries and results have to be transported. The integration challenge consists in dealing with the implementation-specific details such as additional constants and properties. We pinpoint their role in safe and unsafe integration schemes and propose a proof-theoretic solution based on modular theory-graphs that include the meta-logical foundations. This also gives a clean conceptual basis for earlier attempts that explain integration via “content/semantic markup”.

## 1 Introduction

The aim of integrating Computer Algebra Systems (CAS) and Deduction Systems (DS) is twofold: to bring the efficiency of CAS algorithms to DS (without sacrificing correctness) and to bring the correctness assurance of the proof theoretic foundations of DS to CAS computations (without sacrificing efficiency). In general, the integration of computation and reasoning systems can be organized either by extending the internals of one systems by methods (data structure and algorithms) from the other, or by passing representations of mathematical objects and system state between independent systems, thus delegating parts of the computation to more efficient or secure platforms. We will deal with the latter approach here, which again has two distinct sets of problems. The first addresses engineering problems and revolves about communication protocol questions like shared state, distributed garbage collection, and translating input syntaxes of the different systems. The syntax questions have been studied extensively in the last decade and led to universal content markup languages for mathematics like MathML and OpenMath to organize communication. The second set of problems comes from the fact that passing mathematical objects between systems can only be successful if their meaning is preserved in the communication. This meaning is given via logical consequence in the logical system together with the axioms and definitions of (or inscribed in) the respective systems.

We will address this in the current paper, starting from the observation that content level communication between mathematical systems, to be effective, cannot always respect logical consequence. On the other hand, there is the problem of trusting the communication itself, that boils down to studying the preservation of logical consequence. Surprisingly, this problem has not received in the literature the attention it deserves. Moreover, the problem of faithful safe communication, which preserves not only the consequence relation but also the intuitive meaning of a formal object, is not even always perceived as a structural problem of content level languages.

For example, people with a strong background in first order logic tend to assume that faithful and safe communication can always be achieved simply by strengthening the specifications; others believe that encoding logical theories is already sufficient for safe communication and do not appreciate that the main problem is just moved to faithfulness. Several people from the interactive theorem proving world have raised concerns about trusting CAS and solved the issue by re-checking the results or the traces of the computation (here called proof sketches). Sometimes this happens under the assumption that the computation is already correct and just needs to be re-checked, neglecting the interesting case when the proof sketch cannot be refined to a valid proof (or computation) without major patching (see [DdC99] for a special case).

In section 3, we analyze the integration problem for mathematical systems from a formal position — basing our deliberations on the consequence relation. Our integration framework arises by adding some key innovations to the MMT language described in [RK10], which arises as a generalization of OpenMath and a clarification of OMDoc. Therefore, we sketch the MMT framework first in Sect. 2. Then we describe how integration can be realized our framework using partial theory morphisms in Sect. 4. Sect. 5 discusses related work and Sect. 6 concludes the paper.

## 2 The MMT Language

Agreeing on a common syntax like OpenMath is the first step towards system integration. This already enables a number of structural services such as storage and transport or editing and browsing that they do not depend on the semantics of the processed expressions. But while we have a good solution for a joint syntax, it is significantly harder to agree on a joint semantics. Fixing a semantics for a system requires a foundational commitment that excludes systems based on other foundations. The weakness of the (standard) OpenMath content dictionaries can be in part explained by this problem: The only agreeable content dictionaries are those where any axioms (formal or informal) are avoided that would exclude some foundations.

MMT was designed to overcome this problem by placing it in between frameworks like OpenMath and OMDoc on the one hand and logical frameworks like LF and CIC on the other hand. The basic idea is that a system’s foundation itself is represented as a content dictionary. Thus, both meta and object lan-

guage are represented uniformly as MMT *theories*. Furthermore, *theory morphisms* are employed to translate between theories, which makes MMT expressive enough to represent translation between meta-languages and thus to support cross-foundation integration. As MMT permits the representation of logics as theories and internalizes the meta-relation between theories, this provides the starting point to analyze the cross-foundation integration challenge within a formal framework.

*Syntax* We will work with a very simple fragment of the MMT language that suffices for our purposes, and refer to [RK10] for the full account. It is given by the following grammar where  $[-]$  denotes optional parts and  $T$ ,  $v$ ,  $c$ , and  $x$  are identifiers:

|               |             |       |  |
|---------------|-------------|-------|--|
| Theory graph  | $\gamma$    | $::=$ | $\cdot \mid \gamma, T \stackrel{[T]}{=} \{\vartheta\} \mid \gamma, v : T \rightarrow T \stackrel{v}{=} \{\sigma\}$ |
| Theory body   | $\vartheta$ | $::=$ | $\cdot \mid \vartheta, c[: O] [= O']$  |
| Morphism body | $\sigma$    | $::=$ | $\cdot \mid \sigma, c \mapsto O$   |
| Objects       | $O$         | $::=$ | OpenMath objects   |
| Morphisms     | $\mu$       | $::=$ | $v \mid id_T \mid \mu \circ \mu$   |
| Contexts      | $C$         | $::=$ | $x_1 : O_1, \dots, x_n : O_n$  |
| Substitutions | $s$         | $::=$ | $x_1 := O_1, \dots, x_n := O_n$  |

In particular, we omit the module system of MMT that permits imports between theories.

$T \stackrel{L}{=} \{\vartheta\}$  declares a *theory*  $T$  with *meta-theory*  $L$  defined by the list  $\vartheta$  of symbol declarations. The intuition of meta-theories is that  $L$  is the meta-language that declares the foundational symbols used to type and define the symbol declarations in  $\vartheta$ .

All *symbol declarations* in a theory body are of the  $c : O = O'$ . This declares a new symbol  $c$  where both the type  $O$  and the definiens  $O'$  are optional. If given, they must be  $T$ -objects, which are defined as follows. A symbol is called *accessible* to  $T$  if it is declared in  $T$  or accessible to the meta-theory of  $T$ . An OpenMath object is called a  *$T$ -object* if it only uses symbols that are accessible to  $T$ .

*Example 1.* Consider the natural numbers defined within the calculus of constructions (see [BC04]). We represent this in MMT using a theory **CIC** declaring untyped, undefined symbols such as **Type**,  $\lambda$  and  $\rightarrow$ . Then **Nat** is defined as a theory with meta-theory **CIC** giving symbol declarations such as  $\text{N} : \text{OMS(cd = CIC, name = Type)}$  or  $\text{succ} : \text{OMA}(\text{OMS(cd = CIC, name = \(\rightarrow\)), OMS(cd = Nat, name = N), OMS(cd = Nat, name = N))}$ .

*$S$ -contexts*  $C$  are lists of variable declarations  $\dots, x_i : O_i, \dots$  for  $S$ -objects  $O_i$ .  *$S$ -substitutions*  $s$  for an  $S$ -context  $C$  are lists of variable assignments  $\dots, x_i := o_i, \dots$ . In an object  $O$  in context  $C$ , exactly the variables in  $C$  may occur freely; then for a substitution  $s$  for  $C$ , we write  $O[s]$  for the result of replacing every free occurrence of  $x_i$  with  $o_i$ .

Relations between MMT theories are expressed using theory morphisms. Given two theories  $S$  and  $T$ , a *theory morphism* from  $S$  to  $T$  is declared using  $v : S \rightarrow T \stackrel{l}{=} \{\sigma\}$ . Here  $\sigma$  must contain one assignment  $c \mapsto O$  for every

symbol  $c$  declared in the body of  $S$ , and for some  $T$ -objects  $O$ . If  $S$  and  $T$  have meta-theories  $L$  and  $M$ , then  $v$  must also include a meta-morphism  $l : L \rightarrow M$ .

Every  $v : S \rightarrow T \stackrel{l}{=} \{\sigma\}$  induces a *homomorphic extension*  $v(-)$  that maps  $S$ -objects to  $T$ -objects.  $v(-)$  is defined by induction on the structure of OpenMath objects. The base case  $v(c)$  for a symbol  $c$  is defined as follows: If  $c$  is accessible to the meta-theory of  $S$ , we put  $v(c) := l(c)$ ; otherwise, we must have  $c \mapsto O$  in  $\sigma$ , and we put  $v(c) := O$ .  $v(-)$  also extends to contexts and substitutions in the obvious way.

By experimental evidence, all declarative languages for mathematics currently known can be represented naturally in MMT. In particular, MMT uses the Curry-Howard representation [CF58][How80] of propositions as types and proofs as terms. Thus, an axiom named  $a$  asserting  $F$  is a special cases of a symbol  $a$  of type  $F$ , and a theorem named  $t$  asserting  $F$  with proof  $p$  is a special case of a symbol  $t$  with type  $F$  and definiens  $p$ . All inference rules needed to form  $p$ , are symbols declared in the meta-theory.

*Semantics* The use of meta-theories makes the logical foundation of a system part of an MMT theory and makes the syntax of MMT foundation-independent. The analogue for the semantics is more difficult to achieve: The central idea is that the semantics of MMT is parametric in the semantics of the foundation.

To make this precise, we call a theory without a meta-theory *foundational*. A *foundation* for MMT consists of a foundational theory  $L$  and two judgments for typing and equality of objects:

- $\gamma; C \vdash_T O : O'$  states that  $O$  is a  $T$ -object over  $C$  typed by the  $T$ -object  $O'$ ,
- $\gamma; C \vdash_T O = O'$  states the equality of two  $T$ -objects over  $C$ ,

defined for an arbitrary theory  $T$  declared in  $\gamma$  with meta-theory  $L$ . In particular, MMT does not distinguish terms, types, and values at higher universes — all expressions are OpenMath objects with an arbitrary binary typing relation between them. We will omit  $C$  when it is empty.

These judgments are similar to those used in almost all declarative languages, except that we do not commit to a particular inference system — all rules are provided by the foundation and are transparent to MMT except for the rules for the base cases of  $T$ -objects:

$$\frac{T \stackrel{L}{=} \{\vartheta\} \text{ in } \gamma \quad c : O = O' \text{ in } \vartheta}{\gamma \vdash_T c : O} \mathcal{T}_! \qquad \frac{T \stackrel{L}{=} \{\vartheta\} \text{ in } \gamma \quad c : O = O' \text{ in } \vartheta}{\gamma \vdash_T c = O'} \mathcal{T}_=$$

and accordingly if  $O$  or  $O'$  are omitted. For example, adding the usual rules for the calculus of constructions yields a foundation for the foundational theory **CIC**.

Given a foundation, MMT defines (among others) the two judgments

- $\gamma \vdash \mu : S \rightarrow T$  states that  $\mu$  is a theory morphism from  $S$  to  $T$ ,
- if  $\gamma \vdash \mu_i : S \rightarrow T$ , then  $\gamma \vdash \mu_1 = \mu_2$  states that  $\vdash_T \mu_1(c) = \mu_2(c)$  for all symbols  $c$  that are accessible to  $S$ ,
- $\gamma \vdash_S s : C$  states that  $s$  is a well-typed for  $C$ , i.e., for every  $x_i := o_i$  in  $s$  and  $x_i : O_i$  in  $C$ , we have  $\gamma \vdash_S o_i : O_i$ ,

–  $\gamma \vdash \mathcal{G}$  states that  $\mathcal{G}$  is a well-formed theory graph.

In the sequel, we will omit  $\gamma$  if it is clear from the context.

The most important MMT rule for our purposes is the rule that permits adding an assignment to a theory morphism: If  $S$  contains a declaration  $c : O_1 = O_2$ , then a theory morphism  $v : S \rightarrow T \stackrel{l}{=} \{\sigma\}$  may contain an assignment  $c \mapsto O$  only if  $\vdash_T O : v(O_1)$  and  $\vdash_T O = v(O_2)$ . The according rule applies if  $c$  has no type or no definiens. Of course, this means that assignments  $c \mapsto O$  are redundant if  $c$  has a definiens; but it is helpful to state the rule in this way to prepare for our definitions below.

Due to these rules, we obtain that if  $\gamma \vdash \mu : S \rightarrow T$  and  $\vdash_S O : O'$  or  $\vdash_S O = O'$ , then  $\vdash_T \mu(O) : \mu(O')$  and  $\vdash_T \mu(O) = \mu(O')$ , respectively. Thus, typing and equality are preserved along theory morphisms.

Due to the Curry-Howard representation, this includes the preservation of provability:  $\vdash_T p : F$  states that  $p$  is a well-formed proof of  $F$  in  $T$ . And if  $S$  contains an axiom  $a : F$ , a morphism  $\mu$  from  $S$  to  $T$  must map  $a$  to a  $T$ -object of type  $\mu(F)$ , i.e., to a  $T$ -proof of  $\mu(F)$ . This yields the well-known intuition of a theory morphism. In particular, if  $\mu$  is the identity on those symbols that do not represent axioms, then  $\vdash \mu : S \rightarrow T$  implies that every  $S$ -theorem is an  $T$ -theorem.

### 3 Integration Challenges

In this section, we will develop some general intuitions about system integration and then give precise definitions in MMT. A particular strength of MMT is that we can give these precise definition without committing to a particular foundational system and thus without loss of generality.

The typical integration situation is that we have two systems  $\mathcal{S}_i$  for  $i = 1, 2$  that implement a shared specification  $Spec$ . For example, these systems can be computer algebra systems or (semi-)automated theorem provers. Our integration goal is to move problems and results between  $\mathcal{S}_1$  and  $\mathcal{S}_2$ .

*Specifications and Systems* Let us first assume a single system  $\mathcal{S}$  implementing  $Spec$ , whose properties are given by logical consequence relations  $\Vdash_{Spec}$  and  $\Vdash_{\mathcal{S}}$ . We call  $\mathcal{S}$  *sound* if  $\Vdash_{\mathcal{S}} F$  implies  $\Vdash_{Spec} F$  for every formula  $F$  in the language of  $Spec$ . Conversely, we call  $\mathcal{S}$  *complete* if  $\Vdash_{Spec} F$  implies  $\Vdash_{\mathcal{S}} F$ .

While these requirements seem quite natural at first, they are too strict for practical purposes. It is well-known that soundness fails for many CASs, which compute wrong results by not checking side conditions during simplification. Reasons for incompleteness can be theoretical — e.g., when  $\mathcal{S}$  is a first-order prover and  $Spec$  a higher-order specification — or practical — e.g., due to resource limitations.

Moreover, soundness also fails in the case of underspecification:  $\mathcal{S}$  is usually much stronger than  $Spec$  because it must commit to concrete definitions and implementations for operations that are loosely specified in  $Spec$ . A typical example is the representation of undefined terms (see [Far04] for a survey of techniques).

If  $\text{Spec}$  specifies the rational numbers using in particular  $\forall x.x \neq 0 \Rightarrow x/x = 1$ , and  $\mathcal{S}$  defines  $1/0 = 2/0 = 0$ , then  $\mathcal{S}$  is not sound because  $1/0 = 2/0$  is not a theorem of  $\text{Spec}$ .

We can define the above notions in MMT as follows. A *specification*  $\text{Spec}$  is an MMT theory; its meta-theory (if any) is called the *specification language*. A system implementing  $\text{Spec}$  consists of an MMT theory  $\mathcal{S}$  and an MMT theory morphism  $v : \text{Spec} \rightarrow \mathcal{S}$ ; the meta-theory of  $\mathcal{S}$  (if any) is called the *implementation language*. With this definition and using the Curry-Howard representation of MMT, we can provide a deductive system for the consequence relations used above:  $\Vdash_{\text{Spec}} F$  iff there is a  $p$  such that  $\vdash_{\text{Spec}} p : F$ ; and accordingly for  $\Vdash_{\mathcal{S}}$ .

In the simplest case, the morphism  $v$  is an inclusion, i.e., for every symbol in  $\text{Spec}$ ,  $\mathcal{S}$  contains a symbol of the same name. Using an arbitrary morphism  $v$  provides more flexibility, for example, the theory of the natural numbers with addition and multiplication implements the specification of monoids in two different ways via two different morphisms.

*Example 2.* We use a theory for second-order logic as the specification language; it declares symbols for  $\forall$ ,  $=$ , etc.  $\text{Spec} = \text{Nat}$  is a theory for the natural numbers; it declares symbols  $N$ ,  $0$  and  $\text{succ}$  as well as one symbol  $a : F$  for each Peano axiom  $F$ .

For the implementation language, we use a theory **ZF** for ZF set theory; it has meta-theory first-order logic and declares symbols for **set**,  $\in$ ,  $\emptyset$ , etc. Then we can implement the natural numbers in a theory  $\mathcal{S} = \text{Nat}$  declaring, e.g., a symbol **0** defined as  $\emptyset$ , a symbol **succ** defined such that  $\text{succ}(n) = n \cup \{n\}$ , and prove one theorem  $a : F = p$  in  $\mathcal{S}$  for each Peano axiom. Note that **Nat** yields theorems about the natural numbers that cannot be expressed in  $\text{Spec}$ , for example  $\Vdash_{\text{ZF}} 0 \in 1$ . We obtain a morphism  $\mu_1 : \text{Nat} \rightarrow \text{Nat}$  using  $N \mapsto N$ ,  $0 \mapsto \mathbf{0}$  etc.

Continuing Ex. 1, we obtain a different implementation  $\mu_2 : \text{Nat} \rightarrow \text{Nat}$  using  $N \mapsto \mathbb{N}$ ,  $0 \mapsto 0$  etc.

To capture practice in formal mathematics, we have to distinguish between the definitional and the axiomatic method. The *axiomatic* method fixes a formal system  $L$  and then describes mathematical notions in  $L$ -theories  $T$  using free symbols and axioms.  $T$  is interpreted in models, which may or may not exist. This is common in model theoretical logics, especially first-order logic, and in algebraic specification. In MMT,  $T$  is represented as a theory with meta-theory  $L$  and with only undefined constants. In Ex. 2,  $L$  is second-order logic and  $T$  is  $\text{Spec}$ .

The *definitional* method, on the other hand, fixes a formal system  $L$  together with a minimal theory  $T_0$  and then describes mathematical notions using definitional extensions  $T$  of  $T_0$ . The properties of the notions defined in  $T_0$  are derived as theorems. The interpretation of  $T$  is uniquely determined given a model of  $T_0$ . This is common in proof theoretical logics, especially LCF-style proof assistants, and in set theory. In Ex. 2,  $L$  is first-order logic,  $T_0$  is ZF, and  $T$  is  $\mathcal{S}$ .

*Types of Integration* Let us now consider a specification  $Spec$  and two implementations  $\mu_i : Spec \rightarrow \mathcal{S}_i$ . To simplify the notation, we will write  $\vdash$  and  $\vdash_i$  instead of  $\vdash_{Spec}$  and  $\vdash_{\mathcal{S}_i}$ . We first describe different ways how to integrate  $\mathcal{S}_1$  and  $\mathcal{S}_2$  intuitively.

*Borrowing* means to use  $\mathcal{S}_1$  to prove theorems in the language of  $\mathcal{S}_2$ . Thus, the input to  $\mathcal{S}_1$  is a conjecture  $F$  and the output is an expression  $\vdash_1 p : F$ . In general, since MMT does not prescribe a calculus for proofs, the object  $p$  can be a formal proof term, a certificate, proof sketch, or simply a yes/no answer.

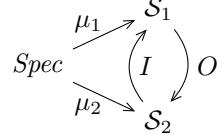
*Computation* means to reuse a  $\mathcal{S}_1$  computation in  $\mathcal{S}_2$ . Thus, the input of  $\mathcal{S}_1$  is an expression  $t$ , and the output is a proof  $p$  with an expression  $t'$  such that  $\vdash_1 p : t = t'$ . To be useful,  $t'$  should be simpler than  $t$  in some way, e.g., maximally simplified or even normalized.

*Querying* means answering a query in  $\mathcal{S}_1$  and transferring the results to  $\mathcal{S}_2$ . This is similar to borrowing in that the input to  $\mathcal{S}_1$  is a formula  $F$ . However, now  $F$  may contain free variables, and the output is not only a proof  $p$  but also a substitution  $s$  for the free variables such that  $\vdash_1 p : F[s]$ .

In all cases, a translation  $I$  must be employed to translate the input from  $\mathcal{S}_1$  to  $\mathcal{S}_2$ . Similarly, we need a translation  $O$  in the opposite direction to translate the output  $t'$  and  $s$  and (if available)  $p$  from  $\mathcal{S}_2$  to  $\mathcal{S}_1$ .

To define these integration types formally in MMT, we first note that borrowing is a special case of querying if  $F$  has no free variables. Similarly, computation is a special case of querying if  $F$  has the form  $t = X$  for a variable  $X$  that does not occur in  $t$ .

To define querying in MMT, we assume a specification, two implementations, and morphisms  $I$  and  $O$  as on the right.  $I$  and  $O$  must satisfy  $O \circ I = id_{\mathcal{S}_2}$ ,  $O \circ \mu_1 = \mu_2$ , and  $I \circ \mu_2 = \mu_1$ . Then we obtain the following *general form of an integration problem*: Given an  $\mathcal{S}_2$ -context  $C$  and a query  $C \vdash_2 ? : F$  (where  $?$  denotes the requested proof), find a substitution  $\vdash_1 s : I(C)$  and a proof  $\vdash_1 p : I(F)[s]$ . Then MMT guarantees that  $\vdash_2 O(p) : F[O(s)]$  so that we obtain  $O(s)$  as the solution. Moreover, only the existence of  $O$  is necessary but not  $O$  itself — once a proof  $p$  is found in  $\mathcal{S}_1$ , the existence of  $O$  ensures that  $F$  is true in  $\mathcal{S}_2$ , and it is not necessary to translate  $p$  to  $\mathcal{S}_2$ .



We call the above scenario *safe bidirectional communication* between  $\mathcal{S}_1$  and  $\mathcal{S}_2$  because  $I$  and  $O$  are theory morphisms and thus guarantee that consequence and truth are preserved in both directions. This scenario is often implicitly assumed by people coming from the first-order logic community. Indeed, if  $\mathcal{S}_1$  and  $\mathcal{S}_2$  are automatic or interactive theorem provers for first-order logic, then the logic of the two systems is the same and both  $\mathcal{S}_1$  and  $\mathcal{S}_2$  are equal to  $Spec$ .

If we are only interested in *safe directed communication*, i.e., transferring results from  $\mathcal{S}_1$  to  $\mathcal{S}_2$ , then it is sufficient to require only  $O$ . Indeed, often  $\mu_2$  is an inclusion, and the input parameters  $C$  and  $F$ , which are technically  $\mathcal{S}_2$ -objects, only use symbols from  $Spec$ . Thus, they can be moved directly to  $Spec$  and  $\mathcal{S}_1$ , and  $I$  is not needed.

Similarly, the substitution  $s$  can often be stated in terms of  $\text{Spec}$ . In that case,  $O$  is only needed to translate the proof  $p$ . If the proof translation is not feasible,  $O$  may be omitted as well. Then we speak of *unsafe communication* because we do not have a guarantee that the communication of results is correct. For example, let  $\mathcal{S}_1$  and  $\mathcal{S}_2$  be two CASs, that may compute wrong results by not checking side conditions during simplification. Giving a theory morphism  $O$  means that the “bugs” of the system  $\mathcal{S}_1$  must be “compatible” with the “bugs” of  $\mathcal{S}_2$ , which is quite unlikely.

The above framework for safe communication via theory morphisms is particularly appropriate for the integration of axiomatic systems. However, if  $\mathcal{S}_1$  and  $\mathcal{S}_2$  employ different mathematical foundations or different variants of the same foundation, it can be difficult to establish the necessary theory morphisms. In MMT, this means that  $\mathcal{S}_1$  and  $\mathcal{S}_2$  have different meta-theories so that  $I$  and  $O$  must include a meta-morphism. Therefore, unsafe communication is often used in practice, and even that can be difficult to implement.

Our framework is less appropriate if  $\mathcal{S}_1$  or  $\mathcal{S}_2$  are developed using the definitional method. For example, consider Aczel’s encoding of set theory in type theory [Acz99,Wer97]. Here  $\mathcal{S}_1 = \mathbf{Nat}$  as in Ex. 2, and  $\mathcal{S}_2 = \mathbf{Nat}$  as in Ex. 1. Aczel’s encoding provides the needed meta-morphism  $l : \mathbf{ZF} \rightarrow \text{CIC}$  of  $O$ . But because  $\mathbf{Nat}$  is definitional, we already have  $O = l$ , and we have no freedom to define  $O$  such that it maps the concepts of  $\mathbf{Nat}$  to their counterparts in  $\mathbf{Nat}$ . Formally, in MMT, this means that the condition  $O \circ \mu_1 = \mu_2$  fails. Instead, we obtain two versions of the natural numbers in CIC: a native one given by  $\mu_2$  and the translation of  $\mathbf{Nat}$  given by  $O \circ \mu_1$ . Indeed, the latter must satisfy all  $\mathbf{ZF}$ -theorems including, e.g.,  $0 \in 1$ , which is not even a well-formed formula over  $\mathbf{Nat}$ . We speak of *faithful communication* if  $O \circ \mu_1 = \mu_2$  can be established even when  $\mathcal{S}_1$  is definitional. This is not possible in MMT without the extension we propose below.

## 4 A Framework for System Integration

In order to realize faithful communication within MMT, we introduce *partial theory morphisms* that can filter out those definitional details of  $\mathcal{S}_1$  that need not and cannot be mapped to  $\mathcal{S}_2$ . We will develop this new concept in general in Sect. 4.1 and then apply it to the integration problem in Sect. 4.2.

### 4.1 Partial Theory Morphisms in MMT

*Syntax* We extend the MMT syntax with the production  $O ::= \top$ . The intended use of  $\top$  is to put assignments  $c \mapsto \top$  into the body of a morphism  $v : S \rightarrow T \stackrel{l}{=} \{\sigma\}$  in order to make  $v$  undefined at  $c$ . We say that  $v$  *filters*  $c$ . The homomorphic extension  $v(-)$  remains unchanged and is still total: If  $O$  contains filtered symbols, then  $v(O)$  contains  $\top$  as a subobject. In that case, we say  $v$  *filters*  $O$ .

*Semantics* We refine the semantics as follows. A *dependency cut*  $D$  for an MMT theory  $T$  is a pair  $(D_{type}, D_{def})$  of two sets of symbols accessible to  $T$ . Given such a dependency cut, we define *dependency-aware judgments*  $\gamma \vdash_D O : O'$  and  $\gamma \vdash_D O = O'$  as follows.  $\gamma \vdash_D O : O'$  means that there is a derivation of  $\gamma \vdash_T O : O'$  that uses the rules  $\mathcal{T}_!$  and  $\mathcal{T}_=$  at most for the constants in  $D_{type}$  and  $D_{def}$ , respectively.  $\gamma \vdash_D O = O'$  is defined accordingly.

In other words, if we have  $\gamma' \vdash_D O : O'$  and obtain  $\gamma'$  by changing the type of any constant not in  $D_{type}$  or the definiens of any constant not in  $D_{def}$ , then we still have  $\gamma' \vdash_D O : O'$ . Then a *foundation* consists of a foundational theory  $L$  together with dependency-aware judgments for typing and equality whenever  $T$  has meta-theory  $L$ .

We make a crucial change to the MMT rule for assignments in a theory morphism: If  $S$  contains a declaration  $c : O_1 = O_2$ , then a theory morphism  $v : S \xrightarrow{l} \{\sigma\}$  may contain the assignment  $c \mapsto O$  only if the following two conditions hold: (i) if  $O_1$  is not filtered by  $v$ , then  $\vdash_T O : v(O_1)$ ; (ii) if  $O_2$  is not filtered by  $v$ , then  $\vdash_T O = v(O_2)$ . The according rule applies if  $O_1$  or  $O_2$  are omitted.

In [RK10], a stricter condition is used. There, if  $O_1$  or  $O_2$  are filtered, then  $c$  must be filtered as well. While this is a natural strictness condition for filtering, it is inappropriate for our use cases: For example, filtering all  $L$ -symbols would entail filtering all  $S$ -symbols.

Our weakened strictness condition is still strong enough to prove the central property of theory morphisms: If  $\gamma \vdash \mu : S \rightarrow T$  and  $\vdash_D O : O'$  for some  $D = (D_{type}, D_{def})$  and  $v$  does not filter  $O, O'$ , the type of a constant in  $D_{type}$ , or the definiens of a constant in  $D_{def}$ , then  $\vdash_T \mu(O) : \mu(O')$ . The according result holds for the equality judgment.

Finally, we define the *weak equality of morphisms*  $\mu_i : S \rightarrow T$ . We define  $\vdash \mu_1 \leq \mu_2$  in the same way as  $\vdash \mu_1 = \mu_2$  except that  $\vdash_T \mu_1(c) = \mu_2(c)$  is only required if  $c$  is not filtered by  $\mu_1$ . We say that  $\vdash \eta : T \rightarrow S$  is a *partial inverse* of  $\mu : S \rightarrow T$  if  $\vdash \eta \circ \mu = id_S$  and  $\vdash \mu \circ \eta \leq id_T$ .

*Example 3.* Consider the morphism  $\mu_1 : Nat \rightarrow \mathbf{Nat}$  from Ex. 2. We build its partial inverse  $\eta : \mathbf{Nat} \rightarrow Nat \xrightarrow{l} \{\sigma\}$ . The meta-morphism  $l$  filters all symbols of  $\mathbf{ZF}$ , e.g.,  $l(\emptyset) = \top$ . Then the symbol  $\mathbf{N}$  of  $\mathbf{Nat}$  has filtered type and filtered definiens. Therefore, the conditions (i) and (ii) above are vacuous, and we use  $\mathbf{N} \mapsto N$  in  $\sigma$ . Then all remaining symbols of  $\mathbf{Nat}$  (including the theorems) have filtered definiens but unfiltered types. For example, for  $\mathbf{0} : \mathbf{N} = \emptyset$  we have  $\eta(\emptyset) = \top$  but  $\eta(\mathbf{N}) = N$ . Therefore, condition (ii) is vacuous, and we map these symbols to their counterparts in  $Nat$ , e.g., using  $\mathbf{0} \mapsto 0$  in  $\sigma$ . These assignments are type-preserving as required by condition (i) above, e.g.,  $\vdash_{Nat} \eta(\mathbf{0}) : \eta(\mathbf{N})$ .

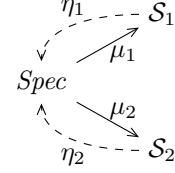
## 4.2 Integration via Partial Theory Morphisms

The following gives a typical application of our framework by safely and faithfully communicating proofs from a stronger system to a weaker system:

*Example 4.* In [IR11], we gave formalizations of Zermelo-Fraenkel ( $ZFC$ ) set theory and Mizar's Tarski-Grothendieck set theory ( $TG$ ) using the logical framework  $LF$  as the common meta-theory.  $ZFC$  and  $TG$  share the language of first-order set theory. But  $TG$  is stronger than  $ZFC$  because of Tarski's axiom, which implies, e.g., the sentence  $I$  stating the existence of infinite sets (which is an axiom in  $ZFC$ ) and large cardinals (which is unprovable in  $ZFC$ ). For example, we have an axiom  $a_\infty : I$  in  $ZFC$ , and an axiom  $tarski : T$  and a theorem  $t_\infty : I = P$  in  $TG$ . Many  $TG$ -theorems do not actually depend on this additional strength, but they do depend on  $t_\infty$  and thus indirectly on  $tarski$ .

Using our framework, we can capture such a theorem as the case of a  $TG$ -theorem  $\vdash_D p : F$  where  $F$  is the theorem statement and  $t_\infty \in D_{type}$  but  $t_\infty \notin D_{def}$  and  $tarski \notin D_{type}$ . We can give a partial theory morphism  $v : TG \rightarrow ZFC \stackrel{id_{LF}}{\equiv} \{ \dots, t_\infty \mapsto a_\infty, \dots \}$ . Then  $v$  does not filter  $p$ , and we obtain  $\vdash_{ZFC} v(p) : F$ .

Assume now that we have two implementations  $\mu_i : Spec \rightarrow \mathcal{S}_i$  of  $Spec$  and partial inverses  $\eta_i$  of  $\mu_i$ , where  $\mathcal{S}_i$  has meta-theory  $L_i$ . This leads to the diagram on the right where (dashed) edges are (partial) theory morphisms. We can now obtain the translations  $I : \mathcal{S}_2 \rightarrow \mathcal{S}_1$  and  $O : \mathcal{S}_1 \rightarrow \mathcal{S}_2$  as  $I = \mu_1 \circ \eta_2$  and  $O = \mu_2 \circ \eta_1$ . Note that  $I$  and  $O$  are partial inverses of each other.



As in Sect. 3 let  $C \vdash_2 ? : F$  be a query in  $\mathcal{S}_2$ . If  $\eta_2$  does not filter any symbols in  $C$  or  $F$ , we obtain the translated problem  $I(C) \vdash_1 ? : I(F)$ . Let us further assume that there is an  $\mathcal{S}_1$ -substitution  $\vdash_1 s : I(C)$  and a proof  $\vdash_1 p : I(F)[s]$  such that  $p$  and  $s$  are not filtered by  $\eta_1$ . Because  $I$  and  $O$  are mutually inverse and morphism application preserves typing, we obtain the solution  $\vdash_2 O(p) : F[O(s)]$ .

The condition that  $\eta_2$  does not filter  $C$  and  $F$  is quite reasonable in practice: Otherwise, the meaning of the query would depend on implementation-specific details of  $\mathcal{S}_2$ , and it is unlikely that  $\mathcal{S}_1$  should be able to find an answer anyway. On the other hand, the morphism  $\eta_1$  is more likely to filter the proof  $p$ . Moreover, since the proof must be translated from  $L_1$  to  $L_2$  passing through  $Spec$ , the latter must include a proof system to allow translation of proofs. In practice this is rarely the case, even if the consequence relation of  $Spec$  can be expressed as an inference system. For example, large parts of mathematics or the OpenMath content dictionaries implicitly (import) first-order logic and ZF set theory.

We outline two ways how to remedy this: We can *communicate filtered proofs* or change the morphisms to *widen the filters* to let more proofs pass.

*Communicating Filtered Proofs* Firstly, if the proof rules of  $\mathcal{S}_1$  are filtered by  $\eta_1$ , what is received by  $\mathcal{S}_2$  after applying the output translation  $O$  is a filtered proof, i.e., a proof object that contains the constant  $\top$ .  $\top$  represents gaps in the proof that were lost in the translation.

In an extreme case, all applications of proof rules become  $\top$ , and the only unfiltered parts of  $O(p)$  are formulas that occurred as intermediate results during the proof. In that case,  $O(p)$  is essentially a list of formulas  $F_i$  (a proof sketch

in the sense of [Wie03]) such that  $I(F_1) \wedge \dots \wedge I(F_{i-1}) \vdash_1 I(F_i)$  for  $i = 1, \dots, n$ . In order to refine  $O(p)$  into a proof, we have to derive  $\vdash_1 F_n$ . Most of the time, it will be the case that  $F_1, \dots, F_{i-1} \vdash_2 F_i$  for all  $i$ , and the proof is obtained compositionally if  $\mathcal{S}_2$  can fill the gaps through automated reasoning. When this happens, the proof sketch is already a complete declarative proof.

*Example 5.* Let  $\mathcal{S}_1$  and  $\mathcal{S}_2$  be implementations of the rational numbers with different choices for division by zero. In  $\mathcal{S}_1$ , division by zero yields a special value for undefined results, and operations on undefined values yield undefined results; then we have the  $\mathcal{S}_1$ -theorem  $t$  asserting  $\forall a, b, c. a(b/c) \doteq (ab)/c$ . In  $\mathcal{S}_2$ , we have  $n/0 \doteq 1$  and  $n\%0 \doteq n$ ; then we have the  $\mathcal{S}_2$ -theorems  $t_1, t_2, t_3$  asserting  $\forall m, n. n \doteq (n/m) * m + n\%m$ ,  $\forall m. m/m \doteq 1$ , and  $\forall m. m\%m \doteq 0$ .

The choice in  $\mathcal{S}_2$  reduces the number of case analyses in basic proofs. But  $t$  is not a theorem of  $\mathcal{S}_2$ ; instead, we only have a theorem  $t'$  asserting  $\forall a, b, c. c \neq 0 \Rightarrow a(b/c) \doteq (ab)/c$ . On the other hand,  $\mathcal{S}_1$  is closer to common mathematics, but the  $t_i$  are not theorems of  $\mathcal{S}_1$  because the side condition  $m \neq 0$  is needed.

Hence, we do not have a total theory morphism  $O : \mathcal{S}_1 \rightarrow \mathcal{S}_2$ , but we can give a partial theory morphism  $O$  that filters  $t$ . Now consider, for example, a proof  $p$  over  $\mathcal{S}_1$  that instantiates  $t$  with some values  $A, B, C$ . When translating  $p$  to  $\mathcal{S}_2$ ,  $t$  is filtered, but we can still communicate  $p$ , and  $\mathcal{S}_2$  can treat  $O(p)$  as a proof sketch. Typically,  $t$  is applied in a context where  $C \neq 0$  is known anyway so that  $\mathcal{S}_2$  can patch  $O(p)$  by using  $t'$  — which can easily be found by automated reasoning.

Integration in the other direction works accordingly.

*Widening the Filters* An alternative solution is to use additional knowledge about  $\mathcal{S}_1$  and  $\mathcal{S}_2$  to obtain a translation where  $O(p)$  is not filtered. In particular, if  $p$  is filtered completely, we can strengthen  $Spec$  by adding an inference system for the consequence relation of  $Spec$ , thus obtaining  $Spec'$ . Then we can extend the morphisms  $\mu_i$  accordingly to  $\mu'_i$ , which amounts to proving that  $\mathcal{S}_i$  is a correct implementation of  $Spec$ .

Now  $\eta_i$  can be extended as well so that its domain becomes bigger, i.e., the morphism  $\eta_1$  and thus  $O$  filter less proofs and become “wider”.

Note that we are flexible in defining  $Spec'$  as required by the particular choices of  $L_1$  and  $L_2$ . That way the official specification remains unchanged, and we can maximize the filters for every individual integration scenario.

*Example 6 (Continuing Ex. 3).* A typical situation is that we have a theorem  $F$  over **Nat** whose proof  $p$  uses the Peano axioms and the rules of first-order logic but does not expand the definitions of the natural numbers. Moreover, if  $a : A = P$  is a theorem in **Nat** that establishes one of the Peano axioms, then  $p$  will refer to  $a$ , but will not expand the definition of  $a$ . Formally, we can describe this as  $\vdash_D p : F$  where  $\mathbf{0}, a \in D_{type}$  but  $\mathbf{0}, a \notin D_{def}$ .

We can form  $Spec'$  by extending  $Spec$  with proof rules for first-order logic and extend  $\eta$  to  $\eta'$  accordingly. Since  $\eta$  does not filter the types of  $\mathbf{0}$  and  $a$ , we

$$\begin{array}{ccc} & \eta'_1 \dashrightarrow \mathcal{S}_1 & \\ Spec \hookrightarrow Spec' & \swarrow \quad \nearrow \mu'_1 & \\ & \eta'_2 \dashrightarrow \mathcal{S}_2 & \end{array}$$

obtain a proof  $\vdash_{\text{Spec}} \eta'(p) : \eta'(F)$  due to the type-preservation properties of our partial theory morphisms. Despite the partiality of  $\eta'$ , the correctness of this proof is guaranteed by the framework.

Both ways to integrate systems are not new and have been used ad hoc in concrete integration approaches, see Sect. 5. With our framework, we are able to capture them in a rigorous framework where their soundness can be studied formally.

## 5 Related Work

In the MoWGLI project [MoW], semantics markup is introduced to distinguish from specifications given in the content markup languages OpenMath and OM-Doc, and the implementations given in the calculus of constructions. This corresponds closely to the use of meta-theories in MMT: Their content markup corresponds to MMT theories without meta-theory; and semantics markup corresponds to MMT theories with meta-theory CIC.

A framework very similar to ours was given in [CFW03]. Our MMT theories with meta-theory correspond to their biform theories, except that the latter adds algorithms. Our theory morphisms  $I$  and  $O$  correspond to their translations `export` and `import`.

Integration by borrowing is the typical scenario of integrating theorem provers and proof assistants. For example, Leo-II [BPTF07] or the Sledgehammer tactic of Isabelle [Pau94] ( $\mathcal{S}_2$ ) use first-order provers ( $\mathcal{S}_1$ ) to reason in higher-order logic. Here the input translation  $I$  is partial inverse of the inclusion from first-order logic to higher-order logic. A total translation from modal logic to first-order logic is used in [HS00]. In all cases, the safety is verified informally on the meta-level and no output translation  $O$  in our sense is used. But Isabelle makes the communication safe by reconstructing a proof from the proof (sketch) returned by the prover.

The above systems are called on demand using an input translation  $I$ . Alternatively a collection of  $\mathcal{S}_1$ -proofs can be translated via an output translation  $O$  for later reuse in  $\mathcal{S}_2$ ; in that case no input translation  $I$  is used at all. Examples are the translations from Isabelle/HOL in HOL Light [McL06], from HOL Light to Isabelle/HOL [OS06], from HOL Light to Coq [KW10], or from Isabelle/HOL to Isabelle/ZF [KS10].

The translation from HOL to Isabelle/HOL is notable because it permits faithful translations, e.g., the real numbers of HOL can be translated to the real numbers of Isabelle/HOL, even though the two systems define them differently. The safety of the translation is achieved by recording individual  $\mathcal{S}_1$ -proofs and replaying them in  $\mathcal{S}_2$ . This was difficult to achieve even though  $\mathcal{S}_1$  and  $\mathcal{S}_2$  are based on the same logic.

The translation given in [KW10] is the first faithful translation from HOL proofs to CIC proofs. Since the two logics are different, in order to obtain a total map the authors widen the filter by assuming additional axioms on CIC (excluded middle and extensionality of functions). This technique is not exploitable

when the required axioms are inconsistent. Moreover, the translation is suboptimal, since it uses excluded middle also for proofs that are intuitionistic. To improve the solution, we could use partial theory morphisms that map case analysis over boolean in HOL to  $\top$ , and then use automation to avoid excluded middle in CIC when the properties involved are all decidable.

In all above examples but [KW10], the used translations are not verified within a logical framework. The Logosphere [PSK<sup>+</sup>03] project used the proof theoretical framework LF to provide statically verified logic translations that permit inherently safe communication. The most advanced such proof translation is one from HOL to Nuprl [NSM01].

The theory of institutions [GB92] provides a general model theoretical framework in which borrowing has been studied extensively [CM97] and implemented successfully [MML07]. Here the focus is on giving the morphism  $I$  explicitly and using a model theoretical argument to establish the existence of some  $O$ ; then communication is safe without explicitly translating proofs.

Integration by computation is the typical scenario for the integration of computer algebra systems, which is the main topic of the Calculemus series of conferences. For typical examples, see [DM05] where the computation is performed by a CAS, and [AT07] where the computation is done by a term rewriting system. Communication is typically unsafe. Alternatively, safety can be achieved if the results of the CAS — e.g., the factorization of a polynomial — can be verified formally in a DS as done in [HT98] and [Sor00].

Typical applications of integration by querying are conjunctive query answering for a description logic. For example, in [TSP08], a first-order theorem prover is used to answer queries about the SUMO ontology.

The communication of filtered proofs essentially leads to formal proof sketch in the sense of [Wie03]. The idea of abstracting from a proof to a proof sketch corresponds to the assertion level proofs used in [Mei00] to integrate first-order provers. The recording and replaying of proof steps in [OS06] and the reconstruction of proofs in Isabelle are also special cases of the communication of filtered proofs.

## 6 Conclusion

In this paper we addressed problems of preserving the semantics in protocol-based integration of mathematical reasoning and computation systems. We analyzed the problem from a foundational point of view and proposed a framework based on theory graphs, partial theory morphisms, and explicit representations of meta-logics that allows to state solutions to the integration problem.

The main contribution and novelty of the paper is that it paves the way towards a *theory of integration*. We believe that this theory of integration is practical because it requires only a simple extension of the MMT framework, which already takes scalability issues very seriously [KRZ10].

## References

- Acz99. Peter Aczel. On relating type theories and set theories. *Lecture Notes in Computer Science*, 1657, 1999.
- AT07. Andrea Asperti and Enrico Tassi. Higher order proof reconstruction from paramodulation-based refutations: The unit equality case. In *Calculemus/MKM*, pages 146–160, 2007.
- BC04. Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.
- BPTF07. C. Benzmüller, L. Paulson, F. Theiss, and A. Fietzke. The LEO-II Project. In *Automated Reasoning Workshop*, 2007.
- CF58. H. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.
- CFW03. J. Carette, W. Farmer, and J. Wajs. Trustable Communication between Mathematics Systems. In T. Hardin and R. Rioboo, editors, *Proceedings of Calculemus*, pages 58–68, 2003.
- CM97. M. Cerioli and J. Meseguer. May I Borrow Your Logic? (Transporting Logical Structures along Maps). *Theoretical Computer Science*, 173:311–347, 1997.
- DdC99. David Delahaye and Roberto di Cosmo. Information retrieval in a Coq proof library using type isomorphisms. In *Proceedings of TYPES 99*, volume Lecture Notes in Computer Science. Springer-Verlag, 1999.
- DM05. David Delahaye and Micaela Mayero. Dealing with Algebraic Expressions over a Field in Coq using Maple. *Journal of Symbolic Computation (JSC)*, 39(5):569–592, May 2005.
- Far04. W. Farmer. Formalizing Undefinedness Arising in Calculus. In *IJCAR*, volume 3097 of *LNCS*, pages 475–489, 2004.
- GB92. J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.
- How80. W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.
- HS00. U. Hustadt and R. Schmidt. MSPASS: Modal Reasoning by Translation and First-Order Resolution. In R. Dyckhoff, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference (TABLEAUX 2000)*, pages 67–71, 2000.
- HT98. J. Harrison and L. Théry. A Skeptic's Approach to Combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.
- IR11. M. Iancu and F. Rabe. Formalizing Foundations of Mathematics. *Mathematical Structures in Computer Science*, 2011. To appear, see [http://kwarc.info/frabe/Research/IR\\_foundations\\_10.pdf](http://kwarc.info/frabe/Research/IR_foundations_10.pdf).
- KRZ10. M. Kohlhase, F. Rabe, and V. Zholudev. Towards MKM in the Large: Modular Representation and Scalable Software Architecture. In S. Autexier, J. Calmet, D. Delahaye, P. Ion, L. Rideau, R. Rioboo, and A. Sexton, editors, *Intelligent Computer Mathematics*, volume 6167 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2010.
- KS10. A. Krauss and A. Schropp. A Mechanized Translation from Higher-Order Logic to Set Theory. In M. Kaufmann and L. Paulson, editors, *Proceedings of the Interactive Theorem Proving conference*, 2010. to appear in LNCS.

- KW10. C. Keller and B. Werner. Importing HOL Light into Coq. In M. Kaufmann and L. Paulson, editors, *Proceedings of the Interactive Theorem Proving conference*, 2010. to appear in LNCS.
- McL06. S. McLaughlin. An Interpretation of Isabelle/HOL in HOL Light. In N. Shankar and U. Furbach, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*. Springer, 2006.
- Mei00. A. Meier. System Description: TRAMP: Transformation of Machine-Found Proofs into ND-Proofs at the Assertion Level. In D. McAllester, editor, *Automated Deduction*, volume 1831 of *Lecture Notes in Computer Science*, pages 460–464. Springer, 2000.
- MML07. T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522, 2007.
- MoW. MoWGLI project deliverables.  
[http://mowgli.cs.unibo.it/html\\_no\\_frames/deliverables/index.html](http://mowgli.cs.unibo.it/html_no_frames/deliverables/index.html)
- NSM01. P. Naumov, M. Stehr, and J. Meseguer. The HOL/NuPRL proof translator - a practical approach to formal interoperability. In *14th International Conference on Theorem Proving in Higher Order Logics*. Springer, 2001.
- OS06. S. Obua and S. Skalberg. Importing HOL into Isabelle/HOL. In N. Shankar and U. Furbach, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*. Springer, 2006.
- Pau94. L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
- PSK<sup>+</sup>03. F. Pfenning, C. Schürmann, M. Kohlhase, N. Shankar, and S. Owre. The Logosphere Project, 2003. <http://www.logosphere.org/>.
- RK10. F. Rabe and M. Kohlhase. A Scalable Module System. To be submitted, see <http://kwarc.info/frabe/Research/mmt.pdf>, 2010.
- Sor00. V. Sorge. Non-trivial Symbolic Computations in Proof Planning. In *FroCoS*, pages 121–135. Springer, 2000.
- TSP08. S. Trac, G. Sutcliffe, and A. Pease. Integration of the TPTPWorld into SigmaKEE. In B. Konev, R. Schmidt, and S. Schulz, editors, *Practical Aspects of Automated Reasoning*, volume 373 of *CEUR Workshop Proceedings*, 2008.
- Wer97. Benjamin Werner. Sets in types, types in sets. In Martin Abadi and Taka-hashi Ito editors, editors, *Theoretical Aspect of Computer Software TACS'97, Lecture Notes in Computer Science*, volume 1281, pages 530–546. Springer-Verlag, 1997.
- Wie03. Freek Wiedijk. Formal proof sketches. In Wan Fokkink and Jaco van de Pol, editors, *7th Dutch Proof Tools Day, Program + Proceedings*, 2003. CWI, Amsterdam.

# Towards Logical Frameworks in the Heterogeneous Tool Set Hets

Mihai Codescu<sup>1</sup>, Fulya Horozal<sup>2</sup>, Michael Kohlhase<sup>2</sup>, Till Mossakowski<sup>1</sup>,  
Florian Rabe<sup>2</sup>, Kristina Sojakova<sup>3</sup>

DFKI GmbH, Bremen, Germany,  
Computer Science, Jacobs University, Bremen, Germany,  
Carnegie Mellon University, Pittsburgh, USA

**Abstract.** LF is a meta-logical framework that has become a standard tool for representing logics and studying their properties. Its focus is proof theoretic, employing the Curry-Howard isomorphism: propositions are represented as types, and proofs as terms.

Hets is an integration tool for logics, logic translations and provers, with a model theoretic focus, based on the meta-framework of institutions, a formalisation of the notion of logical system.

In this work, we combine these two worlds. The benefit for LF is that logics represented in LF can be (via Hets) easily connected to various interactive and automated theorem provers, model finders, model checkers, and conservativity checkers - thus providing much more efficient proof support than mere proof checking as is done by systems like Twelf. The benefit for Hets is that (via LF) logics become represented formally, and hence trustworthiness of the implementation of logics is increased, and correctness of logic translations can be mechanically verified. Moreover, since logics and logic translations are now represented declaratively, the effort of adding new logics or translations to Hets is greatly reduced.

This work is part of a larger effort of building an atlas of logics and translations used in computer science and mathematics.

## 1 Introduction

There is a large manifold of different logical systems used in computer science, such as propositional, first-order, higher-order, modal, description, temporal logics, and many more. These logical systems are supported by software, like (semi-)automated theorem provers, model checkers, computer algebra systems, constraint solvers, or concept classifiers, and each of these software systems comes with different foundational assumptions and input languages, which makes them non-interoperable and difficult to compare and evaluate in practice.

There are two main approaches to remedy this situation. The model theoretic approach of *institutions* [GB92] [Mes89] provides a formalisation of the notion of logical system. The benefit is that a large body of meta-theory can be developed independent of the specific logical system, including specification languages for structuring large logical theories. Recently, even a good part of

model theory has been generalised to this setting [Dia08]. Moreover, the Heterogeneous Tool Set (Hets, [MML07]) provides an institution-independent software interface, such that a heterogeneous proof management involving different tools (as listed above) is practically realised. In Hets, logic translations, formalized as so-called institution comorphisms, become first-class citizens. Heterogeneous specification and proof management is done relative to a graph of logics and translations.

The proof theoretic approach of logical frameworks starts with one “universal” logic that is used as a *logical framework*. This is used for representing logics as theories (in the “universal” logic of the framework). For instance, the Edinburgh Logical Framework LF [HHP93] has been used extensively to represent logics [HST94][PSK<sup>+</sup>03][AHMP98], many of them included in the Twelf distribution [PS99]. Logic representations in Isabelle [Pau94] are notable for the size of the libraries in the encoded logics, especially for HOL [NPW02]. Logic representations in rewriting logic [MOM96] using the Maude system [CELM96] include the examples of equational logic, Horn logic and linear logic. A notable property of rewriting logic is *reflection* i.e. one can represent rewriting logic within itself. Other systems employed to encode logics include Coq [BC04], Agda [Nor05], and Nuprl [CAB<sup>+</sup>86]. Only few logic *translations* have been formalized systematically in this setting. Important translations represented using the logic programming interpretation of LF include cut elimination [Pfe00] and the HOL-Nuprl translation [SS04]. The latter guided the design of the Delphin system [PS08] for logic translations.

Both approaches provide the theoretical and practical infrastructure to define logics. However, there are two major differences. Firstly, Hets is based on model theory – the semantics of implemented logics and the correctness of translations are determined by model theoretic arguments. Proof theory is only used as a tool to discharge proof obligations and is not represented explicitly.

Secondly, the logics of Hets are specified on the meta-level rather than within the system itself. Each logic or logic translation has to be specified by implementing a Haskell interface that is part of the Hets code, and tools for parsing and static analysis have to be provided. Consequently, only Hets developers but not users can add them. Besides the obvious disadvantage of the cost involved when adding logics, this representation does not provide us with a way to reason about the logics or their translations themselves. In particular, each logic’s static analysis is part of the trusted code base, and the translations cannot be automatically verified for correctness.

The present work unites and unifies these two approaches. We give a general definition of logical framework that covers systems such as LF, Isabelle, and Maude and implement it in Hets. We follow a “logics as theories/translations as morphisms” approach such that a theory graph in a logical framework leads to a graph of institutions and comorphisms via a general construction. This means that new logics can now be added to Hets in a purely declarative way. Moreover, the declarative nature means that logics themselves are no longer only formulated in the semi-formal language of mathematics, but now are fully

formal objects, such that one can reason about them (e.g. prove soundness of proof systems or logic translations) within proof systems like Twelf.

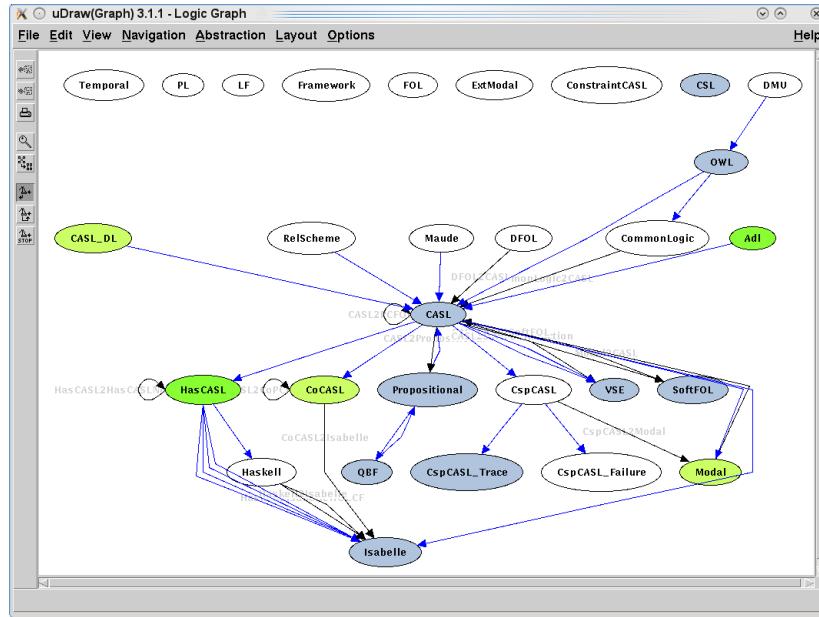
Our work is part of the ongoing project LATIN (Logic Atlas and Integrator, [\[KMR09\]](#)). One of the main goals of LATIN is to fully integrate proof and model theoretic frameworks. In the long run, we envision that these provers also return proof terms, which Hets can then fill into the original file and rerun Twelf on it to validate the proof. Thus, Hets becomes the mediator that orchestrates the interaction between external theorem provers and Twelf as a trusted proof checker.

This paper is organized as follows. We give introductions to the model and proof theoretic approaches and the LATIN logic graph in Sect. [2](#). In Sect. [3](#) we define the notion of a logical framework and describe its implementation in Hets in Sect. [4](#). We will use an encoding of first-order logic in the logical framework LF as a running example.

## 2 Preliminaries

### 2.1 The Heterogeneous Tool Set

The Heterogeneous Tool Set (Hets, [\[MML07\]](#)) is a set of tools for multi-logic specifications, which combines parsers, static analyzers, and theorem provers. Hets provides a heterogeneous specification language built on top of CASL [\[ABK<sup>+</sup>02\]](#) and uses the development graph calculus [\[MAH06\]](#) as a proof management component. The graph of logics supported by Hets and their translations is presented in Fig. [1](#).



**Fig. 1.** Hets logic graph

Hets formalizes the logics and their translations using the abstract model theory notions of institutions and institution comorphisms (see [GB92]).

**Definition 1.** An institution is a quadruple  $I = (\text{Sign}, \text{Sen}, \text{Mod}, \models)$  where:

- $\text{Sign}$  is a category of signatures;
- $\text{Sen} : \text{Sign} \rightarrow \text{Set}$  is a functor to the category  $\text{Set}$  of small sets and functions, giving for each signature  $\Sigma$  its set of sentences  $\text{Sen}(\Sigma)$  and for any signature morphism  $\varphi : \Sigma \rightarrow \Sigma'$  the sentence translation function  $\text{Sen}(\varphi) : \text{Sen}(\Sigma) \rightarrow \text{Sen}(\Sigma')$  (denoted by a slight abuse also  $\varphi$ );
- $\text{Mod} : \text{Sign}^{\text{op}} \rightarrow \text{Cat}$  is a functor to the category of categories and functors  $\text{Cat}$ <sup>1</sup> giving for any signature  $\Sigma$  its category of models  $\text{Mod}(\Sigma)$  and for any signature morphism  $\varphi : \Sigma \rightarrow \Sigma'$  the model reduct functor  $\text{Mod}(\varphi) : \text{Mod}(\Sigma') \rightarrow \text{Mod}(\Sigma)$  (denoted  $\_|_{\varphi}$ );
- a satisfaction relation  $\models_{\Sigma} \subseteq |\text{Mod}(\Sigma)| \times \text{Sen}(\Sigma)$  for each signature  $\Sigma$

such that the following satisfaction condition holds:

$$M'|_{\varphi} \models_{\Sigma'} e \Leftrightarrow M' \models_{\Sigma} \varphi(e)$$

for each  $M' \in |\text{Mod}(\Sigma')|$  and  $e \in \text{Sen}(\Sigma)$ , expressing that truth is invariant under change of notation and context.

For example, the institution of unsorted first-order logic  $\text{FOL}$  has signatures consisting of a set of function symbols and a set of predicate symbols, with their arities. Signature morphisms map symbols such that their arities are preserved. Models are first-order structures, and sentences are first-order formulas. Sentence translation function means replacement of the translated symbols. Model reduct means reassembling the model's components according to the signature morphism. Satisfaction is the usual satisfaction of a first-order sentence in a first-order structure.

**Definition 2.** Given two institutions  $I_1$  and  $I_2$  with  $I_i = (\text{Sig}_i, \text{Sen}_i, \text{Mod}_i, \models^i)$ , an institution comorphism from  $I_1$  to  $I_2$  consists of a functor  $\Phi : \text{Sig}_1 \rightarrow \text{Sig}_2$  and natural transformations  $\beta : \text{Mod}_2 \circ \Phi \Rightarrow \text{Mod}_1$  and  $\alpha : \text{Sen}_1 \Rightarrow \text{Sen}_2 \circ \Phi$ , such that the satisfaction condition

$$M' \models_{\Phi(\Sigma)}^2 \alpha_{\Sigma}(e) \iff \beta_{\Sigma}(M') \models_{\Sigma}^1 e,$$

where  $\Sigma$  is a  $I_1$  signature,  $e$  is a  $\Sigma$ -sentence in  $I_1$  and  $M'$  is a  $\Phi(\Sigma)$ -model in  $I_2$ .

The process of extending Hets with a new logic can be summarized as follows. First, we need to provide Haskell datatypes for the constituents of the logic, e.g. signatures, morphisms and sentences. This is done via instantiating various Haskell type classes, namely *Category* (for the signature category of the

---

<sup>1</sup> We disregard here the foundational issues, but notice however that  $\text{Cat}$  is actually a so-called quasi-category.

institution), *Sentences* (for the sentences), *Syntax* (for abstract syntax of basic specifications, and a parser transforming input text into this abstract syntax), *StaticAnalysis* (for the static analysis, turning basic specifications into theories, where a theory is a signature and a set of sentences). All this is assembled in the type class *Logic*, which additionally provides logic-specific tools like provers and model finders. For displaying the output of model finders, also (finite) models are represented in Hets, and these can even be translated against comorphisms. The model theoretic foundation of Hets also is visible by the fact that *StaticAnalysis* contains methods for checking amalgamability properties that are defined model theoretically (and therefore not available in purely proof theoretic logical frameworks). The type class *Logic* is used to represent logics in Hets internally. Finally, the new logic is made available by adding it to the list of Hets' known logics.

The input language of Hets is HetCASL. It combines logic-specific syntax of basic specifications (as specified by an instance of *Syntax*) with the logic-independent structuring constructs of CASL (like extension, union, translation of specifications, or hiding parts). Moreover, there are constructs for choosing a particular logic, as well as for translating a specification along an institution comorphism.

## 2.2 Proof Theoretic Logical Frameworks

We use the term *proof theoretic* to refer to logical frameworks whose semantics is or can be given in a formalist and thus mechanizable way without reference to a platonic universe. These frameworks are declarative formal languages with an inference system defining a consequence relation between judgments. They come with a notion of language extensions called signatures or theories, which admits the structure of a category. Logic encodings represent the syntax and proof theory of a logic as a theory of the logical framework, and logical consequence is represented in terms of the consequence relation of the framework.

The most important logical frameworks are LF, Isabelle, and Maude. LF [HHP93] is based on dependent type theory; logics are encoded as LF signatures, proofs as terms using the Curry-Howard correspondences, and consequence between formulas as type inhabitation. The main implementation is Twelf [PS99]. The Isabelle system [Pau94] implements higher-order logic [Chu40]; logics are represented as HOL theories, and consequence between formulas as HOL propositions. The Maude system [CELM96] is related to rewriting logic [MOM96]; logics are represented as rewrite theories, and consequence between formulas as rewrite judgments. Other languages such as Coq [BC04] or Agda [Nor05] can be used as logical frameworks as well, but this is not the primary application encountered in practice.

In the following, we give an overview over **LF**, which we will use as a running example. LF extends simple type theory with dependent function types and is related to Martin-Löf type theory [ML74]. The grammar gives a simplified

grammar for LF:

```

Signatures  $\Sigma ::= \cdot \mid \Sigma, c : E \mid \Sigma, c : E = E$ 
Morphisms  $\sigma ::= \cdot \mid \mid \sigma, c := E$ 
Expressions  $E ::= \text{type} \mid c \mid x \mid E E \mid \lambda_{x:E} E \mid \Pi_{x:E} E \mid E \rightarrow E$ 

```

LF **expressions**  $E$  are grouped into kinds  $K$ , kinded type-families  $A : K$ , and typed terms  $t : A$ . The kinds are the base kind **type** and the dependent function kinds  $\Pi_{x:A} K$ . The type families are the constants  $a$ , applications  $a t$ , and the dependent function type  $\Pi_{x:A} B$ ; type families of kind **type** are called types. The terms are constants  $c$ , applications  $t t'$ , and abstractions  $\lambda_{x:A} t$ . We write  $A \rightarrow B$  instead of  $\Pi_{x:A} B$  if  $x$  does not occur in  $B$ .

An LF **signature**  $\Sigma$  is a list of kinded type family declarations  $a : K$  and typed constant declarations  $c : A$ . Both may carry definitions, i.e.,  $c : A = t$  and  $a : K = A$ , respectively. Due to the Curry-Howard representation, propositions are encoded as types as well; hence a constant declaration  $c : A$  may be regarded as an axiom  $A$ , while  $c : A = t$  additionally provides a proof  $t$  for  $A$ . Hence, an LF signature corresponds to what usually is called a logical *theory*.

Relative to a signature  $\Sigma$ , closed expressions are related by the judgments  $\vdash_\Sigma E : E'$  and  $\vdash_\Sigma E = E'$ . Equality of terms, type families, and kinds are defined by  $\alpha\beta\eta$ -equality. All judgments for typing, kinding, and equality are decidable.

Given two signatures  $\Sigma$  and  $\Sigma'$ , an LF **signature morphism**  $\sigma : \Sigma \rightarrow \Sigma'$  is a typing- and kinding-preserving map of  $\Sigma$ -symbols to  $\Sigma'$ -expressions. Thus,  $\sigma$  maps every constant  $c : A$  of  $\Sigma$  to a term  $\sigma(c) : \bar{\sigma}(A)$  and every type family symbol  $a : K$  to a type family  $\sigma(a) : \bar{\sigma}(K)$ . Here,  $\bar{\sigma}$  is the homomorphic extension of  $\sigma$  to  $\Sigma$ -expressions, and we will write  $\sigma$  instead of  $\bar{\sigma}$  from now on.

Signature morphisms preserve typing, i.e., if  $\vdash_\Sigma E : E'$ , then  $\vdash_{\Sigma'} \sigma(E) : \sigma(E')$ , and correspondingly for kinding and equality. Due to the Curry-Howard encoding of axioms, this corresponds to the theorem preservation of theory morphisms. Composition and identity are defined in the obvious way, and we obtain a category  $\mathbb{LF}$ .

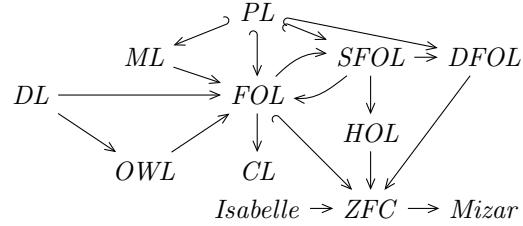
In [RS09], a **module system** was given for LF and implemented in Twelf. The module system permits to build both signatures and signature morphisms in a structured way. Its expressivity is similar to that of development graphs [AHMS99].

### 2.3 A Logic Atlas in LF

In the LATIN project [KMR09], we aim at the creation of a logic atlas based on LF. The Logic Atlas is a multi-graph of LF signatures and morphisms between them. Currently it contains formalizations of various logics, type theories, foundations of mathematics, algebra, and category theory.

Among the logics formalized in the Atlas are propositional (*PL*), first (*FOL*) and higher-order logic (*HOL*), sorted (*SFOL*) and dependent first-order logic (*DFOL*), description logics (*DL*), modal (*ML*) and common logic (*CL*) as illustrated in the diagram below. Single arrows ( $\rightarrow$ ) in this diagram denote translations between formalizations and hooked arrows ( $\hookrightarrow$ ) denote imports. Among the

foundations are encodings of Zermelo-Fraenkel set theory, Isabelle’s higher-order logic, and Mizar’s Set theory [IR10].



Actually the graph is significantly more complex as we use the LF module system to obtain a maximally modular design of logics. For example, first-order, modal, and description logics are formed from orthogonal modules for the individual connectives, quantifiers, and axioms. For example, the  $\wedge$  connective is only declared once in the whole Atlas and imported into the various logics and foundations and related to type theoretic product via the Curry-Howard correspondence.

Moreover, we use individual modules for syntax, proof theory, model theory so that the same syntax can be combined with different interpretations. For example, the formalization of first-order logic ([HR10]) consists of the signatures *Base* and  $FOL^{Syn}$  for syntax,  $FOL^{Pf}$  for proof theory, and  $FOL^{Mod}$  for model theory as illustrated in the diagram on the right. *Base* contains declarations  $o : \text{type}$  and  $i : \text{type}$  for the type of formulas and first-order individuals, and a truth judgment for formulas.  $FOL^{Syn}$  contains declarations for all logical connectives and quantifiers (see Fig. 4).  $FOL^{truth}$  is an inclusion morphism from *Base* to  $FOL^{Syn}$ .  $FOL^{Pf}$  consists of declarations for judgments and inference rules associated with each logical symbol declared in  $FOL^{Syn}$ .  $FOL^{pf}$  is an inclusion morphism from  $FOL^{Syn}$  to  $FOL^{Pf}$ .  $FOL^{Mod}$  contains declarations that axiomatize the properties of FOL-models. In particular, it contains a declaration of a set *univ* for the universe. It also includes a formalization *ZFC* of the Zermelo-Fraenkel set theory in which the models are defined. The morphism  $FOL^{mod}$  interprets  $FOL^{Syn}$  in  $FOL^{Mod}$ .

$$\begin{array}{c}
FOL^{Pf} \\
FOL^{pf} \nearrow \\
Base \xrightarrow{FOL^{truth}} FOL^{Syn} \\
\downarrow FOL^{mod} \\
FOL^{Mod} \\
M \Downarrow \\
ZFC
\end{array}$$

Then individual FOL-models are represented as LF signature morphisms from  $FOL^{Mod}$  to *ZFC* that are the identity on *ZFC*. Given such a morphism  $M$ , the composition  $FOL^{mod}; M$  yields the interpretation of  $FOL^{Syn}$  in *ZFC*. This yields a representation of models as LF signature morphisms.

### 3 Logical Frameworks

#### 3.1 Main Definition

Following the approach taken in [Rab10], we use logical frameworks that are based on a formal language given by a category of theories. We deliberately restrict attention to a special case that makes the ideas clearest and discuss generalizations in Sect. 3.2.

**Definition 3 (Inclusions).** A category has *inclusions* if it has a broad subcategory that is a partial order. We write  $B \hookrightarrow C$  for the inclusion morphism from  $B$  to  $C$ , and given  $A \xrightarrow{f} B \hookrightarrow C \xrightarrow{g} D$ , we abbreviate  $f|_C^C = (B \hookrightarrow C) \circ f$  and  $g|_B = g \circ (B \hookrightarrow C)$ .

**Definition 4 (Logical Framework).** A tuple  $(\mathbb{C}, \text{Base}, \mathbf{Sen}, \vdash)$  is a logical framework if

- $\mathbb{C}$  is a category that has inclusions and pushouts along inclusions,
- $\text{Base}$  is an object of  $\mathbb{C}$ ,
- $\mathbf{Sen}$  is a functor  $\mathbb{C} \setminus \text{Base} \rightarrow \mathcal{SET}$ ,
- for  $t : \text{Base} \rightarrow \Sigma$ ,  $\vdash_t$  is a unary predicate on  $\mathbf{Sen}(t)$ ,
- $\vdash$  is preserved under signature morphisms: if  $\vdash_t F$  then  $\vdash_{t'} \mathbf{Sen}(\sigma)(F)$  for any morphism  $\sigma : t \rightarrow t'$  in  $\mathbb{C} \setminus \text{Base}$ .

$\mathbb{C}$  is the category of theories of the logical framework. Our focus is on declarative frameworks where theories are lists of named declarations. Typically these have inclusions and pushouts along them in a natural way.

Logics are encoded as theories  $\Sigma$  of the framework, but not all theories can be naturally regarded as logic encodings. Logic encodings must additionally distinguish certain objects over  $\Sigma$  that encode logical notions. Therefore, we consider  $\mathbb{C}$ -morphisms  $t : \text{Base} \rightarrow \Sigma$  where  $\text{Base}$  makes precise what objects must be distinguished.

We leave the structure of  $\text{Base}$  abstract, but we require that slices  $t : \text{Base} \rightarrow \Sigma$  provide at least a notion of sentences and truth for the logic encoded by  $\Sigma$ . Therefore,  $\mathbf{Sen}(t)$  gives the set of sentences, and the predicate  $\vdash_t F$  expresses the truth of  $F$ .

*Example 1 (LF).* We define a logical framework  $\mathbb{LF}$  based on the category  $\mathbb{C} = \mathbb{LF}$ .  $\mathbb{LF}$  has inclusions by taking the subset relation between sets of declarations. Given  $\sigma : \Sigma \rightarrow \Sigma'$  and an inclusion  $\Sigma \hookrightarrow \Sigma$ , a pushout is given by

$$(\sigma, c := c) : (\Sigma, c : A) \rightarrow (\Sigma', c : \sigma(A))$$

(except for possibly renaming  $c$  if it is not fresh for  $\Sigma'$ ). The pushouts for other inclusions are obtained accordingly.

$\text{Base}$  is the signature with the declarations  $o : \text{type}$  and  $\text{ded} : o \rightarrow \text{type}$ . For every slice  $t : \text{Base} \rightarrow \Sigma$ , we define  $\mathbf{Sen}(t)$  as the set of closed  $\beta\eta$ -normal

LF-terms of type  $t(o)$  over the signature  $\Sigma$ . Moreover,  $\vdash_t F$  holds iff the  $\Sigma$ -type  $t(\text{ded}) F$  is inhabited.

Given  $t : \text{Base} \rightarrow \Sigma$  and  $t' : \text{Base} \rightarrow \Sigma'$  and  $\sigma : \Sigma \rightarrow \Sigma'$  such that  $\sigma \circ t = t'$ , we define the sentence translation by  $\mathbf{Sen}(\sigma)(F) = \sigma(F)$ . Truth is preserved: assume  $\vdash_t F$ ; thus  $t(\text{ded}) F$  is inhabited over  $\Sigma$ ; then  $\sigma(t(\text{ded}) F) = t'(\text{ded}) \sigma(F)$  is inhabited over  $\Sigma'$ ; thus  $\vdash_{t'} \mathbf{Sen}(\sigma)(F)$ .

*Example 2 (Isabelle).* A logical framework based on Isabelle is defined similarly.  $\mathbb{C}$  is the category of Isabelle theories and theory morphisms.  $\text{Base}$  consists of the declarations  $\text{bool} : \text{type}$  and  $\text{trueprop} : \text{bool} \rightarrow \text{prop}$  where  $\text{prop}$  is the type of Isabelle propositions. Given  $t : \text{Base} \rightarrow \Sigma$ , we define  $\mathbf{Sen}(t)$  as the set of  $\Sigma$ -terms of type  $t(\text{bool})$ , and  $\vdash_t F$  holds if  $t(\text{trueprop}) F$  is an Isabelle theorem over  $\Sigma$ .

We use logical frameworks to define institutions. The basic idea is that slices  $t : \text{Base} \rightarrow L^{\text{Syn}}$  define logics ( $L^{\text{Syn}}$  specifies the syntax of the logic), signatures of that logic are extensions  $L^{\text{Syn}} \hookrightarrow \Sigma^{\text{Syn}}$ , and sentences and truth are given by  $\mathbf{Sen}$  and  $\vdash$ . We could represent the logic's models in terms of the models of the logical framework, but that would complicate the mechanizable representation of models. Therefore, we represent models as  $\mathbb{C}$  morphisms into a fixed theory that represents the foundation of mathematics. We need one auxiliary definition to state this precisely:

**Definition 5.** Fix a logical framework, and assume  $L^{\text{mod}} : L^{\text{Syn}} \rightarrow L^{\text{Mod}}$  in  $\mathbb{C}$  as in the diagram below.

$$\begin{array}{ccccc}
& & \Sigma^{\text{Mod}} & & \Sigma'^{\text{Mod}} \\
L^{\text{Mod}} \curvearrowleft & \xrightarrow{\quad} & \Sigma^{\text{Mod}} \xrightarrow{\sigma^{\text{mod}}} & \Sigma'^{\text{Mod}} \\
\uparrow L^{\text{mod}} & & \uparrow \Sigma^{\text{mod}} & & \uparrow \Sigma'^{\text{mod}} \\
L^{\text{Syn}} \curvearrowleft & \xrightarrow{\quad} & \Sigma^{\text{Syn}} \xrightarrow{\sigma^{\text{syn}}} & \Sigma'^{\text{Syn}} \\
& & \curvearrowright & &
\end{array}$$

Firstly, for every inclusion  $L^{\text{Syn}} \hookrightarrow \Sigma^{\text{Syn}}$ , we define  $\Sigma^{\text{Mod}}$  and  $\Sigma^{\text{mod}}$  such that  $\Sigma^{\text{Mod}}$  is a pushout. Secondly, for every  $\sigma^{\text{syn}} : \Sigma^{\text{Syn}} \rightarrow \Sigma'^{\text{Syn}}$ , we define  $\sigma^{\text{mod}} : \Sigma^{\text{Mod}} \rightarrow \Sigma'^{\text{Mod}}$  as the unique morphism such that the above diagram commutes.

Then we are ready for our main definition:

**Definition 6.** Fix a logical framework  $\mathbb{F} = (\mathbb{C}, \text{Base}, \mathbf{Sen}, \vdash)$ . Assume  $L = (L^{\text{Syn}}, L^{\text{truth}}, L^{\text{Mod}}, \mathcal{F}, L^{\text{mod}})$  as in the following diagram:

$$\begin{array}{ccccc}
& & \mathcal{F} & & \\
& & \xrightarrow{id_{\mathcal{F}}} & & \\
& \downarrow & m \uparrow & \swarrow m' & \\
L^{Mod} & \longleftrightarrow & \Sigma^{Mod} & \xrightarrow{\sigma^{mod}} & \Sigma'^{Mod} \\
\uparrow L^{mod} & & \uparrow \Sigma^{mod} & & \uparrow \Sigma'^{mod} \\
Base & \xrightarrow{L^{truth}} & L^{Syn} & \xrightarrow{\sigma^{syn}} & \Sigma'^{Syn}
\end{array}$$

Then we define the institution  $\mathbb{F}(L) = (\mathbf{Sig}^L, \mathbf{Sen}^L, \mathbf{Mod}^L, \models^L)$  as follows:

- $\mathbf{Sig}^L$  is the full subcategory of  $\mathbb{C} \setminus L^{Syn}$  whose objects are inclusions. To simplify the notation, we will write  $\Sigma^{Syn}$  for an inclusion  $L^{Syn} \hookrightarrow \Sigma^{Syn}$  below.
- $\mathbf{Sen}^L$  is defined by

$$\mathbf{Sen}^L(\Sigma^{Syn}) = \mathbf{Sen}(L^{truth}|^{\Sigma^{Syn}}) \quad \mathbf{Sen}^L(\sigma) = \mathbf{Sen}(\sigma).$$

- $\mathbf{Mod}^L$  is defined by

$$\mathbf{Mod}^L(\Sigma^{Syn}) = \{m : \Sigma^{Mod} \rightarrow \mathcal{F} \mid m|_{\mathcal{F}} = id_{\mathcal{F}}\} \quad \mathbf{Mod}^L(\sigma^{syn})(m') = m' \circ \sigma^{mod}$$

All model categories are discrete.

- We make the following abbreviation: For a model  $m \in \mathbf{Mod}^L(\Sigma^{Syn})$ , we write  $\overline{m}$  for  $m \circ \Sigma^{mod} \circ L^{truth}|^{\Sigma^{Syn}} : Base \rightarrow \mathcal{F}$ . Then we define satisfaction by

$$m \models_{\Sigma^{Syn}}^L F \quad \text{iff} \quad \vdash_{\overline{m}} \mathbf{Sen}(m \circ \Sigma^{mod})(F).$$

**Theorem 1.** In the situation of Def. [6],  $\mathbb{F}(L)$  is an institution.

*Proof.* We need to show the satisfaction condition. So assume  $\sigma : \Sigma^{Syn} \rightarrow \Sigma'^{Syn}$ ,  $F \in \mathbf{Sen}^L(\Sigma^{Syn})$ , and  $m' \in \mathbf{Mod}^L(\Sigma'^{Syn})$ . First observe that  $\overline{m'} = m' \circ \Sigma'^{mod} \circ L^{truth}|^{\Sigma'^{Syn}} = (m' \circ \sigma^{mod}) \circ \Sigma^{mod} \circ L^{truth}|^{\Sigma^{Syn}} = \overline{m' \circ \sigma^{mod}}$ . Then  $\mathbf{Mod}^L(\sigma)(m') \models_{\Sigma^{Syn}}^L F$  iff  $\vdash_{\overline{m' \circ \sigma^{mod}}} \mathbf{Sen}((m' \circ \sigma^{mod}) \circ \Sigma^{mod})(F)$  iff  $\vdash_{\overline{m'}} \mathbf{Sen}(m' \circ \Sigma'^{mod})(\mathbf{Sen}(\sigma^{syn})(F))$  iff  $m' \models_{\Sigma'^{Syn}}^L \mathbf{Sen}^L(\sigma^{syn})(F)$ .

*Example 3 (FOL).* We can now obtain an institution from the encoding of first-order logic in Sect. [2.3] based on the logical framework  $\mathbb{F}^{LF}$ . First-order logic is encoded as the tuple  $FOL = (FOL^{Syn}, FOL^{truth}, FOL^{Mod}, ZFC, FOL^{mod})$ .  $FOL^{Syn}$ ,  $FOL^{Mod}$ ,  $ZFC$  and  $FOL^{mod}$  are as in Sect. [2.3].  $FOL^{truth}$  is the inclusion from  $Base$  to  $FOL^{Syn}$ .

We obtain an institution comorphism  $\mathbb{FOL} \rightarrow \mathbb{F}^{LF}(FOL)$  as follows. Signatures of  $\mathbb{FOL}$  are mapped to the extension of  $FOL^{Syn}$  with declarations  $f : i \rightarrow \dots \rightarrow i \rightarrow i$  for function symbols  $f$ ,  $p : i \rightarrow \dots \rightarrow i \rightarrow o$  for predicate symbols  $p$ . If we want to map  $\mathbb{FOL}$  theories as well, we add declarations  $ax : \text{ded } F$  for every axiom  $F$ . Signature morphisms are mapped in the obvious way. The sentence translation is an obvious bijection. The model translation maps every  $m : \Sigma^{Mod} \rightarrow \mathcal{F}$  to the model whose universe is given by  $m(univ)$ , which interprets symbols  $f$  and  $p$  according to  $m(f)$  and  $m(p)$ .

Logical frameworks can also be used to encode institution comorphisms in an intuitive way:

**Theorem 2.** Fix a logical framework  $\mathbb{F} = (\mathcal{C}, \text{Base}, \mathbf{Sen}, \vdash)$ . Assume two logics  $L = (L^{\text{Syn}}, L^{\text{truth}}, L^{\text{Mod}}, \mathcal{F}, L^{\text{mod}})$  and  $L' = (L'^{\text{Syn}}, L'^{\text{truth}}, L'^{\text{Mod}}, \mathcal{F}, L'^{\text{mod}})$ . Then a comorphism  $\mathbb{F}(L) \rightarrow \mathbb{F}(L')$  is induced by morphisms  $(l^{\text{syn}}, l^{\text{mod}})$  if the following diagram commutes

$$\begin{array}{ccccc}
 & & \mathcal{F} & & \\
 & \swarrow & & \searrow & \\
 L^{\text{Mod}} & \xrightarrow{l^{\text{mod}}} & L'^{\text{Mod}} & & \\
 \uparrow L^{\text{mod}} & \nearrow L^{\text{truth}} & \text{Base} & \searrow L'^{\text{truth}} & \uparrow L'^{\text{mod}} \\
 L^{\text{Syn}} & \xrightarrow{l^{\text{syn}}} & L'^{\text{Syn}} & &
\end{array}$$

*Proof.* A signature  $L^{\text{Syn}} \hookrightarrow \Sigma^{\text{Syn}}$  is translated to  $L'^{\text{Syn}} \hookrightarrow \Sigma'^{\text{Syn}}$  by pushout along  $l^{\text{syn}}$  yielding  $\sigma^{\text{syn}} : \Sigma^{\text{Syn}} \rightarrow \Sigma'^{\text{Syn}}$ . Sentences are translated by applying  $\sigma^{\text{syn}}$ . We obtain  $\sigma^{\text{mod}} : \Sigma^{\text{Mod}} \rightarrow \Sigma'^{\text{Mod}}$  as the unique morphism through the pushout  $\Sigma^{\text{Mod}}$ . Then models are translated by composition with  $\sigma^{\text{mod}}$ . We omit the details.

It is easy to see that comorphisms that are embeddings can be elegantly represented in this way, as well as many inductively defined encodings. However, the assumptions of this theorem are too strong to permit the encoding of some less trivial comorphisms. For example, non-compositional sentence translations, which come up when translating modal logic to first-order logic, cannot be represented as signature morphisms. Or signature translations that do not preserve the number of non-logical symbols, which come up when translating partial to total function symbols, often cannot be represented as pushouts. More general constructions for the special case of LF are given in [Rab10] and [Soj10].

### 3.2 Generalizations

In Ex. 3, we do not obtain a comorphism in the opposite direction. There are three reasons for that. Firstly,  $\mathbb{F}^{\text{LF}}(\text{FOL})$  contains a lot more signatures than needed because the definition of  $\mathbf{Sig}^L$  permits any extension of  $L^{\text{Syn}}$ , not just the ones corresponding to function and predicate symbols. Secondly, the discrete model categories of  $\mathbb{F}^{\text{LF}}(\text{FOL})$  cannot represent the model morphisms of  $\text{FOL}$ . Thirdly, only a (countable) subclass of the models of  $\text{FOL}$  can be represented as  $\text{LF}$  morphisms. Moreover, Def. 4 and 6 are restricted to institutions, i.e., the syntax and model theory of a logic, and exclude the proof theory. We look at these problems below.

*Signatures* In order to solve the first problem we need to restrict  $\mathbb{F}(L)$  to a subcategory of  $\mathbf{Sig}^L$ . However, it is difficult to single out the needed subcategory in a mechanizable way. Therefore, we restrict attention to those logical frameworks where  $\mathbb{C}$  is the category of theories of a declarative language.

In a declarative language, the theories are given by a list of typed symbol declarations. In order to formalize this definition without committing to a type system, we use MMT expressions ([Rab08](#)) as the types. MMT expressions are formed from variables, constants, applications  $@(E, l)$  of an expression  $E$  to a list of expressions  $l$ , bindings  $\beta(E, l, E')$  of a binder  $E$  with scope  $E'$  binding a list of variables types by the elements of  $l$ . To that we add jokers  $*$ , which matches an arbitrary expression, and  $\overline{E}$ , which matches a list of expressions each of which matches  $E$ .

Such MMT expression patterns give us a generic way to pattern-match declarations of the logical framework. If a concrete logic definition contains a set  $P$  of patterns, we represent its logical signatures as  $\mathbb{C}$ -objects  $\Sigma^{Syn}$  that extend  $L^{Syn}$  only with declarations matching one of the patterns in  $P$ . For example, the patterns for first-order logic from Ex. [3](#) would be  $@(\rightarrow, \bar{i}, i)$  and  $@(\rightarrow, \bar{i}, o)$  for function and predicate symbols of arbitrary arity, and  $@(\text{ded}, *)$  for axioms.  $\bar{i}$  stands for a list of  $i$  representing an arbitrary number of arguments;  $*$  stands for an arbitrary expression, which in this case must be a sentence to be well-typed.

*Model Morphisms* Regarding the second problem, if  $\mathbb{C}$  is a 2-category, we can define the model morphisms of  $\mathbb{F}(L)$  as 2-cells in  $\mathbb{C}$ . However it is difficult in practice to obtain 2-categories for type theories such as LF or Isabelle. In [Soj10](#), we give a syntactical account of logical relations that behave like 2-cells in sufficiently many ways to yield model morphisms.

*Undefinable Models* The third problem is the most fundamental one because no formalist logical framework can ever encode all models of a Platonic universe. Our encoding of ZFC is strong enough to encode any **definable** model. We call a model  $M$  definable if it arises as the solution to a formula  $\exists^1 M.F(M)$  for some parameter-free formula  $F(x)$  of the first-order language of ZFC. This restriction is philosophically serious but in our experience not harmful in practice. Indeed, if infinite LF signatures are allowed, using canonical models constructed in completeness proofs, in many cases *all* models can be represented up to elementary equivalence.

*Proof Theory* Our examples from Sect. [2.3](#) already encoded the proof theory of first-order logic in a way that treats proof theory and model theory in a balanced way. Our definitions can be easily generalized to this setting.

Logic encodings in a logical framework become 6-tuples  $(L^{Syn}, L^{truth}, L^{Mod}, \mathcal{F}, L^{mod}, L^{Pf}, L^{pf})$  for  $L^{Pf} : L^{Syn} \rightarrow L^{Pf}$ .  $L^{Pf}$  encodes the proof theory of a logic, which typically means to add auxiliary syntax, judgments, and proof rules to  $L^{Syn}$ . Def. [5](#) can be extended to obtain  $\Sigma^{pf} : \Sigma^{Syn} \rightarrow \Sigma^{Pf}$  as a pushout in the same way as  $\Sigma^{mod}$ . Finally the logical framework must be extended with a

component that yields a data structure of proofs (such as entailment systems or proof trees) for every slice out of *Base*.

For example, for the framework  $\text{F}^{\text{LF}}$ , the proof trees for proofs of  $F$  using assumptions  $F_1, \dots, F_n$  can be defined as the  $\beta\eta$ -normal LF terms over  $\Sigma^{\text{Pf}}$  of type  $\Sigma^{\text{Pf}}(L^{\text{truth}}(\text{ded})F_1 \rightarrow \dots \rightarrow L^{\text{truth}}(\text{ded})F_n \rightarrow L^{\text{truth}}(\text{ded})F)$ . A similar construction was given in [Rab10].

## 4 Logical Frameworks in Hets

The differences between LF and Hets mentioned in Sect. 2 exhibit complementary strengths, and a major goal of our work is to combine them. We have enhanced Hets with a component that allows the dynamic definition of new logics. The user specifies a logic by giving the representation of its constituents (syntax, model theory) in a logical framework and the combined system recognizes the new logic and integrates it into the Hets logic graph. The implementation follows the Hets principles of high abstraction and separation on concerns: we provide an implementation for the general concept of logical frameworks, which we describe in Sect. 4.1. This is further instantiated for the particular case of LF in Sect. 4.2. Finally, in Sect. 4.3 we present a complete description of the steps necessary to add a new logic in Hets using the framework of LF.

### 4.1 Implementing Logical Frameworks in Hets

To be able to specify a new object logic  $L$  in Hets, we start by declaring a Haskell type class *LogicalFramework*, which is instantiated by the logics which can be used as logical frameworks i.e. in which object logics can be specified by the user. Such candidates are for example LF, Maude and Isabelle. The class provides a selector for the *Base* signature and a method *writeLogic*, which generates the instances of the classes *Syntax*, *Sentences*, *StaticAnalysis*, and *Logic* for the object logic  $L$ .

Each logic implementing *LogicalFramework* must likewise implement the class *Category*, from which we get the category  $\mathbb{C}$  mentioned in Def. 4. The sentence functor **Sen** is specified implicitly by the *writeLogic* method: the instantiation of the *StaticAnalysis* class determines exactly which sentences are valid for a particular signature of  $L$ , thus giving **Sen** on objects. Since the current implementation of logics in Hets does not include satisfaction of sentences in models, the predicate  $\vdash_t$  is currently not represented as its main purpose is to define the satisfaction relation for object logics.

At the syntactic level, we must provide a way to write down new logic definitions in HetCASL, the underlying heterogenous algebraic specification language of Hets. Since definitions of new logics have a different status than usual algebraic specifications, we extend the language at the library level.

*Concrete Syntax* We add the following concrete syntax (on the right) to HetCASL in order to define new logics. Here  $L$  is the name of the newly defined logic and  $\mathbb{F}^{LF}$  is an identifier pointing to the logical framework used. The identifiers  $L^{truth}, L^{mod}, L^{pf}$  are the components of the new logic  $L$ . They refer to previously declared signature morphisms of  $\mathbb{F}^{LF}$  and the signatures representing  $L^{Syn}, L^{Mod}, L^{Pf}$  can be inferred from them. The declaration of patterns is optional.

After encountering a `newlogic` declaration, Hets invokes a static analyzer, which retrieves the signatures and morphisms constituting the components of the logic  $L$ . The analyzer verifies the correct shape of the induced diagram and instantiates the *Logic* class for the logic  $L$  as specified by the *writeLogic* method of the framework used to encode  $L$ .

The logic  $L$  arising from the `newlogic L` declaration differs from the one described in Def. 6 in that it does not use slice categories - the signatures of  $L$  are those signatures of  $\mathbb{F}$  which extend  $L^{Syn}$  and the morphisms of  $L$  are those morphisms of  $\mathbb{F}$  which are identity on  $L^{Syn}$ . This has the advantage that the data types representing the signatures and morphisms of  $\mathbb{F}$  can be directly reused for  $L$  and no separate instantiation of the class *Category* is required.

## 4.2 LF as a Logical Framework in Hets

To instantiate the *LogicalFramework* class for LF, we will make use of the instance of *Logic* class for  $\mathbb{LF}$ <sup>2</sup>

The aim here is to reuse Twelf for parsing and static analysis: Any applicable input file - i.e. a Twelf file or a HetCASL file which has LF as the designated logic - is forwarded to Twelf, where parsing, static analysis and reconstruction of types and implicit arguments are performed. If the analysis succeeds, Twelf stores the output as an OMDoc version of the input file, which is then imported into Hets using standard XML technologies. Hets reads the imported OMDoc input and transforms it into corresponding  $\mathbb{LF}$  signatures and morphisms in their Hets internal representation.

The instantiation of the *LogicalFramework* class specifies the *Base* signature as the LF signature containing the symbols *o* and *ded*, described in Sect. 3. The *writeLogic* method specifies how to implement the *Logic* class for object logics using LF as the framework. While most data types and methods are inherited directly from LF, the method providing the static analysis of basic specifications is implemented differently. As before, it uses Twelf to verify the well-formedness of the specifications; the input signatures are assumed to be given relative to the  $L^{Syn}$  signature supplied when defining the object logic.

---

<sup>2</sup> An institution for  $\mathbb{LF}$  can be defined as for example in [Rab08].

```
newlogic L =
meta F
syntax Ltruth
models Lmod
proofs Lpf
patterns P
```

### 4.3 Example: Adding FOL as a New Logic in Hets

We will now illustrate the steps needed to add first-order logic as a new logic in Hets. First, we need to gather the components of  $FOL$  in a new logic definition, as in Fig. 2. The first three lines are the imports of the morphism  $FOL^{truth}$  from  $Base$  to  $FOL^{Syn}$ , the morphism  $FOL^{mod}$  from  $FOL^{Syn}$  to  $FOL^{Mod}$ , and the morphism  $FOL^{pf}$  from  $FOL^{Syn}$  to  $FOL^{Pf}$  as in Ex. 3, from their respective directories. Note that the directory structure is used to ease the modular design of logics in the Logic Atlas.

```

from ../first-order/syntax/fol get FOL_truth %%FOLtruth
from ../first-order/model_theory/fol get FOL_mod %%FOLmod
from ../first-order/proof_theory/fol get FOL_pf %%FOLpf

newlogic FOL =
  meta LF %%FLF
  syntax FOL_truth %%FOLtruth
  models FOL_mod %%FOLmod
  proofs FOL_pf %%FOLpf
end

```

**Fig. 2.** Defining  $FOL$  as a new object logic.

As a result of calling Hets on the above file, a new sub-directory is added to the source folder of Hets. The subdirectory contains automatically generated files with the instances needed for the logic  $FOL$ . Moreover, the Hets variable containing the list of available logics is updated to include  $FOL$ . After recompiling Hets, the new logic is added to the logic graph of Hets (the node  $FOL$  in Fig. 1 for the dynamically-added logic) and can be used in the same way as any of the built-in logics.

In particular, we can use the new object logic to write specifications. For example, the specification in Fig. 3 uses  $FOL$  as a current logic and declares a constant symbol  $c$  and a predicate  $p$ , together with an axiom that the predicate  $p$  holds for the constant  $c$ . Notice that the syntax for logics specified in a logical framework  $M$  is inherited from the framework (in our case  $LF$ ), but it has been extended with support for sentences, in the usual CASL syntax i.e. prefixed by the “.” character.

Fig. 4 presents the theory of SP as displayed from within Hets; the theory is automatically assumed to extend  $FOL^{Syn}$ . Since in Hets all imports are internally flattened, the theory of SP when displayed will include all symbols from  $FOL^{Syn}$ .

## 5 Conclusion

We have described a prototypical integration of the institution-based Heterogeneous Tool Sets (Hets) with logical frameworks in general and LF and the Twelf

```

logic FOL
spec SP =
  c : i.
  p : i -> o.

  . p c
end

```

**Fig. 3.** Specification in the new object logic.

```

logic FOL
o : type.
ded : o -> type.
i : type.
true : o.
false : o.
not : o -> o.
imp : o -> o -> o.
and : o -> o -> o.
or : o -> o -> o.
forall : (i -> o) -> o.
exists : (i -> o) -> o.
c : i.
p : i -> o.

. p c % (gen_ax_0)

```

**Fig. 4.** Theory of SP

tool in particular. The structuring language used by Hets has a model theoretic semantics, which has been reflected in the proof theoretic logical framework LF by representing models as theory morphisms into some foundation. While LF is the logical framework of our current choice, both the theory and the implementation are so general that other frameworks like Isabelle can be used as well.

Proof theory of the represented logics has been treated only superficially in the present work, but in fact, we have represented proof calculi for all the LATIN logics within LF. Representing models as well has enabled us to formally prove soundness of the calculi. It is straightforward to extend the construction of institution out of logic representations in logical frameworks such that they deliver institutions with proofs. Hets will be extended in the future to deal with proof terms as well.

While the theory and implementation described in this paper make it possible to add logics to Hets in a purely declarative way, further work is needed in order to turn this into a scalable tool. Firstly, the logic translations-as-theory morphisms approach needs to be generalised in order to cover more practically useful examples. Secondly, the new LF generated logics present in Hets need to be connected (via institution comorphisms) to the existing hard-coded logics in order to share the connection of the latter to theorem provers and other tools. Thirdly, it will be desirable to have a declarative interface for specifying the syntax of new logics, such that one is not forced to use the syntax of the logical framework. We are currently examining whether Eclipse and xtext are helpful here. Finally, also the various tool interfaces of Hets should be made more declarative, such that Hets logics specified in a logical framework can be directly connected to theorem provers and other tools, instead of using a comorphism into a hard-coded logic. Then, in the long run, it will be possible to entirely replace the hard-coded logics with declarative logic specifications in a logical framework — and only the latter need to be hard-coded into Hets.

We explicitly invite researchers outside the LATIN project to contribute their logics. This should usually be a matter importing the aspects that are provided by Logic Atlas theories, and LF-encoding the aspects that are not.

*Acknowledgments* This paper mainly addresses the model theoretic side of the logic atlas developed in the LATIN project — funded by the German Research Council (DFG) under grant KO-2428/9-1.

## References

- ABK<sup>+</sup>02. E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. Mosses, D. Sannella, and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196, 2002.
- AHMP98. A. Avron, F. Honsell, M. Miculan, and C. Paravano. Encoding modal logics in logical frameworks. *Studia Logica*, 60(1):161–208, 1998.
- AHMS99. S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an Evolutionary Formal Software-Development Using CASL. In D. Bert, C. Choppo, and P. Mosses, editors, *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1999.
- BC04. Y. Bertot and P. Castéran. *Coq’Art: The Calculus of Inductive Constructions*. Springer, 2004.
- CAB<sup>+</sup>86. R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.
- CELM96. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4, pages 65–89, 1996.
- Chu40. A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.
- Dia08. R. Diaconescu. *Institution-independent Model Theory*. Birkhäuser, 2008.
- GB92. J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.
- HHP93. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
- HR10. F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. Under review, see [http://kwarc.info/frabe/Research/HR\\_folsound\\_10.pdf](http://kwarc.info/frabe/Research/HR_folsound_10.pdf) 2010.
- HST94. R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.
- IR10. M. Iancu and F. Rabe. Formalizing Foundations of Mathematics. Under review, see [http://kwarc.info/frabe/Research/IR\\_foundations\\_10.pdf](http://kwarc.info/frabe/Research/IR_foundations_10.pdf) 2010.
- KMR09. M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN Project, 2009. See <https://trac.omdoc.org/LATIN/>.
- MAH06. T. Mossakowski, S. Autexier, and D. Hutter. Development Graphs - Proof Management for Structured Specifications. *Journal of Logic and Algebraic Programming*, 67(1-2):114–145, 2006.

- Mes89. J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Proceedings, Logic Colloquium, 1987*, pages 275–329. North-Holland, 1989.
- ML74. P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the '73 Logic Colloquium*, pages 73–118. North-Holland, 1974.
- MML07. T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522, 2007.
- MOM96. N. Martí-Oliet and J. Meseguer. Rewriting Logic as a Logical and Semantic Framework. In *Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 352–358, 1996.
- Nor05. U. Norell. The Agda WiKi, 2005. <http://wiki.portal.chalmers.se/agda>
- NPW02. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.
- Pau94. L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
- Pfe00. F. Pfenning. Structural cut elimination: I. intuitionistic and classical logic. *Information and Computation*, 157(1-2):84–141, 2000.
- PS99. F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.
- PS08. A. Poswolsky and C. Schürmann. System Description: Delphin A Functional Programming Language for Deductive Systems. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Metalinguages: Theory and Practice*, pages 135–141. ENTCS, 2008.
- PSK<sup>+</sup>03. F. Pfenning, C. Schürmann, M. Kohlhase, N. Shankar, and S. Owre. The Logosphere Project, 2003. <http://www.logosphere.org/>
- Rab08. F. Rabe. *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen, 2008. Available at <http://kwarc.info/frabe/Research/phdthesis.pdf>.
- Rab10. F. Rabe. A Logical Framework Combining Model and Proof Theory. To be submitted, see [http://kwarc.info/frabe/Research/rabe\\_combining\\_09.pdf](http://kwarc.info/frabe/Research/rabe_combining_09.pdf) 2010.
- RS09. F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, pages 40–48. ACM Press, 2009.
- Soj10. K. Sojakova. Mechanically Verifying Logic Translations, 2010. Master’s thesis, Jacobs University Bremen.
- SS04. C. Schürmann and M. Stehr. An Executable Formalization of the HOL/Nuprl Connection in the Metalogical Framework Twelf. In *11th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2004.

# Project Abstract: Logic Atlas and Integrator (LATIN) \*

Mihai Codescu<sup>1</sup>, Fulya Horozal<sup>2</sup>, Michael Kohlhase<sup>2</sup>, Till Mossakowski<sup>1</sup>, and Florian Rabe<sup>2</sup>

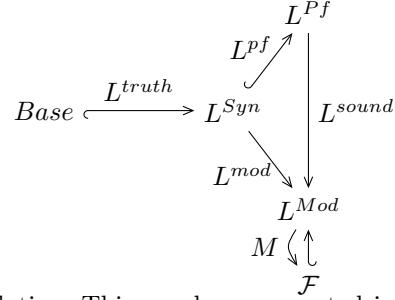
<sup>1</sup> Safe and Secure Cognitive Systems, German Research Centre for Artificial Intelligence (DFKI), Bremen, Germany

<sup>2</sup> Computer Science, Jacobs University Bremen, Germany  
<http://latin.omdoc.org>

LATIN aims at developing methods, techniques, and tools for interfacing logics and related formal systems. These systems are at the core of mathematics and computer science and are implemented in systems like (semi-)automated theorem provers, model checkers, computer algebra systems, constraint solvers, or concept classifiers. Unfortunately, these systems have differing domains of applications, foundational assumptions, and input languages, which makes them non-interoperable and difficult to compare and evaluate in practice.

The LATIN project develops a foundationally unconstrained framework for the representation of logics and translations between them [9, 1]. The LATIN framework (i) subsumes existing proof theoretical frameworks such as LF and model theoretical frameworks such as institutions [3] and (ii) supplants them with a uniform knowledge representation language based on OMDoc. Special attention is paid to generality, modularity, scalability, extensibility, and interoperability.

Logics are represented as theories and translations as theory morphisms. Logic representations formalize the syntax, proof theory, and model theory of a logic within the LATIN framework. The representations of the model theory are parametric in the foundation of mathematics, which is represented as a theory itself; then individual models are represented as theory morphisms into the foundation. This can be represented in a diagram such as the one above, where the syntax of a logic  $L$  is represented as a theory  $L^{Syn}$ , which is then extended with the representation of proof rules to represent the proof theory as  $L^{Pf}$ . Moreover, the model theory of the logic can be represented as a theory  $L^{Mod}$ , based on the representation of a foundation  $\mathcal{F}$  which is included the model theory; the models are represented by the arrow  $M$ . The  $Base$  theory represents the type of formulas and the notion of truth for them. Moreover, we can represent soundness proofs as a morphism




---

\* The LATIN project is supported by the Deutsche Forschungsgemeinschaft (DFG) within grant KO 2428/9-1.

$L^{sound}$  from the proof theory to the model theory of  $L$ . Similarly, logic translations formalize the translations of syntax, proof theory, and model theory. This “logics-as-theories” approach makes system behaviors as well as their represented knowledge interoperable and thus comparable at multiple levels.

The LATIN framework has been implemented generically within the Heterogeneous Tool Set Hets [7] and instantiated with the logical frameworks LF, Isabelle, and Maude. Hets is a general institution-based framework for integration of formal methods and heterogeneous specification and proof management. While Hets implements a large number of logics and translations, their semantics and correctness had previously been determined only by model theoretic arguments. Within the LATIN project, Hets has been extended to support adding logics semi-automatically using a logic specification in one of the supported logical frameworks. This brings the advantage that the logics of Hets are represented fully formally and verified mechanically, and that new logics can be added dynamically.

To evaluate the developed framework and provide a service to the community, the project builds an atlas of logics used in automated reasoning, mathematics, and software engineering. The concrete logic representations span over 1000 theories and morphisms and can be found at the project web site, they include (i) *Type theory*, including a modular development of the lambda cube, Martin-Löf Type Theory, and Isabelle, (ii) *Logics*, including first-order, higher-order, modal, and description logics, (iii) *Set theory* including ZFC and the Mizar variant of Tarski-Grothendieck set theory. The atlas also includes a growing number of logic translations including, e.g., the relativization translations from modal, description, and sorted first-order logics to unsorted first-order logic, the interpretation of type theory in set theory, the negative translation from classical to intuitionistic logic, and the translation from first to higher-order logic. Elaborate case studies were documented in [4, 5, 10]. The LATIN atlas is extensible, and new logics can be added easily — including the reuse of already formalized logic features — and related to the existing logics via translations.

To make the logic atlas scalable, we base it on the knowledge representation language MMT [11]. MMT refines the markup language for structured theories that is part of OMDoc and provides a formal semantics for it. Moreover, MMT comes with a scalable infrastructure [6] centered around a flexible and foundation-independent API.

In the authoring work flow of LATIN, representations are written in Twelf [8] using our module system for it [12]. Twelf converts the content into OMDoc/MMT, which indexes and stores it in the SVN+XML database TNTBase [13]. In the application work flow, these OMDoc/MMT documents are imported into Hets. In the presentation work flow, the MMT web server uses XQueries to retrieve LATIN content and user-defined notations from TNTBase, which are used to render the content as JOBAD-enabled [2] XHTML+MathML. All stages of this pipeline are semantics-aware so that, for example, the web server can offer interactive dynamic services such as definition lookup or toggling the display of inferred types.

The browsable version of the atlas is available at the project web site. When browsing it, keep in mind that the logical framework, the formalizations in it, and the whole infrastructure processing them are ongoing work and thus subject to both constant improvement and temporary failures.

## References

1. M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, F. Rabe, and K. Sojakova. Towards Logical Frameworks in the Heterogeneous Tool Set Hets. In *Workshop on Abstract Development Techniques*, Lecture Notes in Computer Science. Springer, 2011. To appear.
2. J. Gićeva, C. Lange, and F. Rabe. Integrating Web Services into Active Mathematical Documents. In J. Carette and L. Dixon and C. Sacerdoti Coen and S. Watt, editor, *Intelligent Computer Mathematics*, volume 5625 of *Lecture Notes in Computer Science*, pages 279–293. Springer, 2009.
3. J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.
4. F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. *Theoretical Computer Science*, 2011. To appear, see [http://kwarc.info/frabe/Research/HR\\_folsound\\_10.pdf](http://kwarc.info/frabe/Research/HR_folsound_10.pdf).
5. M. Iancu and F. Rabe. Formalizing Foundations of Mathematics. *Mathematical Structures in Computer Science*, 2011. To appear, see [http://kwarc.info/frabe/Research/IR\\_foundations\\_10.pdf](http://kwarc.info/frabe/Research/IR_foundations_10.pdf).
6. M. Kohlhase, F. Rabe, and V. Zholudev. Towards MKM in the Large: Modular Representation and Scalable Software Architecture. In S. Autexier, J. Calmet, D. Delahaye, P. Ion, L. Rideau, R. Rioboo, and A. Sexton, editors, *Intelligent Computer Mathematics*, volume 6167 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2010.
7. T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522, 2007.
8. F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.
9. F. Rabe. A Logical Framework Combining Model and Proof Theory. Submitted to Mathematical Structures in Computer Science, see [http://kwarc.info/frabe/Research/rabe\\_combining\\_09.pdf](http://kwarc.info/frabe/Research/rabe_combining_09.pdf), 2010.
10. F. Rabe. Representing Isabelle in LF. In K. Crary and M. Miculan, editors, *Logical Frameworks and Meta-languages: Theory and Practice*, volume 34 of *EPTCS*, pages 85–100, 2010.
11. F. Rabe and M. Kohlhase. A Scalable Module System. Under review, see <http://arxiv.org/abs/1105.0548>, 2011.
12. F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, pages 40–48. ACM Press, 2009.
13. V. Zholudev and M. Kohlhase. TNTBase: a Versioned Storage for XML. In *Proceedings of Balisage: The Markup Conference 2009*, volume 3 of *Balisage Series on Markup Technologies*. Mulberry Technologies, Inc., 2009.

# Representing Model Theory in a Type-Theoretical Logical Framework<sup>☆</sup>

Fulya Horozal<sup>a</sup>, Florian Rabe<sup>a</sup>

<sup>a</sup>*Jacobs University Bremen, Germany*

---

## Abstract

In a broad sense, logic is the field of formal languages for knowledge and truth that have a formal semantics. It tends to be difficult to give a narrower definition because very different kinds of logics exist. One of the most fundamental contrasts is between the different methods of assigning semantics. Here two classes can be distinguished: model theoretical semantics based on a foundation of mathematics such as set theory, and proof theoretical semantics based on an inference system possibly formulated within a type theory.

Logical frameworks have been developed to cope with the variety of available logics unifying the underlying ontological notions and providing a meta-theory to reason abstractly about logics. While these have been very successful, they have so far focused on either model or proof theoretical semantics. We contribute to a unified framework by showing how a type/proof theoretical Edinburgh Logical Framework LF can be applied to the representation of model theoretical logics.

We give a comprehensive formal representation of first-order logic covering both its proof and its model theoretical semantics and its soundness in LF. For the model theory, we have to represent the mathematical foundation itself in LF, and we provide two solutions for that. Firstly, we give a meta-language that is strong enough to represent the model theory while being simple enough to be treated as a fragment of untyped set theory. Secondly, we represent Zermelo-Fraenkel set theory and show how it subsumes our meta-language. All representations are given in and mechanically verified by the Twelf implementation of LF. Moreover, we use the Twelf module system to treat all connectives and quantifiers independently. Thus, individual connectives are available for reuse when representing other logics, and we obtain the first version of a feature library from which logics can be pieced together.

Our results and methods are not restricted to first-order logic and scale to a wide variety of logical systems thus demonstrating the feasibility of comprehensively formalizing large scale representation theorems in a logical framework.

---

---

<sup>☆</sup>This work was partially supported by the Deutsche Forschungsgemeinschaft within grant KO 2428/9-1.

## Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>  | <b>2</b>  |
| <b>2</b> | <b>Preliminaries</b>   | <b>4</b>  |
| 2.1      | First-Order Logic . . . . .                                      | 4         |
| 2.2      | LF and Twelf . . . . .   | 7         |
| <b>3</b> | <b>A Logical Framework Combining Proof and Model Theory</b>      | <b>10</b> |
| <b>4</b> | <b>Representing First-Order Logic</b>                            | <b>12</b> |
| 4.1      | Syntax . . . . .   | 13        |
| 4.2      | Proof Theory . . . . .   | 15        |
| 4.3      | A Meta-Language for the Representation of Model Theory . . . . . | 17        |
| 4.4      | Model Theory . . . . .   | 20        |
| 4.5      | Adequacy . . . . .   | 23        |
| 4.6      | Soundness . . . . .  | 27        |
| <b>5</b> | <b>Representing Set-Theoretical Model Theory</b>                 | <b>27</b> |
| 5.1      | Representing Set Theory . . . . .                                | 29        |
| 5.2      | Viewing Higher-Order Logic in Set Theory . . . . .               | 35        |
| 5.3      | Viewing Model Theory in Set Theory . . . . .                     | 35        |
| 5.4      | Representing Model Theory . . . . .                              | 36        |
| 5.5      | Adequacy . . . . .   | 38        |
| <b>6</b> | <b>Related Work</b>  | <b>41</b> |
| <b>7</b> | <b>Conclusion</b>  | <b>43</b> |

### 1. Introduction

Logic has been an important research topic in mathematics and computer science since the foundational crisis of mathematics. Research on logics has included the difficult and sometimes contentious question how to choose the ontological foundations of logic. Logical frameworks have proved an important research result to answer this question – they are abstract formalisms that permit the formal definition of specific logics.

Today we observe that there are two groups of logical frameworks: those based on set theoretical foundations of mathematics that characterize logics *model theoretically*, and those based on type theoretical foundations that characterize logics *proof theoretically*. The former go back to Tarski’s view of consequence ([Tar33], [TV56]) with institutions ([GB92], [GR02]) and general logics ([Mes89]) being state of the art examples. The latter are usually based on the Curry-Howard correspondence ([CF58], [How80]), examples being Automath ([dB70]), Isabelle ([Pau94]), and the Edinburgh Logical Framework (LF, [HHP93]).

While some model-theoretical frameworks attempt to integrate proof theory (e.g., [Mes89, MGD05, Dia06]), the opposite integration is less developed. This is unfortunate because many of the results and techniques developed for proof theoretical logics could also benefit model-theoretical reasoning.

We are particularly interested in logic encodings in the Edinburgh Logical Framework (LF), which is related to Martin-Löf type theory and can be seen as the dependently-typed corner of the  $\lambda$ -cube ([Bar92]). LF represents syntax and proof theoretical semantics of a logic using higher order abstract syntax and the judgments-as-types paradigm ([ML96]). This has proved very successful for proof-theoretical logic representations ([HST94, AHMP98, Pfe00, NSM01]).

In [Rab08], we introduced a framework that attempts to preserve and exploit the respective advantages of model and proof theoretical representation. The central idea is to also represent the model theory of a logic in a type-theoretical logical framework by specifying models in a suitable meta-language.

In this paper we show how to implement such logic representations in LF. We pick LF because we have recently equipped the Twelf implementation of LF with a strong module system [PS99, RS09]. This module system is rigorously based on theory morphisms, which have proved very successful to reason about model-theoretical logic representations (e.g., [GB92, AHMS99, SW83]). Therefore, it is particularly appropriate for an encoding that combines proof- and model-theoretical aspects.

Our central results are (i) the full representation of first-order logic (FOL) comprising syntax, proof theory, and model theory, and (ii) a formal proof of the soundness of FOL based on this representation. In particular, the soundness proof is verified mechanically by the LF implementation Twelf. While this is interesting in itself, the main value of our work is not the encoding but the methodology we employ. We use FOL as an example logic mainly because it is most widely known and thus interferes least with the rather abstract subject matter. Other logics can be represented analogously.

Furthermore, we use the LF module system for a modular development of syntax, proof theory, model theory, and soundness, i.e., all connectives and quantifiers are treated separately in all four parts of the encoding. These modules can be reused flexibly to encode other logics. For example, we obtain encodings for any logic that arises by omitting some connectives or quantifiers from FOL. Less trivially, the encoding of each connective or quantifier can be reused for any logic using them. For example, this enabled one of our students to extend the work presented here to sorted FOL within two days.

Our approach is especially interesting when studying rarer or new logics, for which no smoothed-out semi-formal definitions are available yet. In particular, our framework can be used for the rapid prototyping of logics. Since it covers both proof and model theory, it permits an approach that we call *syntax-semantics-codesign*, to coin a phrase: Researchers can give a fully formal and mechanically verified definition of a formal language and its semantics at a level of convenience and elegance that competes with working it out on paper.

In Sect. 2, we describe some preliminaries and introduce some notation: FOL in Sect. 2.1, and LF in Sect. 2.2. In Sect. 3, we sketch the framework we will

use. The main sections of this paper are Sect. 4 and 5. In the former, we give the encoding of FOL in LF where we use a variant of higher-order logic as a simple and convenient meta-language to represent the models. In the latter, we extend the encoding to cover set theory itself as a foundation of mathematics, in which models are expressed. Thus, we can give a comprehensive representation of FOL and its set-theoretical model theory in LF. Both in Sect. 4 and 5, we describe the encoding of FOL in a way that makes the general methodology apparent and provides a template for the encoding of other logics.

A preliminary version of this paper has appeared as [HR09]. The present version has been fully revised and substantially extended. Most importantly, the encoding of set theory, which was only sketched in [HR09], has been worked out. Among the changes we made, two are especially notable. Firstly, we changed the meta-language employed to represent models from Martin-Löf type theory to higher-order logic. This was motivated by the desire to separate types and propositions rather than identify them. Secondly, in [HR09], we identified some features missing in the implementation of the LF module system. These have been added by now, which enabled us to completely refactor the LF encodings.

Our approach is very extensible, and we have treated or are currently working on corresponding representations of sorted, higher-order, and description logics. These are part of a logic atlas that is developed as a collaborative research effort within the LATIN project ([KMR09]). All Twelf sources are available from the project website.

## 2. Preliminaries

### 2.1. First-Order Logic

In this section, we will introduce first-order logic in order to give an overview of the definitions and notations we will use. The definitions here also serve as the reference definitions when proving the adequacy of our encodings.

**Definition 1** (Signatures). A FOL-signature is a triple  $(\Sigma_f, \Sigma_p, ar)$  where  $\Sigma_f$  and  $\Sigma_p$  are disjoint sets of function and predicate symbols, respectively, and  $ar : \Sigma_f \cup \Sigma_p \rightarrow \mathbb{N}$  assigns arities to symbols. We will treat constants and boolean variables as the special case of arity 0.

**Definition 2** (Expressions). A FOL-context is a list of variables. For a signature  $\Sigma$  and a context  $\Gamma$ , the terms over  $\Sigma$  and  $\Gamma$  are formed from the variables in  $\Gamma$  and the application of function symbols  $f \in \Sigma_f$  to terms according to  $ar(f)$ . The formulas over  $\Sigma$  and  $\Gamma$  are formed from the application of predicate symbols  $p \in \Sigma_p$  to a number of terms according to  $ar(p)$  as well as  $\doteq$ , *true*, *false*,  $\neg$ ,  $\wedge$ ,  $\vee$ ,  $\Rightarrow$ ,  $\forall$ , and  $\exists$  in the usual way. Formulas in the empty context are called  $\Sigma$ -sentences, and we write  $Sen(\Sigma)$  for the set of sentences.

**Definition 3** (Theories). A FOL-theory is a pair  $(\Sigma, \Theta)$  for a signature  $\Sigma$  and a set  $\Theta \subseteq Sen(\Sigma)$  of axioms.

**Definition 4** (Signature Morphisms). Given two signatures  $\Sigma = (\Sigma_f, \Sigma_p, ar)$  and  $\Sigma' = (\Sigma'_f, \Sigma'_p, ar')$ , a FOL-signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$  is an arity-preserving mapping from  $\Sigma_f$  to  $\Sigma'_f$  and from  $\Sigma_p$  to  $\Sigma'_p$ .

The homomorphic extension of  $\sigma$  – which we also denote by  $\sigma$  – is the mapping from terms and formulas over  $\Sigma$  to terms and formulas over  $\Sigma'$  that replaces every symbol  $s \in \Sigma_f \cup \Sigma_p$  with  $\sigma(s)$ . The sentence translation  $Sen(\sigma) : Sen(\Sigma) \rightarrow Sen(\Sigma')$  arises as the special case of applying  $\sigma$  to sentences.

*Example 5* (Monoids and Groups). We will use the theories  $Monoid = (MonSig, MonAx)$  and  $Group = (GrpSig, GrpAx)$  of monoids and groups as running examples.  $MonSig_f$  is the set  $\{\circ, e\}$  where  $\circ$  is binary (written infix) and  $e$  is nullary, and  $MonSig_p$  is empty.  $MonAx$  consists of the axioms for

- associativity:  $\forall x \forall y \forall z x \circ (y \circ z) \doteq (x \circ y) \circ z$ ,
- left-neutrality:  $\forall x e \circ x \doteq x$ ,
- right-neutrality:  $\forall x x \circ e \doteq x$ .

The theory  $Group$  extends  $Monoid$ , i.e.,  $GrpSig$  adds a unary function symbol  $inv$  (written as superscript  $-1$ ) to  $MonSig$ , and  $GrpAx$  adds axioms for

- left-inverseness:  $\forall x x^{-1} \circ x \doteq e$ ,
- right-inverseness:  $\forall x x \circ x^{-1} \doteq e$

to  $MonAx$ . The inclusion mapping  $MonGrp$  is a signature morphism from  $MonSig$  to  $GrpSig$ . It is also a theory morphism from  $Monoid$  to  $Group$ .

There are various ways to define the *proof theory* of FOL. In this paper we choose the natural deduction calculus (ND) with introduction and elimination rules. We will use the phrase *proof theoretical semantics* when speaking about the induced provability relation; we will not consider proof normalization, which some authors mean when using that phrase.

**Definition 6** (Proof Theoretical Theorems). Given a theory  $(\Sigma, \Theta)$ , we say that  $F \in Sen(\Sigma)$  is a theorem of  $(\Sigma, \Theta)$  if the judgment  $F_1, \dots, F_n \vdash_{\Sigma} F$  is derivable for some  $\{F_1, \dots, F_n\} \subseteq \Theta$  using the calculus shown in Fig. 1. We write this as  $\Theta \vdash_{\Sigma} F$ .

**Definition 7** (Proof Theoretical Theory Morphisms). A signature morphism from  $\Sigma$  to  $\Sigma'$  is a *proof theoretical theory morphism* from  $(\Sigma, \Theta)$  to  $(\Sigma', \Theta')$ , written  $\sigma : (\Sigma, \Theta) \xrightarrow{P} (\Sigma', \Theta')$ , if  $Sen(\sigma)$  maps the axioms of  $(\Sigma, \Theta)$  to theorems of  $(\Sigma', \Theta')$ , i.e., for all  $F \in \Theta$ ,  $\Theta' \vdash_{\Sigma'} Sen(\sigma)(F)$  holds.

**Lemma 8** (Proof Translation). Assume a theory morphism  $\sigma : (\Sigma, \Theta) \rightarrow (\Sigma', \Theta')$ . If  $F$  is a theorem of  $(\Sigma, \Theta)$ , then  $Sen(\sigma)(F)$  is a theorem of  $(\Sigma', \Theta')$ . In other words, provability is preserved along theory morphisms.

We develop the *model theory* of FOL as an institution ([GB92]).

|  |   |  |
|--|---|--|
| $\frac{}{\Theta \vdash_{\Sigma} \text{true}}$  | $\frac{\Theta \vdash_{\Sigma} \text{false}}{\Theta \vdash_{\Sigma} F}$  |  |
| $\frac{\Theta, F \vdash_{\Sigma} \text{false}}{\Theta \vdash_{\Sigma} \neg F}$   | $\frac{\Theta \vdash_{\Sigma} \neg F \quad \Theta \vdash_{\Sigma} F}{\Theta \vdash_{\Sigma} \text{false}}$  |  |
| $\frac{\Theta \vdash_{\Sigma} F \quad \Theta \vdash_{\Sigma} G}{\Theta \vdash_{\Sigma} F \wedge G}$  | $\frac{\Theta \vdash_{\Sigma} F \wedge G}{\Theta \vdash_{\Sigma} F}$  | $\frac{\Theta \vdash_{\Sigma} F \wedge G}{\Theta \vdash_{\Sigma} G}$   |
| $\frac{\Theta, F \vdash_{\Sigma} G}{\Theta \vdash_{\Sigma} F \Rightarrow G}$   | $\frac{\Theta \vdash_{\Sigma} F \Rightarrow G \quad \Theta \vdash_{\Sigma} F}{\Theta \vdash_{\Sigma} G}$  |  |
| $\frac{\Theta \vdash_{\Sigma} F}{\Theta \vdash_{\Sigma} F \vee G}$   | $\frac{\Theta \vdash_{\Sigma} G}{\Theta \vdash_{\Sigma} F \vee G}$  | $\frac{\Theta \vdash_{\Sigma} F \vee G \quad \Theta, F \vdash_{\Sigma} H \quad \Theta, G \vdash_{\Sigma} H}{\Theta \vdash_{\Sigma} H}$ |
| $\frac{\Theta \vdash_{\Sigma} F \quad x \text{ fresh}}{\Theta \vdash_{\Sigma} \forall x F}$  | $\frac{\Theta \vdash_{\Sigma} \forall x F}{\Theta \vdash_{\Sigma} F [x/t]}$   |  |
| $\frac{\Theta \vdash_{\Sigma} F [x/t]}{\Theta \vdash_{\Sigma} \exists x F}$  | $\frac{\Theta \vdash_{\Sigma} \exists x F \quad x \text{ fresh} \quad \Theta, F \vdash_{\Sigma} H}{\Theta \vdash_{\Sigma} H}$                               |  |
| $\frac{F \in \Theta}{\Theta \vdash_{\Sigma} F}$  | $\frac{}{\Theta \vdash_{\Sigma} F \vee \neg F}$   |  |
| $\frac{}{\Theta \vdash_{\Sigma} t \doteq t}$   | $\frac{\Theta \vdash_{\Sigma} s \doteq t}{\Theta \vdash_{\Sigma} t \doteq s}$   | $\frac{\Theta \vdash_{\Sigma} r \doteq s \quad \Theta \vdash_{\Sigma} s \doteq t}{\Theta \vdash_{\Sigma} r \doteq t}$                  |
| $\frac{\Theta \vdash_{\Sigma} s_i \doteq t_i \quad f \in \Sigma_f \quad ar(f)=n}{\Theta \vdash_{\Sigma} f(s_1, \dots, s_n) \doteq f(t_1, \dots, t_n)}$ | $\frac{\Theta \vdash_{\Sigma} s_i \doteq t_i \quad p \in \Sigma_p \quad ar(p)=n}{\Theta \vdash_{\Sigma} p(s_1, \dots, s_n) \Rightarrow p(t_1, \dots, t_n)}$ |  |

Figure 1: Proof Rules

**Definition 9** (Models of a FOL-Signature). A FOL-model of a signature  $\Sigma$  is a pair  $(U, I)$  where  $U$  is a non-empty set (called the *universe*) and  $I$  is an interpretation function of  $\Sigma$ -symbols such that

- $f^I \in U^{U^n}$  for  $f \in \Sigma_f$  with  $ar(f) = n$ ,
- $p^I \subseteq U^n$  for  $p \in \Sigma_p$  with  $ar(p) = n$ .

We write  $Mod(\Sigma)$  for the class of  $\Sigma$ -models.

**Definition 10** (Model Theoretical Semantics). Assume a signature  $\Sigma$ , a context  $\Gamma$ , and a  $\Sigma$ -model  $M = (U, I)$ . An *assignment* is a mapping from  $\Gamma$  to  $U$ . For an assignment  $\alpha$ , the interpretations  $\llbracket t \rrbracket^{M, \alpha} \in U$  of terms  $t$  and  $\llbracket F \rrbracket^{M, \alpha} \in \{0, 1\}$  of formulas  $F$  over  $\Sigma$  and  $\Gamma$  are defined in the usual way by induction on the syntax. Given a sentence  $F$ , we write  $M \models_{\Sigma} F$  if  $\llbracket F \rrbracket^M = 1$ .

Given a theory  $(\Sigma, \Theta)$ , we write the class of  $(\Sigma, \Theta)$ -models as

$$Mod(\Sigma, \Theta) = \{M \in Mod(\Sigma) \mid M \models_{\Sigma} F \text{ for all } F \in \Theta\}$$

**Definition 11** (Model Theoretical Theorems). Given a theory  $(\Sigma, \Theta)$ , we say that  $F \in Sen(\Sigma)$  is a *model theoretical theorem* of  $(\Sigma, \Theta)$  if the following holds for all  $\Sigma$ -models  $M$ : If  $M \models_{\Sigma} A$  for all  $A \in \Theta$ , then also  $M \models_{\Sigma} F$ .

**Definition 12** (Model Reduction). Given a signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$  and a  $\Sigma'$ -model  $M' = (U, I')$ , we obtain a  $\Sigma$ -model  $(U, I)$ , called the *model reduct* of  $M'$  along  $\sigma$ , by putting  $s^I = \sigma(s)^{I'}$  for all symbols of  $\Sigma$ . We write  $Mod(\sigma) : Mod(\Sigma') \rightarrow Mod(\Sigma)$  for the induced model reduction.

**Definition 13** (Model Theoretical Theory Morphisms). Given two theories  $(\Sigma, \Theta)$  and  $(\Sigma', \Theta')$ , a *model theoretical theory morphism* from  $(\Sigma, \Theta)$  to  $(\Sigma', \Theta')$ , written  $\sigma : (\Sigma, \Theta) \xrightarrow{M} (\Sigma', \Theta')$ , is a signature morphism from  $\Sigma$  to  $\Sigma'$  such that  $Mod(\sigma)$  reduces models of  $(\Sigma', \Theta')$  to models of  $(\Sigma, \Theta)$ , i.e, for all  $M' \in Mod(\Sigma', \Theta')$ , we have  $Mod(\sigma)(M') \in Mod(\Sigma, \Theta)$ .

**Lemma 14** (Satisfaction Condition). Assume a FOL-signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$ , a  $\Sigma$ -sentence  $F$ , and a  $\Sigma'$ -model  $M'$ . Then  $M' \models_{\Sigma'} Sen(\sigma)(F)$  iff  $Mod(\sigma)(M') \models_{\Sigma} F$ .

*Example 15* (Continued). The integers form a model  $Int = (\mathbb{Z}, +, 0, -)$  for the theory of groups (where we use a tuple notation to give the universe and the interpretations of  $\circ$ ,  $e$ , and  $inv$ , respectively). The model reduction  $Mod(MonGrp)(Int) = (\mathbb{Z}, +, 0)$  along  $MonGrp$  yields the integers seen as a model of the theory of monoids.

We have given both proof theoretical and model theoretical definitions of *theorem* and *theory morphism*. In general, these must be distinguished to avoid a bias towards proof or model theory. However, they coincide if a logic is sound and complete:

**Theorem 16** (Soundness and Completeness). Assume a FOL-theory  $(\Sigma, \Theta)$  and a  $\Sigma$ -sentence  $F$ . Then  $\Theta \vdash_{\Sigma} F$  iff  $\Theta \models_{\Sigma} F$ . Therefore, for a FOL-signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$ , we have  $\sigma : (\Sigma, \Theta) \xrightarrow{P} (\Sigma', \Theta')$  iff  $\sigma : (\Sigma, \Theta) \xrightarrow{M} (\Sigma', \Theta')$ .

## 2.2. LF and Twelf

LF ([HHP93](#)) is a dependent type theory that extends simple type theory with dependent function types. We will work with the Twelf implementation of LF ([PS99](#)). The main use of LF and Twelf is as a *logical framework* in which deductive systems are represented.

We will develop the syntax and semantics of LF along an example representation of simple type theory (STT). Typically, *kinded type families* are declared to represent the syntactic classes of the system. For STT, we declare

```
tp   :  type
tm  :  tp → type
```

Here `type` is the LF-kind of types, and `tp` is an LF-type whose LF-terms represent the STT-types. And `tp → type` is the kind of type families that are indexed by terms of LF-type `tp`; then `tm A` is the LF-type whose terms represent the STT-terms of type `A`.

*Typed constants* are declared to represent the expressions of the represented system. For STT, we add

$$\begin{array}{lll} \Rightarrow & : tp \rightarrow tp \rightarrow tp & \% \text{infix right } 0 \Rightarrow \\ @ & : tm(A \Rightarrow B) \rightarrow tm A \rightarrow tm B & \% \text{infix left } 1000 @ \\ \lambda & : (tm A \rightarrow tm B) \rightarrow tm(A \Rightarrow B) \end{array}$$

Here `⇒` is a low-binding right-associative infix symbol that takes two `tp`-arguments and returns a `tp`. It represents STT-function type formation. In the following, we will always omit the fixity and associativity declarations if they are clear from the context. In particular, besides `⇒` and `@`, binary symbols such as connectives and equality are always assumed to be declared as infix.

`@` is a strong-binding left-associative infix symbol that takes two arguments – an STT-term of type `A ⇒ B` and an STT-term of type `A` – and returns an STT-term of type `B`. It represents STT-function elimination, i.e., application. In the declaration of `@`, `A` and `B` are free variables. These variables are implicitly  $\Pi$ -bound at the outside. The full type of `@` is  $\Pi_{A:tp}\Pi_{B:tp}tm(A \Rightarrow B) \rightarrow tm A \rightarrow tm B$ , i.e., `@` really takes 4 arguments. This uses the main feature of dependent type theory: The first two arguments `A` and `B` may occur in the types of the later arguments and in the return type. Twelf treats `A` and `B` as *implicit arguments* and infers their values from the other arguments. Thus, we can write `f @ a` instead of `@ A B f a`.

Finally, `λ` represents STT-function introduction, i.e., abstraction. `λ` is declared using *higher-order abstract syntax*. LF-functions of type `S → T` are in bijection to the terms of type `T` with a free variable of type `S`; thus, higher-order arguments can be used to represent binders. The above `λ` takes a term of STT-type `B` with a free variable of type `A` represented as an LF term of type `tm A → tm B`. It returns an STT term of type `A ⇒ B`.

We will always use Twelf notation for the LF primitives of binding and application: The type  $\Pi_{x:A}B(x)$  of dependent functions taking  $x : A$  to an element of  $B(x)$  is written  $\{x : A\} B$ , and the function term  $\lambda_{x:A}t(x)$  taking  $x : A$  to  $t(x)$  is written  $[x : A] t$ . (Therefore, `λ` is available for user-declared symbols.) In particular, in the above example, the STT-term  $\lambda_{x:A}$  is represented as the LF-term  $\lambda[x : A] t$ . Finally, we write `A → B` instead of  $\{x : A\} B$  if `x` does not occur in `B`, and we will also omit the types of bound variables if they can be inferred.

LF employs the Curry-Howard correspondence to represent proofs-as-term ([CF58], [How80]) and extends it to the *judgments-as-types* methodology ([ML96]). For example, we can turn the above STT into a logic by adding a type `prop` of propositions and a truth judgment `True` on it:

```

prop   : type
True   : prop → type
⇒     : prop → prop → prop
⇒ E   : True (F ⇒ G) → True F → True G

```

Here the type  $\text{True } F$  represents the judgment that  $F$  is true. A judgment  $J$  is proved if there is a term of type  $J$ . Consequently, all axioms and inference rules such as the implication elimination rule  $\Rightarrow E$  are represented as constants, and proofs of  $F$  are represented as terms of type  $\text{True } F$ .

Finally, an LF *signature* is a list of kinded type family declarations  $a : K$  and typed constant declarations  $c : A$ . Both may carry definitions, i.e.,  $c : A = t$  introduces  $c$  as an abbreviation for  $t$ . This yields the following grammar for the fragment of LF we will use:

|                |          |   |
|----------------|----------|---|
| Signatures     | $\Sigma$ | $::= \cdot \mid \Sigma, c : A \mid \Sigma, c : A = t \mid \Sigma, a : K \mid \Sigma, a : K = A$ |
| Morphisms      | $\sigma$ | $::= \cdot \mid \sigma, c := t \mid \sigma, a := A$   |
| Kinds:         | $K$      | $::= \text{type} \mid A \rightarrow K$  |
| Type families: | $A, B$   | $::= a \mid [x : A] B \mid B t \mid \{x : A\} B$  |
| Terms:         | $s, t$   | $::= c \mid x \mid [x : A] t \mid s t$  |

Here we have already included LF *signature morphisms*. Given two signatures  $\Sigma$  and  $\Sigma'$ , a signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$  is a typing-preserving map of  $\Sigma$ -symbols to  $\Sigma'$ -expressions. Thus,  $\sigma$  maps every constant  $c : A$  of  $\Sigma$  to a term  $\sigma(c) : \bar{\sigma}(A)$  and every type family symbol  $a : K$  to a type family  $\sigma(a) : \bar{\sigma}(K)$ . Here,  $\bar{\sigma}$  is the homomorphic extension of  $\sigma$  to  $\Sigma$ -expressions, and we will write  $\sigma$  instead of  $\bar{\sigma}$  from now on.

Signature morphisms preserve typing, i.e., if  $\vdash_{\Sigma} E : F$ , then  $\vdash_{\Sigma'} \sigma(E) : \sigma(F)$ . In particular, because  $\sigma$  must map all axioms or inference rules declared in  $\Sigma$  to proofs or derived inference rules, respectively, over  $\Sigma'$ , signature morphisms preserve the provability of judgments.

We will write  $\sigma \sigma'$  for the diagram-order composition of signature morphisms, i.e.,  $(\sigma \sigma')(E) = \sigma'(\sigma(E))$ .

The module system for LF and Twelf ([RS09]) is based on the notion of signature morphisms ([HST94]). The toplevel declarations of modular LF declare named signatures and named signature morphisms, called *views*, e.g.,

```
%sig S = {Σ}. %sig T = {Σ'}. %view v : S → T = {σ}
```

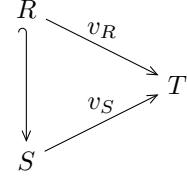
Since signature morphisms must map axioms to proofs, a view has the flavor of a theorem establishing a translation from  $S$  to  $T$  or a representation of  $S$  in  $T$  or a refinement of  $S$  into  $T$ .

Besides views, the module system provides *inclusion* morphisms that hold by definition: `%include S` declared in  $T$  copies all declarations of  $S$  into  $T$  (thus changing  $T$ ). This represents an inheritance or import relationship from  $S$  to  $T$ . The inclusion relation is transitive, and multiple inclusions of the same signature are identified. Twelf uses qualified names to access included symbols, but we will simply assume that included symbols  $c$  of  $S$  are accessible as  $c$  within  $T$ .

Views can be given modularly, too. If  $S$  includes  $R$ , then a view  $v_S$  from  $S$  to  $T$  must map all constants of  $S$ , i.e., also those of  $R$ . Often a view  $v_R$  from  $R$  to  $T$  is already present. In that case  $v_S$  can include  $v_R$  via `%include v_R` and only give maps for the symbols of  $S$ . If  $v_S$  is defined like that, the triangle on the right always commutes.

Thus, we arrive at the following grammar for the fragment of modular LF we will use. Here we use  $E$  for a kind, type family, or term as defined above:

|            |   |
|------------|---|
| Start      | $G ::= \cdot \mid G, D_T \mid G, D_v$   |
| Signatures | $D_T ::= \%sig T = \{\Sigma\}$  |
| Views      | $D_v ::= \%view v : S \rightarrow T = \{\sigma\}$                                     |
| Sign. body | $\Sigma ::= \cdot \mid \Sigma, c : E \mid \Sigma, c : E = E \mid \Sigma, \%include S$ |
| View body  | $\sigma ::= \cdot \mid \sigma, c := E \mid \sigma, \%include v$                       |



### 3. A Logical Framework Combining Proof and Model Theory

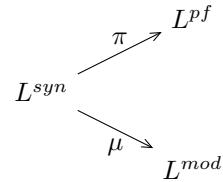
LF was designed as a language for the representation of formal systems. Similarly, the LF module system was designed as a language for the representation of translations between formal systems. This makes it a very appropriate framework for the comprehensive representation of a logic where translations – between different signatures of a logic as well as between syntax and semantics – are prevalent.

In the following, we will give an overview of the logical framework we gave in [Rab08, Rab10]. We will not actually give the framework itself, which requires a further level of abstraction beyond the scope of this paper. Instead, we define what the encoding of an individual logic looks like.

We assume a logic  $L$  defined along the lines of Sect. 2.1. An encoding of  $L$  in our framework consists of three LF signatures  $L^{syn}$  for the syntax,  $L^{pf}$  for the proof theory, and  $L^{mod}$  for the model theory, as well as two LF signature morphisms  $\pi$  and  $\mu$  that translate the syntax into proof and model theory.

$L^{syn}$  contains LF declarations for all symbols occurring in  $L$ -formulas. Type declarations are used for syntactic classes, e.g., sorts, terms, formulas, and judgments (typically including a truth judgment), and constant declarations are used for individual connectives, quantifiers, sorts, functions, predicates, axioms, etc.  $L^{pf}$  is typically an extension of  $L^{syn}$ , i.e.,  $\pi$  is an inclusion morphism.  $L^{pf}$  includes constant declarations for the axioms and inference rules of  $L$ ; it may also contain type declarations for auxiliary judgments.

$L^{syn}$  typically declares at least a type  $o$  of formulas and a type family  $ded : o \rightarrow \text{type}$  for the truth judgment. In the simplest case,  $L^{pf}$  only adds inference rules for  $ded$  to  $L^{syn}$ .



$L^{mod}$  contains declarations that describe models. For model-theoretically motivated logics such as first-order logic,  $L^{syn}$  and  $L^{mod}$  have a similar structure – after all for many logics  $L$ , the syntax was introduced as a way to describe the models in the first place. But for proof-theoretically motivated logics like some modal logics or intuitionistic logics, the model theory was developed a posteriori. For these, the  $L^{syn}$  and  $L^{mod}$  may vary considerably, e.g.,  $L^{mod}$  might contain declarations to describe Kripke frames.

$L^{mod}$  typically includes some meta-language that is used to reason about the truth in models, e.g., set theory. In the simplest case  $\mu$  translates  $o$  to a type of truth values and  $ded$  to a predicate that holds for the designated truth values. Then  $L^{mod}$  axiomatizes the translation of formulas to truth values.

A specific signature  $\Sigma$  of  $L$  is represented as an extension of  $L$ . This corresponds to the uniform logic encodings in LF given in [HST94]. For example, signatures of propositional logic are sets of propositional variables, and the set  $\Sigma = \{p_1, \dots, p_n\}$  is encoded as the LF-signature  $\Sigma^{syn} = L^{syn}$ ,  $p_1 : o, \dots, p_n : o$ . By merging  $\Sigma^{syn}$  with  $L^{pf}$  and  $L^{mod}$ , we obtain  $\Sigma^{pf}$  and  $\Sigma^{mod}$ , respectively. This leads to the diagram below.

$$\begin{array}{ccccc} & & L^{pf} & \longrightarrow & \Sigma^{pf} \\ & \nearrow \pi & & \searrow & \\ L^{syn} & \longrightarrow & \Sigma^{syn} & & \\ & \searrow \mu & & \swarrow & \\ & & L^{mod} & \longrightarrow & \Sigma^{mod} \end{array}$$

Technically,  $\Sigma^{pf}$  and  $\Sigma^{mod}$  are obtained as pushouts, a concept from category theory. We refer to [Lan98] for the basics of category theory and only remark that the category of LF-signatures has pushouts along inclusions ([HST94]). Intuitively,  $\Sigma^{mod}$  is obtained as follows: Take  $L^{mod}$  and add to it the translation along  $\mu$  of all those declarations in  $\Sigma^{syn}$  that are not in  $L^{syn}$ . In other words,  $\Sigma^{mod}$  is the union of  $L^{mod}$  and  $\Sigma^{syn}$  sharing  $L^{syn}$ .  $\Sigma^{pf}$  is obtained accordingly.

Then  $\Sigma$ -sentences  $F$  are represented as  $\beta$ - $\eta$ -normal LF-terms of type  $o$  over the signature  $\Sigma^{syn}$ . We will write  $\lceil F \rceil$  for the LF-term representing the sentence  $F$ . An encoding is adequate for the syntax if this representation is a bijection.

Similarly,  $\Sigma$ -proofs of  $F$  using assumptions  $F_1, \dots, F_n$  are represented as  $\beta$ - $\eta$ -normal LF-terms over  $\Sigma^{pf}$  of type  $\pi(ded F_1 \rightarrow \dots \rightarrow ded F_n \rightarrow ded F)$ . Again we write  $\lceil P \rceil$  for the encoding of the proof  $P$  and say that the encoding is adequate if it is a bijection.

We will elaborate on the representation of  $\Sigma$ -models and the truth in models throughout the text.

It is noteworthy how the framework takes a balanced position between proof and model theoretical perspectives on logic. In particular, the type  $ded F$  is used to represent truth both proof- and model-theoretically. Proof-theoretically, terms of type  $ded F$  represent proofs of  $F$ , model-theoretically  $ded F$  is a predicate on the truth value of  $F$ .

A particular feature of the framework is that soundness can be represented very naturally: A soundness proof of  $L$  is represented as a view from  $L^{pf}$  to  $L^{mod}$  that makes the resulting triangle commute. We will get back to that in Sect. 4.6.

This framework is closely related to the logical frameworks of institutions ([GB92]) and LF ([HHP93]). From the perspective of institutions, it can be seen as utilizing LF-signatures to obtain concrete, strongly typed syntax to define signatures and sentences. Similarly, LF-signature morphisms are used to describe models in a way similar to parchments ([GB86]). A difference is the inclusion of proof theory and the separation into signatures  $\Sigma^{syn}$ ,  $\Sigma^{pf}$ , and  $\Sigma^{mod}$ . Furthermore, the way LF is used to define logics and to do so modularly goes back to ideas from [HST94] and [Tar96].

From the perspective of LF, it adds signature morphisms as a means to reason about translations between signatures and logics in addition to the reasoning about logics and signatures possible with the existing logic representations. In this work, we give a concrete semantic domain, which permits to represent models in the framework as well.

#### 4. Representing First-Order Logic

As described in Sect. 3, the encoding of FOL in LF consists of signatures  $FOL^{syn}$  for the syntax,  $FOL^{pf}$  for the proof theory, and  $FOL^{mod}$  for the model theory, together with two views  $\pi : FOL^{syn} \rightarrow FOL^{pf}$  and  $\mu : FOL^{syn} \rightarrow FOL^{mod}$ . FOL signatures and theories will be encoded as extensions of  $FOL^{syn}$ .

We will describe  $FOL^{syn}$  in Sect. 4.1,  $FOL^{pf}$  and  $\pi$  in Sect. 4.2, and  $FOL^{mod}$  and  $\mu$  in Sect. 4.4.  $FOL^{mod}$  will include a meta-language in which the models are specified. In textbook style descriptions, this meta-language is usually natural language implicitly based on some set-theoretical foundation of mathematics. We have to formalize this meta-language and thus pick an intuitionistic logic on top of simple type theory, which we refer to as HOL. We define it in Sect. 4.3. Then we discuss the adequacy of our encoding in Sect. 4.5. Finally, we prove the soundness of FOL by giving a view from  $FOL^{pf}$  to  $FOL^{mod}$  in Sect. 4.6.

$$\begin{array}{ccc} & & FOL^{pf} \\ & \nearrow \pi & \\ FOL^{syn} & & \searrow \mu \\ & & FOL^{mod} \\ & & \uparrow \\ & & HOL \end{array}$$

When encoding signatures and theories in LF, we have the problem that definitions of FOL signatures usually permit arbitrary objects as symbol names. But LF and Twelf expressions have to be words over a countable alphabet. Therefore, we employ two restrictions that are somewhat severe theoretically but natural for applications in computer science. From now on, all FOL theories have a finite number of function and predicate symbols and axioms. It is straightforward to define encodings for infinite theories, but type-theoretical frameworks usually avoid reasoning about infinite signatures. Furthermore, all

function and predicate symbols are chosen from a fixed countable set, and without loss of generality, we assume this set to be the set of legal Twelf identifiers. Thus, we can use the same names in FOL signatures and their encodings.

#### 4.1. Syntax

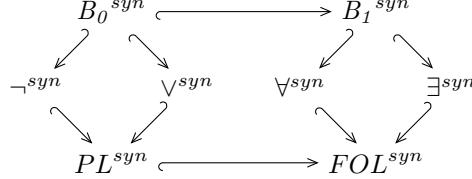


Figure 2: Modular Encoding of FOL Syntax

We encode the signature  $FOL^{syn}$  modularly, where each logical connective and quantifier is declared in a separate LF signature. The modular representation of  $FOL^{syn}$  is illustrated by the diagram in Fig. 2. Each node corresponds to a LF signature and each edge to an inclusion morphism.  $B_0^{syn}$  and  $B_1^{syn}$  are base signatures for propositional and first-order respectively, each connective or quantifier is encoded as an extension of the base signature, and then  $PL^{syn}$  for propositional logic and  $FOL^{syn}$  for first-order logic are encoded by including the needed fragments.

We only give some of the fragments as examples. The full encoding has one signature each for *true*, *false*,  $\neg$ ,  $\vee$ ,  $\wedge$ ,  $\Rightarrow$ ,  $\forall$ ,  $\exists$  and  $\doteq$ , and all of those are included into  $FOL^{syn}$ .

The LF signatures are given in Fig. 3. In the signature  $B_0^{syn}$  we introduce the type  $o$  for formulas and a type family  $ded : o \rightarrow \text{type}$  for the truth judgment on formulas. In the signatures  $\neg^{syn}$  and  $\vee^{syn}$  we introduce  $\neg$  and  $\vee$  respectively. Both  $\neg^{syn}$  and  $\vee^{syn}$  inherit the symbols  $o$  and  $ded$  by including  $B_0^{syn}$ . We merge them to form the signature  $PL^{syn}$ . The signature  $B_1^{syn}$  extends  $B_0^{syn}$  with a type  $i$  for terms. We introduce the universal and existential quantifiers in the signatures  $\forall^{syn}$  and  $\exists^{syn}$ , respectively. Finally, we define  $FOL^{syn}$  by including the signatures  $PL^{syn}$ ,  $\forall^{syn}$  and  $\exists^{syn}$ .

We can now encode FOL-signatures as LF-signatures that extend  $FOL^{syn}$ . The distinction between signatures and theories is not important from the perspective of LF as the encoding of axioms is very similar to the encoding of function and predicate symbols. Furthermore, we can always consider signatures as the special case of theories without axioms. Therefore, we will unify them and use  $\Sigma$  for both signatures and theories.

**Definition 17** (Encoding Syntax). Let  $\Sigma$  be a FOL-signature or theory. We define the LF-encoding  $\Sigma^{syn}$  of  $\Sigma$  as the LF-signature that includes  $FOL^{syn}$  and adds the following symbol declarations:

- $f : \underbrace{i \rightarrow \dots \rightarrow i}_n \rightarrow i$  for function symbols  $f$  of  $\Sigma$  with  $ar(f) = n$ ,

```

%sig  $B_0^{syn} = \{$ 
  o   : type
  ded :  $o \rightarrow type$ 
}
%sig  $\neg^{syn} = \{$ 
  %include  $B_0^{syn}$ 
   $\neg : o \rightarrow o$ 
}
%sig  $\vee^{syn} = \{$ 
  %include  $B_0^{syn}$ 
   $\vee : o \rightarrow o \rightarrow o$ 
}
%sig  $PL^{syn} = \{$ 
  %include  $\neg^{syn}$ 
  %include  $\vee^{syn}$ 
}
%sig  $B_1^{syn} = \{$ 
  %include  $B_0^{syn}$ 
  i   : type
}
%sig  $\forall^{syn} = \{$ 
  %include  $B_1^{syn}$ 
   $\forall : (i \rightarrow o) \rightarrow o$ 
}
%sig  $\exists^{syn} = \{$ 
  %include  $B_1^{syn}$ 
   $\exists : (i \rightarrow o) \rightarrow o$ 
}
%sig  $FOL^{syn} = \{$ 
  %include  $PL^{syn}$ 
  %include  $\forall^{syn}$ 
  %include  $\exists^{syn}$ 
}

```

Figure 3: LF Signatures for FOL Syntax

- $p : \underbrace{i \rightarrow \dots \rightarrow i}_n \rightarrow o$  for predicate symbols  $p$  of  $\Sigma$  with  $ar(p) = n$ ,
- $a : ded \lceil F \rceil$  for axioms  $F$  of  $\Sigma$  and some fresh name  $a$ .

Here  $\lceil F \rceil$  is the encoding of  $F \in Sen(\Sigma)$ . Every  $\Sigma$ -term  $t$  or formula  $F$  in context  $x_1, \dots, x_n$  is encoded as an LF term  $\lceil t \rceil : i$  or  $\lceil F \rceil : o$ , respectively, in context  $x_1 : i, \dots, x_n : i$ .  $\lceil t \rceil$  and  $\lceil F \rceil$  are defined by an obvious induction, and we only give the case of quantifiers as an example:

$$\lceil \forall x F \rceil = \forall [x : i] \lceil F \rceil.$$

**Definition 18** (Encoding Signature Morphisms). Let  $\sigma : \Sigma \rightarrow \Sigma'$  be a FOL signature morphism. Its LF-encoding is the LF signature morphism  $\sigma^{syn} : \Sigma^{syn} \rightarrow \Sigma'^{syn}$  that maps all symbols of  $FOL^{syn}$  to themselves and every function or predicate symbol  $s$  of  $\Sigma$  to  $\sigma(s)$ .

*Example 19.*  $Monoid^{syn}$  is the encoding of the theory  $Monoid$  from Ex. 5. For example, the binary function symbol  $\circ$  in  $MonSig$  is encoded as the symbol  $\circ : i \rightarrow i \rightarrow i$  that takes two arguments of LF-type  $i$  and returns an LF-term of type  $i$ .  $Group^{syn}$  is defined accordingly.

```
%sig Monoidsyn = {
  %include FOLsyn
  o      : i → i → i
  e      : i
  assoc  : ded ∀[x]∀[y]∀[z] x o (y o z) ≈ (x o y) o z
  neutl : ded ∀[x](e o x) ≈ x
  neutr : ded ∀[x](x o e) ≈ x
}
%infix o
```

#### 4.2. Proof Theory

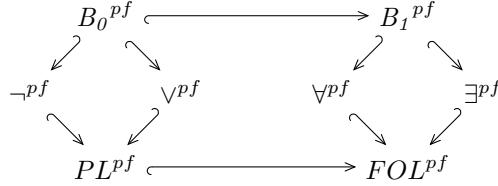


Figure 4: Modular Encoding of FOL Proof Theory

The encoding of the FOL proof theory has the same modular structure as the encoding of the FOL syntax. This is illustrated in Fig. 4 where each node has one additional – not displayed – inclusion from its counterpart in Fig. 2.

The signatures  $B_0^{pf}$  and  $B_1^{pf}$  typically contain the structural rules of the chosen calculus. In our case, they are equal to the signature  $B_0^{syn}$  and  $B_1^{syn}$  because encodings of natural deduction calculi in LF automatically inherit weakening, exchange, and contraction rules from LF. We distinguish these signatures anyway because of the conceptual clarity, and because the analogues of  $B_0^{pf}$  or  $B_1^{pf}$  in the encodings of other logics do contain additional declarations, e.g., when using sequent or tableaux calculi.

```
%sig B_0pf = {
  %include B_0syn
}
%sig B_1pf = {
  %include B_0pf
  %include B_1syn
}
```

The signatures  $\neg^{pf}$ ,  $\forall^{pf}$ , etc. encode the introduction and elimination rules for the individual connectives. We refer to [HHP93] for details about the encoding of proof rules and only give disjunction as an example.

```
%sig ∨pf = {
  %include B₀pf
  %include ∨syn
  orIl : ded F → ded F ∨ G
  orIr : ded G → ded F ∨ G
  orE : ded F ∨ G → (ded F → ded H) → (ded G → ded H)
    → ded H
}
```

Finally, the signatures  $PL^{pf}$  and  $FOL^{pf}$  collect the fragments in almost the same way as for the syntax encodings. The only difference in  $PL^{pf}$  is that the law of excluded middle is added:

```
%sig PLpf = {
  %include ¬pf
  %include ∨pf
  tnd : ded F ∨ ¬F
}
```

Similarly, there is one further proof rule added to  $FOL^{pf}$ : the axiom  $ded \exists[x] true$ . This axiom may be surprising because it is redundant in usual axiomatizations of FOL. This redundancy is due to the  $\forall$  elimination rule, which can instantiate  $\forall[x] true$  with any term including fresh variables. This amounts to assuming non-emptiness of the universe. In LF, only terms that are well-formed in the current context are eligible so that we obtain a system that is complete for a variant of FOL where the universe may be empty. Therefore, we explicitly add an axiom to make it non-empty.

**Definition 20.** For every FOL-signature or theory  $\Sigma$ , the LF signature  $\Sigma^{pf}$  is the canonical pushout of the diagram below.

$$\begin{array}{ccc} FOL^{pf} & \longrightarrow & \Sigma^{pf} \\ \pi \nearrow & & \pi_\Sigma \nearrow \\ FOL^{syn} & \longrightarrow & \Sigma^{syn} \end{array}$$

Here by canonical we mean that  $\Sigma^{pf}$  includes  $FOL^{pf}$  and adds a declaration  $c : \pi_\Sigma(A)$  for every declaration  $c : A$  that  $\Sigma^{syn}$  adds to  $FOL^{syn}$ .  $\pi_\Sigma$  maps  $FOL^{syn}$ -symbols according to  $\pi$  and other symbols to themselves.

Note that in the special case of FOL, both  $\pi$  and  $\pi_\Sigma$  are inclusions.

We have the Curry-Howard representation of proofs as terms:

**Definition 21** (Encoding Proofs). For a FOL-theory  $\Sigma$ , every derivation  $p$  of  $F_1, \dots, F_n \vdash_\Sigma F$  is encoded as an LF term  $\lceil p \rceil : ded \lceil F_1 \rceil \rightarrow \dots \rightarrow ded \lceil F_n \rceil \rightarrow ded \lceil F \rceil$  over  $\Sigma^{pf}$  by a straightforward induction.

#### 4.3. A Meta-Language for the Representation of Model Theory

*Foundations.* As all formal representations, the representation of the model theory requires a suitable meta-language. For the syntax and proof theory, LF is a very appropriate meta-language – this is not surprising because it is what LF was designed to be. LF only offers minimal syntactic means: judgments as types with implication (via  $\rightarrow$ ) and universal quantification (via  $\Pi$ ), and the former is just a special case of the latter. Still, it has been quite successful at covering a large class of logics because the syntax and proof theory of a logic are often (and in fact should usually be – see [ML96]) defined in terms of a grammar, judgments about its expressions, and an inference system for these judgments. LF can be seen as the result of applying Occam’s razor to these requirements.

The situation is different for the representation of model theory. Models are described in the language of mathematics, and it is difficult to formalize this language without making a foundational commitment. For example, we could choose a variant of set theory (as in [PC93, TB85]), higher-order logic ([Chu40], as in [Gor88, NPW02, Har96]), Martin-Löf type theory ([ML74], as in [Nor05]), or the calculus of constructions ([CH88], as in [BC04, ACTZ06]), all of which provide implementations with strong computational support. LF, on the other hand, is too weak to be such a foundation because the type theory is minimalist and the use of higher-order abstract syntax is incompatible with the natural way of adding computational power.

However, because of this weakness, LF can serve as a minimal, neutral framework in which to formalize the foundation itself. Moreover, since the choice of foundation changes the notion of model (and thus possibly the truth of a statement about models), an encoding can only be adequate relative to a fixed foundation. For example, consider first-order logic with models taken in a set theory with large cardinals. Therefore, it is even desirable to make the foundation part of the encoding. This also has the benefit that it becomes possible to build foundations modularly and to compare and translate between them. For example, we have formalized Mizar and Isabelle/HOL along with translations into ZFC set theory [KMR09].

*Approximating Foundations.* In this paper, we choose ZFC set theory as the foundation because it is the standard foundation of mathematics. Therefore, we encode ZFC in LF and use it as the meta-language to define models. However, ZFC behaves badly computationally because it is engineered towards elegance and simplicity rather than decidability or efficiency. Therefore, we also use a second meta-language – a variant of higher-order logic (HOL) – which can be worked with efficiently.

The intuition is that ZFC gives the official definition, and HOL is a sound but incomplete approximation of ZFC. Using the module system, we can state the relation between HOL and ZFC precisely by giving an LF signature morphism  $\varphi$  from HOL to ZFC. Via  $\varphi$ , we can regard ZFC as a refinement of HOL or HOL as a fragment of ZFC. Of course, HOL is not complete (relative to standard models in ZFC) and thus does not necessarily yield adequate encodings of model

theory. However, for certain results, working with HOL is sufficient, and then it is preferable.

Moreover, we can construct a chain of foundations of increasing strength, e.g.,  $\text{HOL} \rightarrow \text{ZF} \rightarrow \text{ZFC}$ , and always work in the weakest possible foundation. This is in keeping with mathematical practice not to commit to a specific foundation unless necessary, and leads to an approach we call *little foundations* (inspired by [FGT92]). For example, our encodings in [KMR09], avoid the use of excluded middle and the axiom of choice whenever possible.

In this section, we will only consider HOL, which we introduce below and use as the meta-language to represent models in Sect. 4.4. In Sect. 5, we encode ZFC set theory in LF, refine our HOL-based semantics into a ZFC-based one, and revisit the encoding of model theory.

*HOL as a Meta-Language.* For the representation of first-order logic, we need the booleans  $\text{bool}$ , an arbitrary set  $\text{univ}$  (the universe), and functions between the universe and the booleans. Among the latter are functions from  $\text{univ}^n$  to  $\text{univ}$ , from  $\text{univ}^n$  to  $\text{bool}$ , and from  $\text{bool}$  and  $\text{bool}^2$  to  $\text{bool}$  interpreting the function symbols, predicate symbols, and the propositional connectives, respectively. Furthermore, the quantifiers must be interpreted as second-order functions from  $\text{bool}^{\text{univ}}$  to  $\text{bool}$ . Finally, these functions should be typed, and that leads us to the choice of HOL as the meta-language.

```
%sig HOL = {
  set      : type
  ==>     : set → set → set
  elem    : set → type
  λ       : (elem A → elem B) → elem (A ==> B)
  @       : elem (A ==> B) → elem A → elem B
  prop    : type
  True   : prop → type
  true   : prop
  false  : prop
  ∧      : prop → prop → prop
  ⇒      : prop → prop → prop
  ∨      : prop → prop → prop
  ⇔      : prop → prop → prop
  ≡      : elem A → elem A → prop
  ∀      : (elem A → prop) → prop
  ∃      : (elem A → prop) → prop
}
```

Figure 5: Encoding of HOL

To define HOL, we encode it as the LF signature given in Fig. 5. Actually, we

only give a partial signature here and omit all the proof rules. The full version of the encoding of HOL can be found at [KMR09]. Note that this signature extends the running example of Sect. 2.2 except that we write *set* and *elem* instead of *tp* and *tm* to emphasize the relation to set theory. *set* is the type of sets, and  $\Rightarrow$  gives the set of functions between two sets. *elem A* is the type of elements of set *A* – this lets us reason about the elements of a set without using the  $\in$  relation. HOL must be a sound but not necessarily a complete fragment of set theory: Thus, the relation  $a : \text{elem } A$  must imply  $a \in A$ , but the inverse does not have to hold. Then  $\lambda$  and @ encode function formation and application. This yields the standard encoding of simple type theory in LF.

Finally, *prop* is the type of propositions. The propositional connectives are declared in the usual way. Equality and the quantifiers take an implicit argument *A* for the set which they operate on. Note that this means that we use equality only between elements of the same set and only use bound quantifiers. We omit the proof rules for HOL here and only state that we use rules for  $\beta$  and  $\eta$  conversion, and natural deduction introduction and elimination rules for the connectives and quantifiers. Equality is axiomatized as a congruence relation on each type. We do not assume the axiom of excluded middle – this turns out to suffice to axiomatize FOL models and makes us more flexible because we are not a priori committed to classical foundations of mathematics.

It is interesting to note that the actual encoding in Twelf is a little different because it already benefits from the LF module system: Our logic library represents the syntax and proof theory of the propositional connectives once and for all, and they are imported both into the object logic FOL and into the meta-language HOL.

The use of HOL as the meta-language means that the model theory of the object language (e.g., FOL) is represented as an extension of the signature *HOL*, i.e., a HOL-theory. Therefore, we define:

**Definition 22** (HOL-theories). A theory of HOL is an LF signature that includes the signature *HOL* and that only adds declarations of the following forms:

- base sets:  $S : \text{set}$ ,
- constant symbols:  $c : \text{elem } S$  for some set *S*,
- axioms:  $a : \text{True } F$  for some proposition *F*.

While the model theory of the object language is represented as a HOL-theory, every individual model is represented as a HOL model. Therefore, we have to define HOL models as well. In Sect. 5 we will show how models can be encoded as syntactical entities of LF, but here we will do something simpler and define (standard) HOL models as platonic objects in an underlying set theoretical universe:

**Definition 23** (HOL-Models). Assume a fixed set-theoretical universe. A model *M* of an HOL-theory *T* is a mapping that assigns:

- to every base set  $S : \text{set}$  a set  $S^M$ ,
- to every constant symbol  $c : \text{elem } S$  an element  $c^M \in S^M$ ,

such that  $F^M$  is true for every axiom  $a : \text{True } F$ . Here  $S^M$  is obtained by recursively replacing  $(S_1 \Rightarrow S_2)^M$  with the set of functions from  $S_1^M$  to  $S_2^M$ . And  $F^M$  is obtained by replacing every base set  $S$  or constant  $c$  occurring in  $F$  with  $S^M$  or  $c^M$ , respectively, and evaluating the result as a proposition over the set theoretical universe.

Def. [23] is somewhat vague. For sake of definiteness, we can assume ZFC as the underlying set theory:

*Example 24* (HOL-Models in ZFC). If our underlying set theory is ZFC, we can define  $F^M$  as follows:

1. replace all  $S$  and  $c$  with  $S^M$  and  $c^M$ , respectively,
2. relativize all quantifiers, i.e.,  $\forall [x : \text{elem } S] F$  becomes  $\forall x \in S^M. F^M$ ,
3. replace  $S \Rightarrow S'$  with the set of functions from  $S^M$  to  $S'^M$ , and replace  $\lambda$  and  $@$  with function formation and function application.

Then we are ready to represent the model theory of object logics in HOL in Sect. [4.4].

#### 4.4. Model Theory

In this section we present our representation of FOL model theory in LF. Our encoding of FOL model theory consists two parts as illustrated in the diagram below: The specification of FOL models (given by the signature  $FOL^{mod}$ ) and the interpretation of FOL syntax in terms of the semantics (given by the view  $\mu$  from  $FOL^{syn}$  to  $FOL^{mod}$ ).

$$FOL^{syn} \xrightarrow{\mu} FOL^{mod}$$

$\uparrow$

*HOL*

Both  $FOL^{mod}$  and  $\mu$  follow the same modular structure as in  $FOL^{syn}$  and  $FOL^{pf}$ , however, for the sake of simplicity, we will present the flat version of our encoding in this paper.

*Specification of FOL Models.* In  $FOL^{mod}$  we first encode the model theoretical notion of truth and the universe of a model. We declare a set  $\text{bool}$  which represents the set  $\{0, 1\}$  of truth values and axiomatize this by declaring 0 and 1, and axioms  $ax_{cons}$  and  $ax_{boole}$  to state that  $\text{bool}$  is indeed the desired 2-element set:

$$\begin{aligned} \text{bool} &: \text{set} \\ 0 &: \text{elem bool} \\ 1 &: \text{elem bool} \\ ax_{cons} &: \text{True } \neg(0 \doteq 1) \\ ax_{boole} &: \{F\} \text{ True } (F \doteq 0 \vee F \doteq 1) \end{aligned}$$

Recall that  $\text{True} : \text{prop} \rightarrow \text{type}$  is the truth judgment of the meta-language HOL, that  $\neg$  and  $\vee$  are the negation and disjunction on HOL propositions, and

$\doteq$  is the typed-equality of HOL terms. In particular, these symbols are different from the symbols of the same name in the syntax of FOL.

We declare the symbol *univ* as a set for the universe of a FOL-model, and add an axiom making *univ* non-empty.

$$\begin{aligned} \textit{univ} &: \textit{set} \\ \textit{nonemp} &: \textit{True } \exists [x : \textit{elem univ}] \textit{ true} \end{aligned}$$

Next we encode the semantics of the logical symbols introduced in  $FOL^{syn}$ . For each logical symbol  $s^{syn}$  in  $FOL^{syn}$ , we declare a symbol  $s^{mod}$ , which represents the semantic operation used to interpret  $s^{syn}$  along with axioms specifying its values. This corresponds to the case-based definition of the semantics of a formula.

As examples we present the cases for  $\vee$  and  $\forall$ . For the interpretation of  $\vee$ , we declare the symbol *or* as a HOL-function from *bool*<sup>2</sup> to *bool* and axiomatize it to be the binary supremum in the boolean 2-element lattice.

$$\begin{aligned} \textit{or} &: \textit{elem} (\textit{bool} \Rightarrow \textit{bool} \Rightarrow \textit{bool}) \\ \textit{or1} &: \textit{True } ((F \doteq 1 \vee G \doteq 1) \Rightarrow (\textit{or} @ F @ G) \doteq 1) \\ \textit{or0} &: \textit{True } ((F \doteq 0 \wedge G \doteq 0) \Rightarrow (\textit{or} @ F @ G) \doteq 0) \end{aligned}$$

Similarly, for the interpretation of  $\forall$ , we specify the function *forall* that takes a *univ*-indexed family *F* of booleans and returns its infimum, i.e., it returns 1 iff all *F@x* is 1 for all *x*.

$$\begin{aligned} \textit{forall} &: \textit{elem} ((\textit{univ} \Rightarrow \textit{bool}) \Rightarrow \textit{bool}) \\ \textit{forall1} &: \{F : \textit{elem} (\textit{univ} \Rightarrow \textit{bool})\} \\ &\quad \textit{True } (\forall [x : \textit{elem univ}] (F @ x) \doteq 1 \Rightarrow (\textit{forall} @ F) \doteq 1) \\ \textit{forall0} &: \{F : \textit{elem} (\textit{univ} \Rightarrow \textit{bool})\} \\ &\quad \textit{True } (\exists [x : \textit{elem univ}] (F @ x) \doteq 0 \Rightarrow (\textit{forall} @ F) \doteq 0) \end{aligned}$$

An overview of the operations and axioms declared for the remaining connectives and quantifiers is given in Fig. 6. In all cases except for equality, we have two axioms of the form  $C_0 \Rightarrow F \doteq 0$  and  $C_1 \Rightarrow F \doteq 1$ , e.g., *and0* for when a conjunction is false and *and1* for when it is true. For equality, our results below require a slightly stronger condition, namely the axiom *eq1* of the form  $C_1 \Leftrightarrow F \doteq 1$  (from which the corresponding  $\neg C_1 \Leftrightarrow F \doteq 0$  can be derived).

*Interpretation Function.* The idea of the view  $\mu$  is that it maps from the syntax to the semantics; it gives the cases of the interpretation function for the logical symbols. This takes the form of a view from  $FOL^{syn}$  to  $FOL^{mod}$ , which must give a  $FOL^{mod}$ -expression for all symbols declared in or included into  $FOL^{syn}$ .

Formulas are interpreted as boolean truth values, and we map the type *o* of formulas to the type *elem bool* of truth values. The truth value 1 is designated, i.e., represents truth. Therefore, we map *ded* to a type family that takes an argument *F* of type *elem bool* and returns the judgment that *F* is equal to 1. FOL-terms are interpreted as elements of the universe. Therefore, we map the type *i* of FOL-terms to the type *elem univ* of elements of *univ*.

|                |   |
|----------------|---|
| <i>true</i>    | $: elem\;bool$  |
| <i>true1</i>   | $: True\;true \doteq 1$   |
| <i>false</i>   | $: elem\;bool$  |
| <i>false0</i>  | $: True\;false \doteq 0$  |
| <i>not</i>     | $: elem(bool \Rightarrow bool)$   |
| <i>not0</i>    | $: True(F \doteq 0 \Rightarrow (not @ F) \doteq 1)$   |
| <i>not1</i>    | $: True(F \doteq 1 \Rightarrow (not @ F) \doteq 0)$   |
| <i>and</i>     | $: elem(bool \Rightarrow bool \Rightarrow bool)$  |
| <i>and1</i>    | $: True(F \doteq 1 \wedge G \doteq 1 \Rightarrow (and @ F @ G) \doteq 1)$                                       |
| <i>and0</i>    | $: True(F \doteq 0 \vee G \doteq 0 \Rightarrow (and @ F @ G) \doteq 0)$   |
| <i>impl</i>    | $: elem(bool \Rightarrow bool \Rightarrow bool)$  |
| <i>impl1</i>   | $: True(F \doteq 0 \vee G \doteq 1 \Rightarrow (impl @ F @ G) \doteq 1)$  |
| <i>impl0</i>   | $: True(F \doteq 1 \wedge G \doteq 0 \Rightarrow (impl @ F @ G) \doteq 0)$                                      |
| <i>exists</i>  | $: elem((univ \Rightarrow bool) \Rightarrow bool)$  |
| <i>exists1</i> | $: \{F : elem(univ \Rightarrow bool)\}$<br>$True(\exists[x](F @ x) \doteq 1 \Rightarrow (exists @ F) \doteq 1)$ |
| <i>exists0</i> | $: \{F : elem(univ \Rightarrow bool)\}$<br>$True(\forall[x](F @ x) \doteq 0 \Rightarrow (exists @ F) \doteq 0)$ |
| <i>eq</i>      | $: elem(univ \Rightarrow univ \Rightarrow bool)$  |
| <i>eq1</i>     | $: True(F \doteq G \Leftrightarrow (eq @ F @ G) \doteq 1)$  |

Figure 6: Specification of FOL Models

$$\begin{aligned} o &:= elem\;bool \\ ded &:= [F : elem\;bool] True(F \doteq 1) \\ i &:= elem\;univ \end{aligned}$$

The interpretation of the logical connectives is as expected. For example, the disjunction  $F \vee G$  is interpreted by applying *or* to  $\mu(F)$  and  $\mu(G)$ . And the universal quantification  $\forall ([x : i] F)$  is interpreted as  $\mu(\forall) \mu([x : i] F) = \text{forall} @ \lambda ([x : elem\;univ] \mu(F))$ .

$$\begin{aligned} \vee &:= [F : elem\;bool] [G : elem\;bool] \text{or} @ F @ G \\ \forall &:= [F : elem(univ \Rightarrow bool)] \text{forall} @ (\lambda F) \end{aligned}$$

*Models of FOL-Theories.* Finally we can define  $\Sigma^{mod}$  just like  $\Sigma^{pf}$ :

**Definition 25.** For every FOL-signature or theory  $\Sigma$ , the LF signature  $\Sigma^{mod}$  is the canonical pushout in the diagram below.

$$\begin{array}{ccc} FOL^{mod} & \hookrightarrow & \Sigma^{mod} \\ \mu \nearrow & & \mu_\Sigma \nearrow \\ FOL^{syn} & \longrightarrow & \Sigma^{syn} \end{array}$$

Here by canonical we mean that  $\Sigma^{mod}$  includes  $FOL^{mod}$  and adds a declaration  $c : \mu_\Sigma(A)$  for every declaration  $c : A$  that  $\Sigma^{syn}$  adds to  $FOL^{syn}$ .  $\mu_\Sigma$  maps  $FOL^{syn}$ -symbols according to  $\mu$  and other symbols to themselves.

*Example 26* (Continued). The signature  $\text{Group}^{\text{mod}}$  looks as follows:

```
%sig Groupmod = {
  %include FOLmod
  o      : elem univ → elem univ → elem univ
  e      : elem univ
  inv   : elem univ → elem univ
  assoc : True (forall @ (λ[x] forall @ (λ[y] forall @ (λ[z]
    eq @ (x o (y o z)) @ ((x o y) o z)))) ≡ 1
  :
}
```

*Notation 27.* Note that the canonical pushout yields declarations  $\circ : \text{elem univ} \rightarrow \text{elem univ} \rightarrow \text{elem univ}$  rather than  $\circ : \text{elem}(\text{univ} \Rightarrow \text{univ} \Rightarrow \text{univ})$ . Thus,  $\text{Group}^{\text{mod}}$  is technically not a HOL-theory in the sense of Def. 22. However, we can give signature morphisms back and forth between these. For example, if we have a signature with  $\circ' : \text{elem}(\text{univ} \Rightarrow \text{univ} \Rightarrow \text{univ})$ , the morphisms map  $\circ'$  to  $\lambda[x] \lambda[y] x o y$  and  $\circ$  to  $[x] [y] \circ' @ x @ y$ . In the following, we will assume that  $\Sigma^{\text{mod}}$  extends  $\text{FOL}^{\text{mod}}$  with declarations of the form  $c : \text{elem } S_1 \Rightarrow \dots \Rightarrow S_n$  and omit the connecting signature morphisms from the notation.

**Definition 28** (Encoding Models). Assume a fixed set theory in which FOL and HOL-models are defined, and pick two arbitrary sets  $\mathcal{F}$  and  $\mathcal{T}$  as the truth values. Then for every FOL-signature  $\Sigma$  and every  $\Sigma$ -model  $M = (U, I)$ , we define a HOL-model  $\lceil M \rceil$  of  $\Sigma^{\text{mod}}$  as follows:

$$0^{\lceil M \rceil} = \mathcal{F} \quad 1^{\lceil M \rceil} = \mathcal{T} \quad \text{bool}^{\lceil M \rceil} = \{\mathcal{F}, \mathcal{T}\} \quad \text{univ}^{\lceil M \rceil} = U,$$

$\text{not}^{\lceil M \rceil}$ ,  $\text{and}^{\lceil M \rceil}$ ,  $\text{or}^{\lceil M \rceil}$ ,  $\text{impl}^{\lceil M \rceil}$ ,  $\text{eq}^{\lceil M \rceil}$ ,  $\text{forall}^{\lceil M \rceil}$ , and  $\text{exists}^{\lceil M \rceil}$  are defined in the obvious way, and for function symbols  $f$  and predicate symbols  $p$  we put

$$f^{\lceil M \rceil} = f^I \quad \text{and} \quad p^{\lceil M \rceil}(u_1, \dots, u_n) = \mathcal{T} \text{ iff } (u_1, \dots, u_n) \in p^I.$$

It is easy to check that this is indeed a HOL-model of  $\Sigma^{\text{mod}}$ , i.e., satisfies all the axioms of  $\Sigma^{\text{mod}}$ .

We could also encode model translations, but we can do this more elegantly in Sect. 5.

*Example 29* (Continued). Consider the model  $\text{Int}$  from Ex. 15. It is encoded as a HOL-model of  $\text{Group}^{\text{mod}}$  by putting  $\text{univ}^{\lceil \text{Int} \rceil} = \mathbb{Z}$ ,  $\circ^{\lceil \text{Int} \rceil} = +$ ,  $e^{\lceil \text{Int} \rceil} = 0$ , and  $\text{inv}^{\lceil \text{Int} \rceil} = -$ . The interpretations of all other symbols are uniquely determined.

#### 4.5. Adequacy

In the previous sections, we have defined the LF signatures and morphisms  $(\text{FOL}^{\text{syn}}, \text{FOL}^{\text{pf}}, \pi, \text{FOL}^{\text{mod}}, \mu)$  intended to encode FOL. ( $\pi$  is simply an inclusion for FOL, but we will keep it in the notation to stress that FOL is only an example for a generic method.) Showing that such an encoding is adequate

means to show that the encoding has the same properties as the encoded logic. If an encoding is adequate, meta-logical results reached by reasoning about the logic encoding are guaranteed to hold for the encoded logic as well.

To give a general formal definition what adequacy means, we need to do three things: (i) define in general what a logic is and when two logics are isomorphic, (ii) define in general what a logic encoding in LF is and how every such logic encoding induces a logic, and then (iii) for a specific logic encoding show that the induced logic is isomorphic to the encoded logic. Especially (ii) requires a large amount of work, which we carried out in [Rab08]. For simplicity, here, we will only consider the special case of adequacy of our encoding of FOL. For other logic encodings, the procedure is the same.

We begin by restating some known results for the adequacy of syntax and proof theory in our notation. See e.g. [HHP93, HST94], the encodings used there are not modular, but that is a minor difference that does not affect the proofs.

**Theorem 30** (Adequacy for Syntax). *For every FOL-signature  $\Sigma$  and context  $\Gamma = x_1, \dots, x_n$ , the formulas are in a natural bijection with the  $\beta\eta$ -normal LF-terms of type  $o$  over  $\Sigma^{syn}$  in context  $x_1 : i, \dots, x_n : i$ .*

**Theorem 31** (Adequacy for Signature Morphisms). *For every FOL-signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$  and every sentence  $F \in \text{Sen}(\Sigma)$ , we have  $\lceil \sigma(F) \rceil = \sigma^{syn}(\lceil F \rceil)$ .*

**Theorem 32** (Adequacy for Proof Theory). *For every FOL-theory  $\Sigma$ , the derivations of  $F_1, \dots, F_n \vdash_{\Sigma} F$  are in a natural bijection with the  $\beta\eta$ -normal LF-terms of type  $\pi_{\Sigma}(\text{ded } \lceil F_1 \rceil \rightarrow \dots \rightarrow \text{ded } \lceil F_n \rceil \rightarrow \text{ded } \lceil F \rceil)$  over  $\Sigma^{pf}$ .*

*In particular,  $F$  is a  $\Sigma$ -theorem iff  $\pi_{\Sigma}(\text{ded } \lceil F \rceil)$  is inhabited over  $\Sigma^{pf}$ .*

**Theorem 33** (Adequacy for Proof Theoretical Theory Morphisms). *For a FOL-signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$ , we have  $\sigma : (\Sigma, \Theta) \xrightarrow{P} (\Sigma', \Theta')$  iff  $\sigma^{pf} : \Sigma^{pf} \rightarrow \Sigma'^{pf}$  can be extended to an LF signature morphism  $(\Sigma, \Theta)^{pf} \rightarrow (\Sigma', \Theta')^{pf}$ .*

*Proof.* This follows from Thm. 32 by extending  $\sigma^{pf}$  such that every  $a : \pi_{\Sigma}(\text{ded } \lceil F \rceil)$  occurring in  $(\Sigma, \Theta)^{pf}$  is mapped to  $\lceil p \rceil$  for some proof  $p$  of  $\text{Sen}(\sigma)(F)$ .  $\square$

To give similar results for the model theory, we need a bijection between FOL-models of  $\Sigma$  and HOL-models of  $\Sigma^{mod}$ . First we prove that every single model of  $FOL^{mod}$  is adequate in the following sense:

**Lemma 34.** *Assume a HOL-model  $M$  of  $FOL^{mod}$ . Then  $\text{bool}^M = \{0^M, 1^M\}$  is a two-element set,  $\text{univ}^M$  is an arbitrary non-empty set, and  $\text{true}^M, \text{false}^M$ , and  $^M, \text{or}^M, \text{impl}^M, \text{eq}^M, \text{forall}^M$ , and  $\exists^M$  are the usual operations in the semantics of first-order logic with respect to the universe  $\text{univ}^M$  and the truth values  $0^M$  for falsity and  $1^M$  for truth.*

*Proof.* Straightforward using the axioms in  $FOL^{mod}$ .  $\square$

Note that the interpretations of the booleans (and thus of the remaining operations on them) are not determined uniquely because the specific choice of truth values remains free. If we wanted to characterize them uniquely, e.g.,  $0^M = \emptyset$  and  $1^M = \{\emptyset\}$ , we would need to have more access to the underlying set theory than provided by HOL. However, once two arbitrary truth values and the universe are fixed, the interpretation of the connectives and quantifiers is determined. Then we can state the adequacy of the model theory as follows:

**Theorem 35** (Adequacy for Model Theory). *Assume a fixed set theory in which FOL- and HOL-models are defined, and pick two arbitrary sets  $\mathcal{F}$  and  $\mathcal{T}$ . Let us call a  $FOL^{mod}$ -model normal if 0 and 1 are interpreted as  $\mathcal{F}$  and  $\mathcal{T}$ .*

*Then for every FOL-signature  $\Sigma$ , there is a natural bijection between FOL-models  $M$  of  $\Sigma$  and normal HOL models of  $\Sigma^{mod}$ . Furthermore, for every  $M$  and every  $\Sigma$ -sentence  $F$ , we have*

$$M \models_{\Sigma} F \quad \text{iff} \quad (\mu_{\Sigma}(ded \ \lceil F \rceil))^{\lceil M \rceil}.$$

Recall that for FOL, we have  $\mu_{\Sigma}(ded \ \lceil F \rceil) = True$  ( $\mu_{\Sigma}(\lceil F \rceil) \doteq 1$ ).

*Proof.* One direction of the bijection is the encoding  $M \mapsto \lceil M \rceil$ . For the inverse direction, assume a normal HOL-model  $M$  of  $\Sigma^{mod}$ . Because  $M$  is normal and because of Lem. 34,  $M$  has no freedom but to pick a non-empty set for  $univ^M$  and operations for  $f^M$  and  $p^M$ . It is easy to see that each such choice yields a FOL-model of  $\Sigma$ , and that the two functions are bijections.

The second claim follows by a straightforward induction on  $F$  using the axioms of  $FOL^{mod}$ .  $\square$

It is tempting to assume that, parallel to Thm. 33,  $\sigma : (\Sigma, \Theta) \xrightarrow{M} (\Sigma', \Theta')$  can be encoded using LF-signature morphisms  $(\Sigma, \Theta)^{mod} \rightarrow (\Sigma', \Theta')^{mod}$ . But this is not the case: The existence of such morphisms is only sufficient but not necessary for  $\sigma$  to be a model-theoretical theory morphism. This is because HOL is not complete with respect to standard models. We will get back to this in Sect. 5.

Taking all these adequacy results together, we see that all results about FOL that can be stated in terms of encodings of syntax, proof theory, and model theory, carry over to FOL. As an example, let us consider soundness and completeness.

**Theorem 36.** *For a FOL-signature  $\Sigma$  and LF terms  $F_i, F$ , define*

- (i) *There is a term of type  $\pi_{\Sigma}(ded \ F_1 \rightarrow \dots \rightarrow ded \ F_n \rightarrow ded \ F)$  over  $\Sigma^{pf}$ .*
- (ii) *For every HOL-model  $M$  of  $\Sigma^{mod}$ , if  $(\mu_{\Sigma}(F_i))^M$  for all  $i$ , then also  $(\mu_{\Sigma}(F))^M$ .*

*Then the logic FOL is sound iff (i) implies (ii), and complete iff (ii) implies (i).*

*Proof.* Immediately using the adequacy of proof and model theory.  $\square$

Among all the meta-logical properties that can be studied after encoding a logic in LF, soundness is particularly interesting because we have the following result:

**Theorem 37.** *If there is a signature morphism  $\sigma$  from  $FOL^{pf}$  to  $FOL^{mod}$  such that  $\pi \sigma = \mu$ , then the logic  $FOL$  is sound.*

*Proof.* We proceed in two steps. First, we show that for every FOL-signature  $\Sigma$  there is a signature morphism  $\sigma_\Sigma$  from  $\Sigma^{pf}$  to  $\Sigma^{mod}$  such that the following diagram commutes:

$$\begin{array}{ccccc}
 & & FOL^{pf} & & \Sigma^{pf} \\
 \pi \nearrow & \downarrow \sigma & & \searrow \pi_\Sigma & \downarrow \sigma_\Sigma \\
 FOL^{syn} & \xrightarrow{\quad} & \Sigma^{syn} & \xrightarrow{\quad} & \Sigma^{mod} \\
 \mu \searrow & \downarrow & \searrow \mu_\Sigma & & \downarrow \\
 & & FOL^{mod} & \xrightarrow{\quad} & \Sigma^{mod}
\end{array}$$

$\sigma_\Sigma$  is simply the universal morphism factoring  $\sigma$  and  $\mu_\Sigma$  through the pushout  $\Sigma^{pf}$ .

Secondly, we show soundness using Thm. 36. So assume (i). Since signature morphisms are type-preserving mappings, there must be a term of type  $\sigma_\Sigma(\pi_\Sigma(ded\ F_1 \rightarrow \dots \rightarrow ded\ F_n \rightarrow ded\ F))$  over  $\Sigma^{mod}$ . Because  $\pi_\Sigma \sigma_\Sigma = \mu_\Sigma$ , this type is equal to  $\mu_\Sigma(ded\ F_1) \rightarrow \dots \rightarrow \mu_\Sigma(ded\ F_n) \rightarrow \mu_\Sigma(ded\ F)$ . Now the implication introduction rule of HOL shows that (ii) holds.  $\square$

These results may be criticized as being implications between statements known to be true. But recall that the same methodology can be applied to a very wide variety of other logics, and we obtain the corresponding result for every such logic.

In general, Thm. 37 is only a sufficient criterion. It cannot be necessary for all logic encodings because HOL is only a (sound but incomplete) fragment of set theory, which may or may not be strong enough to carry out the soundness proof for the encoded logic. However, in our experience such a morphism typically exists for reasonable choices of the meta-language.

It is tempting to look for the analogue of Thm. 37 for *completeness*. But a morphism  $\Sigma^{mod} \rightarrow \Sigma^{pf}$  can usually not be given. For example,  $FOL^{mod}$  is significantly more expressive than  $FOL^{pf}$  and therefore cannot be interpreted in  $FOL^{pf}$ . Even when such a morphism exists, it does not imply completeness. But it is promising to investigate other ways to encode completeness, which we leave to future work. For example, the central idea of many completeness proofs is to construct a canonical model whose objects are given by the syntax of the logic. Since LF provides an excellent way to talk about syntax, it is interesting to use LF to form a canonical model. If the syntax of LF is reflected into the representation of the model theory, it can yield a general way of formalizing completeness proofs.

#### 4.6. Soundness

Now we will apply Thm. [37] to encode the soundness proof of FOL in LF by giving a view  $\sigma_{FOL}$  from  $FOL^{pf}$  to  $FOL^{mod}$ . The structure of  $\sigma_{FOL}$  follows the modular structure of  $FOL^{pf}$ , i.e., the soundness is proved separately for every connective or quantifier. In particular,  $\sigma_{FOL}$  includes the view  $\mu$  for all symbols of  $FOL^{syn}$ . Thus, the LF module system guarantees the commutativity of the diagram on the right.

The remaining symbols of  $FOL^{pf}$  are those encoding proof rules. Each of those must be mapped to a proof term in  $FOL^{mod}$ . This is straightforward, and we only give one example case and refer [KMR09] for the remaining cases.

The proof rule  $orI_l : ded F \rightarrow ded F \vee G$  is included into  $FOL^{pf}$  from  $\vee^{pf}$ . It uses two implicit arguments  $F : o$  and  $G : o$  and must be mapped to a  $FOL^{mod}$ -term of type

$$\{F : elem\; bool\} \{G : elem\; bool\} \; True\; F \doteq 1 \rightarrow True\; (or @ F @ G) \doteq 1$$

Its map is given by

$$orI_l := [F : elem\; bool] [G : elem\; bool] [p : True\; F \doteq 1] \Rightarrow_E or1 (\vee_I p)$$

Here  $\Rightarrow_E$  is the modus ponens rule, and  $\vee_I$  is the left introduction rule of disjunction of the meta-language.  $\Rightarrow_E$  and  $\vee_I$  are among the proof rules declared in HOL, which we omitted in Sect. [4.3].

## 5. Representing Set-Theoretical Model Theory

The representation of models given in Sect. [4.4] uses HOL as a meta-language. HOL is seen as a fragment of the foundation of mathematics, and to work with HOL rather than, e.g., a set theory, has the advantage of being simpler while not committing to a specific foundation. But it also has a drawback: FOL-models are represented as HOL-models and thus as platonic entities that live outside the logical framework LF.

It would be more appealing if FOL-models could be represented as LF entities themselves. This is indeed possible without changing the principal features of our approach: All we have to do is to refine the meta-language HOL so much that it becomes set theory. The refinements can be represented elegantly as LF signature morphisms – in this case from the encoding of HOL to an encoding of set theory.

More generally, we obtain the diagram below. Here  $(L^{syn}, L^{pf}, \pi, L^{mod}, \mu)$  is a logic encoding as before. The foundation of mathematics is encoded as an LF signature  $F$ , and the model theory is defined in terms of a meta-language  $F_0$ , which is a fragment of  $F$ . A view  $\varphi : F_0 \rightarrow F$  encodes the refinement of  $F_0$  into  $F$ , or in other words,  $\varphi$  formalizes in what sense  $F_0$  is a fragment of  $F$ .

$$\begin{array}{ccc} & FOL^{pf} & \\ \pi \nearrow & & \downarrow \sigma_{FOL} \\ FOL^{syn} & & FOL^{mod} \\ \searrow \mu & & \end{array}$$

Finally, we want to give a view  $\mu'$  from  $L^{mod}$  to  $F$ , which translates the  $F_0$ -based encoding of model theory to  $F$ .  $\mu'$  must have some free parameters, and this can be expressed in LF by adding these free parameters to the codomain of the view.  $L^{mod+}$  is the extension of  $F$  with these parameters.

Then, for any choice of these parameters, the composition  $\mu \mu'$  translates the logical syntax into mathematics. In other words, we refine the logic encoding  $(L^{syn}, L^{pf}, \pi, L^{mod}, \mu)$  based on  $F_0$  to a logic encoding  $(L^{syn}, L^{pf}, \pi, L^{mod+}, \mu \mu')$  based on  $F$ .

$$\begin{array}{ccccc}
 & & L^{pf} & & \\
 & \swarrow \pi & & \searrow & \\
 L^{syn} & \xrightarrow{\mu} & L^{mod} & \xrightarrow{\mu'} & L^{mod+} \\
 & \downarrow & \uparrow & \downarrow & \uparrow \\
 F_0 & \xrightarrow{\varphi} & F & &
\end{array}$$

So far we have used first-order logic for  $L$  and higher-order logic for  $F_0$ . In the following, we will give  $F$ ,  $\mu'$ , and  $\varphi$ .  $F$  will be an encoding of Zermelo-Fraenkel set theory (ZFC, [Zer08, Fra22]) encoded as an LF signature  $ZFC$  developed in Sect. 5.1.  $\varphi$  will give the standard semantics of higher-order logic encoded as a view from  $HOL$  to  $ZFC$  developed in Sect. 5.2. Note that these steps are independent of the chosen logic  $L$  because higher-order logic is sufficient for the model theory of many logics (including sorted, simply-typed, modal, description, and intuitionistic logics). We will give  $L^{mod+}$  and the view  $\mu'$  from  $FOL^{mod}$  to  $ZFC$  in Sect. 5.3.  $L^{mod+}$  will arise by adding to  $ZFC$  a free parameter for the universe.

We use ZFC because it is the most widely used foundation of mathematics. Other set theories such as Von-Neumann-Bernays-Gödel ([vN25, Ber37, Göd40]) could be used equally well. Similarly, type theoretic foundations such as HOL ([Chu40]) or the Calculus of Constructions ([CH88]) would work in the same way. We will elaborate on that in Sect. 5.2.

The above diagram still leaves open how individual models can be represented in LF. We will look at that in Sect. 5.4 where we will form a signature  $\Sigma^{mod+}$ , which will be like  $\Sigma^{mod}$  but in terms of  $ZFC$  rather than  $HOL$ , and then represent  $\Sigma$ -models as LF signature morphisms from  $\Sigma^{mod+}$  to  $ZFC$ . Finally we look at the encoding of model theoretical theory morphisms, at which point all aspects of FOL are encoded in LF. But since the  $\Sigma$ -models form a proper class and the LF expressions are countable, this raises adequacy questions, which we discuss in Sect. 5.5.

### 5.1. Representing Set Theory

Now we will represent ZFC set theory in LF. This is a necessary condition for the comprehensive representation of model theory in a logical framework. But it requires a significant investment. Very advanced encodings of set theory have been established in Mizar ([TB85] using Tarski-Grothendieck set theory [Try89, Tar38]), and in Isabelle/ZF ([Pau94, PC93]) employing sophisticated machine support. In particular, these encodings use semi-automated reasoning support and high-level proof description languages such as Isar ([Nip02]) for Isabelle. Our encoding was designed from scratch using hand-written proof terms.

There are two reasons to forgo those sophisticated encodings in favor of LF. Firstly, LF is superior to Isabelle as a logical framework: The dependent type theory permits elegant encodings of logics, and the module system based on signature morphisms permits elegant encodings of translations. We appreciate these fundamental aspects even though Isabelle is vastly superior to LF in terms of automation and tool support. Mizar offers dependent types, but it is a standalone encoding of set theory not based on a logical framework so that it cannot encode other languages such as logics or alternative set theories.

Secondly, our encoding of set theory differs from the above two in two fundamental but non-trivial design aspects. In both cases, only the existence of dependent types makes our design choices possible. These two aspects are the choice of primitive symbols and the use of a type system, which we will detail in Sect. 5.1.2 and 5.1.3, respectively.

The whole encoding of set theory comprises over 1000 lines of Twelf declarations. Therefore, we only showcase the most important features and refer to [KMR09] for the full encoding. In the following we will first explain the logic we use for our set theory in Sect. 5.1.1, then use it to develop untyped set theory in Sect. 5.1.2, and then build a typed set theory on top of the untyped one in Sect. 5.1.3. Only the typed set theory will be strong enough to subsume the meta-language HOL we developed in Sect. 4.3. Finally, we define the 2-element Boolean lattice  $\mathbb{B}$  and its operations in Sect. 5.1.4. The Booleans are not needed to subsume HOL but are needed as the set of truth values when defining the semantics of FOL. All declarations together form the LF signature *ZFC*.

*Notation 38.* This section will require the reader to be very careful in separating levels as the LF encoding of ZFC contains three groups of connectives and quantifiers.

The first two groups are meta-level operations on propositions. They share the propositional connectives which are written normally, e.g.,  $\wedge$ . The first group consists of the symbols used in the (untyped) first-order logic underlying set theory; here equality and quantifiers will be written as  $\doteq^*$ ,  $\forall^*$ , and  $\exists^*$ . The second group consists of the symbols used in the typed set theory that we will develop on top of the untyped one; here equality and quantifiers will be written as  $\doteq$ ,  $\forall$ , and  $\exists$ .

Finally the third group consists of the object level operations on Booleans. These will be written as  $\wedge_*$ ,  $\forall_*$ ,  $\doteq_*$ , etc.

The notations are chosen such that the symbols  $\wedge$ ,  $\forall$ ,  $\doteq$ , etc. are the intended interpretations of their counterparts in HOL, and the symbols  $\wedge_*$ ,  $\forall_*$ ,  $\doteq_*$ , etc. are the intended interpretations of the symbols declared in  $FOL^{mod}$ .

### 5.1.1. Logical Language

We base ZFC on first-order logic with equality. To reason about truth, we use an intuitionistic natural deduction calculus with introduction and elimination rules. The main LF declarations encoding this logic are the following ones.

```
set      : type
prop    : type
True   : prop → type
```

*set* is the single sort of sets, *prop* is the type of propositions, and *True F* is the judgment for the truth of *F*.

We make two additions to the otherwise well-known syntax of first-order logic: sequential connectives and a description operator. Both arise naturally when encoding set theory as we will see below.

*Sequential connectives* mean that, e.g., in an implication  $F \Rightarrow G$ , *G* is only considered if *F* is true. This is very natural in mathematical practice – for example, mathematicians do not hesitate to write  $x \neq 0 \Rightarrow x/x = 1$  when / is only defined for non-zero dividers. This can be solved by using first-order logic with partial functions, but we hold that it is more elegant and closer to mathematics to use a sequential implication, i.e., the truth of *F* is assumed when considering *G*. Similarly, in a sequential conjunction  $F \wedge G$ , *F* is assumed true when considering *G*. We use sequential conjunction and implication; all other connectives are as usual.

Then the LF encoding contains the following declarations for propositions:

```
∧      : {F : prop} (True F → prop) → prop
⇒     : {F : prop} (True F → prop) → prop
¬     : prop → prop.
∨     : prop → prop → prop
↔     : prop → prop → prop
≡*   : set → set → prop
∀*   : (set → prop) → prop
∃*   : (set → prop) → prop
```

Thus  $\wedge$  and  $\Rightarrow$  are applied to two arguments, a formula *F* and another formula which is stated in a context where *F* is true. This is written as, e.g.,  $F \wedge [p] G p$  where *p* is a proof of *F* that may be used by *G*. We will use  $F \wedge G$  and  $F \Rightarrow G$  as abbreviations when *p* does not occur in *G*; this yields the non-sequential variants of the connectives as special cases.

At this point it is not possible for *G* to actually make use of the truth of *F* because proofs cannot occur in formulas. This will change by the use of a description operator, and we will also use it when defining our typed set theory.

The proof rules for the sequential connectives are almost the same as for the usual ones. The only difference is that the proof of the first argument has to be supplied in a few places:

$$\begin{aligned}\wedge I &: \{p : \text{True } F\} \text{ True } Gp \rightarrow \text{True } F \wedge [p] Gp \\ \wedge E_l &: \text{True } F \wedge [p] Gp \rightarrow \text{True } F \\ \wedge E_r &: \{q : \text{True } F \wedge [p] Gp\} \text{ True } G(\wedge E_l q) \\ \Rightarrow I &: (\{p : \text{True } F\} \text{ True } Gp) \rightarrow \text{True } F \Rightarrow [p] Gp \\ \Rightarrow E &: \text{True } F \Rightarrow [p] Gp \rightarrow \{p : \text{True } F\} \text{ True } Gp\end{aligned}$$

Note that these rules contain the rules for the non-sequential connectives as special cases. We omit the well-known encoding of the introduction and elimination proof rules for the remaining connectives.

The *description operator* is a binder that takes a formula  $Fx$  with a free variable and returns the unique  $x$  satisfying  $Fx$ . This is of course not well-formed for all  $F$ . Therefore,  $\delta$  takes a dependent argument, which is a proof of  $\exists^{*!}[x]Fx$ . Here  $\exists^{*!}$  abbreviates the quantifier of unique existence. It can be encoded naturally using

$$\begin{aligned}\exists^{*!} &: (\text{set} \rightarrow \text{prop}) \rightarrow \text{set} = [F](\exists^{*}[x]Fx \wedge (\forall^{*}[y]Fy \Rightarrow y \doteq^* x)). \\ \delta &: \{F : \text{set} \rightarrow \text{prop}\} (\text{True } \exists^{*!}[x]Fx) \rightarrow \text{set}\end{aligned}$$

Here dependent types permit us to require a proof of unique existence as an argument thus guaranteeing that only well-formed terms are formed. This is in contrast to the two description operators that are formalized in Isabelle/ZF or induced by the Mizar type system, respectively. Both are well-formed even for unsatisfiable formulas, in which case they return an arbitrary element. Thus, both Isabelle/ZF and Mizar assume not only the axiom of choice but also the existence of a global choice function, a commitment that we can avoid.

$\delta$  comes with an axiom scheme

$$ax_\delta : \text{True } F (\delta ([x]Fx) P)$$

for an arbitrary proof  $P$ , which states that  $\delta$  indeed yields the element with property  $F$ . Note that proof irrelevance is derivable from  $ax_\delta$ , i.e.,  $(\delta F P)$  returns the same object no matter which proof  $P$  is used.

Note that both sequential connectives and the description operator crucially depend on the existence of dependent types in the logical framework.

### 5.1.2. Untyped Set Theory

Regarding the *primitive symbols*, our encoding attempts to stay as closely to mathematical practice as possible. We only use a single primitive non-logical symbol: the binary predicate  $\in : \text{set} \rightarrow \text{set} \rightarrow \text{prop}$ . This means that the only terms are the variables and those obtained from the description operator. Thus, all mathematical symbols besides  $\in$  are introduced as abbreviations for sets whose (unique) existence has been proved.

Our encoding is in contrast to Mizar where primitive function symbols are used for singleton, unordered pair, and union ([\[Try89\]](#)) together with Tarski's

axiom of universes, and to Isabelle/ZF where primitive function symbols are used for empty set, powerset, union, infinite set, and replacement ([PC93]).

This permits us to follow the literature and encode all ZFC operations as existential axioms. For example, we can use

$$\forall^*[X] \exists^*[u] (\forall^*[y] (\exists[x] x \in X \wedge y \in x) \Rightarrow y \in u)$$

as the axiom of union. From this we can obtain a proof  $P X$  of

$$\exists^{*!}[u] (\forall^*[y] (\exists[x] x \in X \wedge y \in x) \Leftrightarrow y \in u)$$

for an arbitrary set  $X$ . Then we can define the union operation as

$$\bigcup : set \rightarrow set = [X] (\delta ([u] \forall^*[y] (\exists[x] x \in X \wedge y \in x) \Leftrightarrow y \in u) (P X))$$

Similarly, we proceed to define the empty set, unordered pairs, and powersets using the respective axiom, as well as encodings of the sets  $\{x \in A | F(x)\}$  and  $\{f(x) : x \in A\}$  using the axiom schemes of specification and replacement, respectively. Furthermore, we use  $\delta$  and the axiom of infinity to obtain a specific infinite set, in our case the set of natural numbers. For all results described in this paper, we do not use  $\delta$  or any of these seven axioms anywhere else, i.e., all other sets and operations on sets are defined in terms of these seven applications of  $\delta$ .

Our results do not actually require the axioms of choice and excluded middle. Similarly, regularity and infinity are not used for the results presented in this section. However, these axioms may be needed to construct specific models such as models with an infinite domain.

We define ordered pairs  $(x, y)$  as  $\{\{x\}, \{\{y\}, \emptyset\}\}$ . Our definition is unusual and similar to Wiener's  $\{\{\{x\}, \emptyset\}, \{\{y\}\}\}$  ([Wie67]). We do not use the common Kuratowski pairs  $\{\{x\}, \{x, y\}\}$  because reasoning about them often requires the principle of excluded middle to distinguish the cases  $x \doteq^* y$  and  $x \neq^* y$ , which we try to avoid unless necessary. However, we immediately define the projections  $\pi_1$  and  $\pi_2$  and prove the conversions  $\pi_i(x_1, x_2) \doteq^* x_i$  and  $isPair u \Rightarrow (\pi_1(u), \pi_2(u)) \doteq^* u$ , which are known from simple type theory. Afterwards all work is carried out in terms of these derived operations and properties so that the specific definition of pairing becomes less relevant. (The LF module system can check that only the derived operations are used.)

Based on ordered pairs, we can define relations and functions in the usual way. In particular, we define set theoretical notions of  $\lambda$  abstraction and application as operations  $\lambda^* A ([x : set] f x)$  encoding  $\{(x, f(x)) : x \in A\}$  and  $f @^* a$  encoding “the  $b$  such that  $(a, b) \in f$ ”. Again we immediately prove the conversions of  $\beta$ - and  $\eta$ -equality and use only those later on.

In a similar way, we define various other notions such as subset, singleton set, binary union, intersection, difference, disjoint union, etc., and derive natural deduction rules for them.

### 5.1.3. Typed Set Theory

We have already remarked that a major problem with formalizations of set theory is their complexity. Type theories favor algorithmic definitions and decidable notions, and these prove indispensable when formalizing major parts of mathematics in a computer.

It is not surprising that most of the biggest successes of formalized mathematics such as the formalization of the Four Color Theorem and the Kepler Conjecture ([Gon06], [Hal03]) are achieved in type theory-based formalizations of mathematics – Coq ([BC04]) and HOL Light ([Har96]), respectively. Similarly, while Mizar is untyped a priori, it supports a very sophisticated type system as a derived notion that is internally represented using predicates over sets ([Wie07]). Isabelle/ZF offers much weaker support for typed reasoning, and this is one of the main reasons why both tool support and available content are farther developed in Isabelle/HOL ([NPW02]) using a typed foundation rather than in Isabelle/ZF.

In LF, again using the dependent typing, we can derive typed set theory in a rather simple way. The crucial idea is to use the dependent sum type  $\text{elem } A := \Sigma_{x:\text{set}}(\text{True } x \in A)$  to represent the set  $A$ . Thus, elements  $x$  of  $A$  are represented as pairs  $(x, P)$  where  $P$  is a proof that  $x$  is indeed in  $A$ . If we also require proof irrelevance, i.e.,  $(x, P) = (x, P')$ , then the type  $\text{elem } A$  has exactly one term for every element of  $A$ . This is inspired by the Scunak language ([Bro06]), which uses this representation as a primitive notion and provides implementation support for it.

It is a minor inconvenience that LF (unlike Scunak) supports neither dependent sum types nor the ability to make all elements of a type definitionally equal (which would permit to state the proof irrelevance). Therefore, we have to add  $\text{elem}$  and its properties as primitives to our LF encoding as shown below

```

 $\text{elem} : \text{set} \rightarrow \text{type}$ 
 $\text{el} : \{x : \text{set}\} \text{True } x \in A \rightarrow \text{elem } A$ 
 $\text{which} : \text{elem } A \rightarrow \text{set}$ 
 $\text{why} : \{a : \text{elem } A\} \text{True } (\text{which } a) \in A$ 

```

along with an axiom for proof irrelevance and one for  $(\text{which } (\text{el } x P)) \doteq^* x$ .  $\text{el}$ ,  $\text{which}$ , and  $\text{why}$  would simply be pairing and the two projections if LF had sum types. While this increases the needed primitives, these declarations only emulate features that could easily be added to the LF language: Dependent sum types are used in, e.g., [ML74] and [Nor05]; proof irrelevance is added in [LP09].

Using the types  $\text{elem } A$ , we can now lift all the basic untyped operations introduced above to the typed level. In particular, we define typed quantifiers  $\forall, \exists$ , typed equality  $\doteq$ , and typed function spaces  $\implies$  in the following.

Firstly, we define *typed quantifiers* such as  $\forall : (\text{elem } A \rightarrow \text{prop}) \rightarrow \text{prop}$ . In higher-order logic with internal propositions ([Chu40], compare  $\text{prop} : \text{set}$  rather than  $\text{prop} : \text{type}$  in Sect. 4.3), such typed quantification can be defined easily. In untyped set theory, this is intuitively possible using relativization,

e.g.,  $\forall F := \forall^*[x] x \in A \Rightarrow F x$  for  $F : \text{elem } A \rightarrow \text{prop}$ .

However, an attempt to formally define typed quantification like this meets a subtle difficulty: In  $\forall F$ ,  $F$  only needs to be defined for elements of  $A$  whereas in  $\forall^*[x] x \in A \Rightarrow F x$ ,  $F$  must be defined for all sets. Thus,  $\forall$  is more general than  $\forall^*$  in that it permits a weaker argument. Of course, in  $\forall^*[x] x \in A \Rightarrow F x$ , it is intended not to consider  $F x$  if  $x \notin A$ . This is the motivation behind the introduction of sequential connectives in Sect. 5.1.1 above.

Using sequential connectives, we can define  $\forall$  and  $\exists$  as follows:

$$\begin{aligned}\forall &: (\text{elem } A \rightarrow \text{prop}) \rightarrow \text{prop} = [F] (\forall^*[x] x \in A \Rightarrow [p] (F (el x p))) \\ \exists &: (\text{elem } A \rightarrow \text{prop}) \rightarrow \text{prop} = [F] (\exists^*[x] x \in A \wedge [p] (F (el x p)))\end{aligned}$$

Then we can derive introduction and elimination rules for  $\forall$  and  $\exists$ , which look the same as those for the untyped ones.

Secondly, *typed equality* is easy to define:

$$\doteq : \text{elem } A \rightarrow \text{elem } A \rightarrow \text{prop} = [a] [b] (\text{which } a) \doteq^* (\text{which } b)$$

It is easy to see that all rules for  $\doteq^*$  can be lifted to  $\doteq$ .

Finally, we can define *function types* that are defined in terms of untyped functions:

$$\begin{aligned}\implies &: \text{set} \rightarrow \text{set} \rightarrow \text{set} = \dots \\ \lambda &: (\text{elem } A \rightarrow \text{elem } B) \rightarrow \text{elem } (A \implies B) = \dots \\ @_ &: \text{elem } (A \implies B) \rightarrow \text{elem } A \rightarrow \text{elem } B = \dots \\ \text{beta} &: \text{True } ((@(\lambda [x] F x) A) \doteq^* F A) = \dots \\ \text{eta} &: \text{True } ((\lambda [x] (@ F x)) \doteq^* F) = \dots\end{aligned}$$

We omit the quite involved definitions and only mention that the typed quantifiers and thus the sequential connectives are needed in the definitions.

#### 5.1.4. The Booleans

The Booleans  $\mathbb{B}$  are easy to define as the set containing the two elements  $0 = \emptyset$  and  $1 = \{\emptyset\}$ . However, it is interesting to note that there are two different ways to define this set: We can use the unordered pair  $\{0, 1\}$  or the powerset  $\mathcal{P}(1)$ .

Clearly,  $\{0, 1\}$  is a two-element set and  $\{0, 1\} \subseteq \mathcal{P}(1)$ , but it turns out that the two sets are only equal in the presence of the axiom of excluded middle. In fact, – maybe surprisingly – by using the set  $\{x \in 1 \mid F\}$ , it can be shown that  $\{0, 1\} \doteq^* \mathcal{P}(1)$  is equivalent to  $F \vee \neg F$  for all formulas  $F$ .

Therefore, an intuitionistic set theory would have to define  $\mathbb{B} = \{0, 1\}$ , and it presents no fundamental obstacles to do so. But it is more convenient to use  $\mathbb{B} = \mathcal{P}(1)$  because then all operations on the Booleans can be obtained from the lattice operations in  $\mathcal{P}(1)$ . Therefore, we put  $\mathbb{B} = \mathcal{P}(1)$ .

Then most of the operations on the Booleans are straightforward:

$$\begin{aligned}
\neg_* &: \text{elem } \mathbb{B} \implies \mathbb{B} = \lambda[x] 1 \setminus x \\
\wedge_* &: \text{elem } \mathbb{B} \implies \mathbb{B} \implies \mathbb{B} = \lambda[x] \lambda[y] x \cap y \\
\vee_* &: \text{elem } \mathbb{B} \implies \mathbb{B} \implies \mathbb{B} = \lambda[x] \lambda[y] x \cup y \\
\Rightarrow_* &: \text{elem } \mathbb{B} \implies \mathbb{B} \implies \mathbb{B} = \lambda[x] \lambda[y] \text{reflect } (x \subseteq y) \\
\forall_* &: \text{elem } (A \implies \mathbb{B}) \implies \mathbb{B} = \lambda[f] \bigcap (\text{image } f) \\
\exists_* &: \text{elem } (A \implies \mathbb{B}) \implies \mathbb{B} = \lambda[f] \bigcup (\text{image } f)
\end{aligned}$$

where  $\setminus$  returns the difference of two sets,  $\text{image } f$  is the image of the function  $f$ ,  $\text{reflect } F$  encodes the set  $\{x \in 1 \mid F\}$ , and  $\bigcap$  and  $\bigcup$  return the union or intersection, respectively, of a set of sets. We omit their definitions.

In the usual way, we can prove the basic properties of the lattice operations, from which we can prove the intended properties of the Boolean operations.

Finally, we use the axiom of excluded middle once to prove that  $\mathbb{B}$  is equal to  $\{0, 1\}$ .

### 5.2. Viewing Higher-Order Logic in Set Theory

Now that we have developed an encoding of set theory, we want to show that it subsumes the meta-language *HOL* used in Sect. 4 to represent model theory. Let *ZFC* be the LF signature containing our set theory. Then the subsumption can be expressed in LF formally as a view from *HOL* to *ZFC*.

This basic idea of the view is straightforward because all constants of *HOL* are mapped to *ZFC*-constants of the same name, i.e., we have a view

```
%view φ : HOL → ZFC = {
  set    := set
  elem   := elem
  prop   := prop
  True   := True
  :
}
```

The view must also map all proof rules of *HOL* to proofs of the corresponding *ZFC* theorems. For the propositional connectives, those are the same rules assumed for the first-order logic underlying *ZFC*. For the quantifiers and equality, they are derived rules for  $\forall$ ,  $\exists$ , and  $\doteq$ . For  $\beta$ - and  $\eta$ -equality, they are the corresponding derived rules of *ZFC*.

Instead of *ZFC* set theory, we could use any other foundation into which we can give such a view  $φ$ . This includes other set theories but also typed foundations such as the usual higher-order logic with internal propositions. For example, the latter arises if we use the type theory from Sect. 2.2 but with a declaration  $\text{prop} : \text{tp}$ ; in that case the view would map  $\text{prop}$  to  $\text{tm prop}$ .

### 5.3. Viewing Model Theory in Set Theory

Next we should define a view  $μ'$  from *FOL<sup>mod</sup>* to *ZFC*. However, it is not possible to fix the value of  $μ'(\text{univ})$  because it may be different in every model. Thus,  $μ'$  must be parametric in the choice of  $μ'(\text{univ})$  and consequently also in

that of  $\mu'(\text{nonemp})$ . We solve that by introducing  $FOL^{mod+}$  as  $ZFC$  with two free parameters as below

```
%sig FOLmod+ = {
  %include ZFC
  U : set
  P : True ∃*[x] x ∈ U
}
```

and giving a view  $\mu' : FOL^{mod} \rightarrow FOL^{mod+}$  instead.

This view is again modular to interpret every connective separately. Moreover,  $\mu'$  includes (and thus reuses)  $\varphi$  so that the LF module system guarantees that the diagram on the right commutes. We will not present the modular structure here, but rather give examples of the most interesting cases.

$\mu'$  interprets the booleans as the two-element set of truth values, and maps *univ* and *nonemp* to the free parameters *U* and *P* as below.

```
bool      :=  ⊤
0         :=  0
1         :=  1
⋮
univ     :=  U
nonemp   :=  P
```

All connectives and quantifiers are mapped to their counterparts on  $\mathbb{B}$ , e.g.,  $\wedge$  is mapped to  $\wedge_*$  and  $\forall$  to  $\forall_*$ . The proofs of the axioms are simple.

#### 5.4. Representing Model Theory

*Models.* Assume a logic encoding  $(L^{syn}, L^{pf}, \pi, L^{mod+}, \mu, \mu')$  using a foundation  $F$  as before. The basic idea behind the encoding of  $L$ -models  $M$  of  $\Sigma$  is given by the diagram below. Here  $\Sigma^{mod+}$  arises in the same way as  $\Sigma^{mod}$ , i.e., by pushout of  $\Sigma^{syn}$  along  $\mu, \mu'$  over  $L^{syn}$ , or equivalently by pushout of  $\Sigma^{mod}$  along  $\mu'$  over  $L^{mod}$ . Then the encoding  $\lceil M \rceil$  of  $M$  is a morphism from  $\Sigma^{mod+}$  to  $F$  such that following diagram commutes:

$$\begin{array}{ccccc} L^{syn} & \xrightarrow{\quad} & \Sigma^{syn} & & \\ \searrow \mu, \mu' & & \searrow \mu_\Sigma, \mu'_\Sigma & & \\ & L^{mod+} & \xrightarrow{\quad} & \Sigma^{mod+} & \\ & \uparrow & & \downarrow \lceil M \rceil & \\ F & \xrightarrow{id_F} & F & & \end{array}$$

*Notation 39.* The notation  $\lceil M \rceil$  was already used in Sect. 4 for the encoding of  $M$  in HOL. We will reuse it in Sect. 5 for the encoding of a model  $M$  in LF.

This encoding captures and formalizes two of the most central intuitions about the syntax and semantics of formal languages. Firstly, the semantics of a formal language is a structure-preserving translation of the syntax into some semantic realm. For logics the semantic realm is usually mathematics, in our case encoded by a foundation  $F$ . The interpretation function is given by  $\mu_\Sigma \mu'_\Sigma \lceil M \rceil$ . Secondly, the syntax consists of two parts: logical and non-logical symbols. In our case, the semantics of the logical symbols is given by a fixed morphism  $\mu \mu'$ , and the semantics of the non-logical symbols is given by the morphism  $\lceil M \rceil$ .

$\lceil M \rceil$  must map all symbols of  $\Sigma^{mod+}$ , which can be split into three groups. Firstly, symbols included from  $F$  have a fixed meaning in the foundation; the commutativity ensures that  $\lceil M \rceil$  is the identity on them. Secondly, symbols included from  $L^{mod+}$  encode fixed parts of the models that do not depend on the signature; in the case of FOL, this is the universe encoded using the symbols  $U$  and  $P$ . Thirdly, the symbols inherited from  $\Sigma^{syn}$  when constructing the pushout are the non-logical symbols.

We can formalize the above intuitions as follows:

**Definition 40** (Encoding Models). An *LF-based model* of a FOL-signature or theory  $\Sigma$  is a morphism  $I : \Sigma^{mod+} \rightarrow ZFC$  that is a retraction of the inclusion  $ZFC \hookrightarrow \Sigma^{mod+}$ .

Note that this definition includes the case when  $\Sigma$  is a theory. In that case, LF-based models map the axioms in  $\Sigma^{mod+}$  (which stem from the pushout of  $\Sigma^{syn}$ ) to proof terms in  $ZFC$ .

**Definition 41** (Encoding Semantics). Assume an LF-based model  $I$  of a FOL-signature  $\Sigma$  and a term or formula  $E$  over  $\Sigma$ . Then the LF-based semantics of  $E$  is given by  $(\mu_\Sigma \mu'_\Sigma I)(\lceil E \rceil)$ , which we also write as  $\llbracket E \rrbracket^I$ .

Due to the type preservation of LF signature morphism, the LF-based semantics  $\llbracket t \rrbracket^I$  of a term  $t$  is indeed an (encoding of an) element of the universe and the LF-based semantics  $\llbracket F \rrbracket^I$  of a formula is an (encoding of a) truth value.

We have a very strict notion of identity between LF-based models inherited from LF signature morphisms: Two signature morphisms are equal if they agree for all arguments up to  $\beta\eta$ -equality. For the representation of models, we need a more relaxed notion based on whether equality can be proved in ZFC:

**Definition 42** (Equality of Models). Two LF-based model  $I_1$  and  $I_2$  of a FOL-signature or theory  $\Sigma$  are *provably equal* if  $\text{True } I_1(U) \doteq^* I_2(U)$  and all types  $\text{True } I_1(s) \doteq I_2(s)$  for all function or predicate symbols  $s$  of  $\Sigma$  are inhabited.

Note that if  $\Sigma$  is a theory, we do not require the equality of  $I_1(a)$  and  $I_2(a)$  for axioms  $a$ , i.e., the proofs of the axioms are irrelevant (as long as they exist). Similarly, we do not require any equality of  $I_1(P)$  and  $I_2(P)$ .

*Model Reduction.* As before, a FOL-signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$  is encoded as an LF signature morphism  $\sigma^{syn} : \Sigma^{syn} \rightarrow \Sigma'^{syn}$ , and pushout along  $\mu \mu'$  yields

an LF signature morphism  $\sigma^{mod+} : \Sigma^{mod+} \rightarrow \Sigma'^{mod+}$ . Then our representation of models as morphisms permits a very elegant representation of model reduction by composition with  $\sigma^{mod+}$ .

**Theorem 43** (Encoding Model Reduction). *Assume a FOL signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$ . If  $I$  is an LF-based model of  $\Sigma'$ , then  $\sigma^{mod+} I$  is an LF-based model of  $\Sigma$ . Moreover, for a term or formula  $E$  over  $\Sigma$ , we have*

$$\llbracket E \rrbracket^{\sigma^{mod+} I} = \llbracket \sigma(E) \rrbracket^I.$$

*Proof.* Both claims follow immediately from the properties of signature morphisms and the construction of  $\sigma^{mod+}$  by pushout.  $\square$

Readers familiar with institutions will recognize the second claim as the satisfaction condition.

*Example 44* (Continued). Consider the model *Int* from Ex. 15. It can be represented as an LF-based model  $I : GrpSig^{mod+} \rightarrow ZFC$ .  $I(U)$  is the *ZFC*-term for the set of integers, which is straightforward to define using the natural numbers.  $I(P)$  is some proof over *ZFC* that proves  $U$  is non-empty. Then we only have to define  $I$  for the symbols  $\circ$ ,  $e$ , and *inv*, which are mapped to *ZFC*-expressions representing the operations  $+$ ,  $0$ , and *inv* on the integers.

Via pushout, the FOL-signature morphism  $MonGrp : MonSig \rightarrow GrpSig$  from Ex. 5 gives rise to an LF signature morphism  $MonGrp^{mod+} : MonSig^{mod+} \rightarrow GrpSig^{mod+}$ . The encoding of the model reduct  $Mod(MonGrp)(Int)$  is obtained as the composition  $MonGrp^{mod+} I$  as in the following diagram:

$$\begin{array}{ccccc} & & \text{FOL}^{mod+} & \xhookrightarrow{\quad} & \text{MonSig}^{mod+} \xrightarrow{\quad} \text{GrpSig}^{mod+} \\ & \nearrow & & & \downarrow \text{MonGrp}^{mod+} \\ & & & & \downarrow I \\ & & & & ZFC \end{array}$$

$MonGrp^{mod+} I$  agrees with  $I$  but is not defined for *inv*.

Alternatively, we can encode *Int* as a model of the theory *Group*. That model must map from *Group*<sup>mod+</sup> to *ZFC*, i.e., additionally maps the axioms of *Group* to *ZFC*. These are mapped to proofs that the integers do indeed satisfy the axioms of a group.

### 5.5. Adequacy

The encoding of models as signature morphisms raises a difficult question based on a simple cardinality argument: There is a proper class of FOL-models of  $\Sigma$ , but only countably many signature morphisms to encode them. Therefore, we have to look carefully at the adequacy of our encoding.

First of all we have:

**Lemma 45.** *Our encoding of set theory is adequate in the following sense:*

- *Every closed term  $t : \text{set}$  induces a set  $\llcorner t \lrcorner$ .*
- *Two closed terms  $s, t$  induce the same set if the type  $\text{True } s \doteq^* t$  is inhabited.*
- *For every closed term  $t : \text{elem } A$ , the set  $\llcorner \text{which } t \lrcorner$  is an element of  $\llcorner A \lrcorner$ .*

*Proof.* All claims rely on the assumption that every proof rule of the underlying first-order logic and every axiom in  $ZFC$  is sound with respect to the platonic universe of set theory. For researchers objecting to parts of the encoding (e.g., to excluded middle), the corresponding result holds after modifying  $ZFC$ .

For the first claim, we expand all definitions in  $t$ . This yields either a term of the form  $\text{which}(\text{el } t' P)$ , which is provably equal to the smaller term  $t'$  so that we can recurse, or a term of the form  $\delta F Q$ . Thus, for every term  $t : \text{set}$ , we can obtain a provable formula  $\exists^![x] F x \wedge F t$ . This shows the existence of a set that  $t$  represents. The second claim holds because the existence of a term inhabiting  $\text{True } s \doteq^* t$  shows that  $s \doteq^* t$  is a provable formula. The third claim follows easily from the first one.  $\square$

Then we have:

**Theorem 46.** *Assume a FOL-signature  $\Sigma$ . Every LF-based model  $I$  induces a FOL-model  $\llcorner I \lrcorner$ .*

*Proof.* The universe of  $\llcorner I \lrcorner$  is the set  $\llcorner I(U) \lrcorner$ . For  $\Sigma$ -symbols  $s$ , the interpretation of  $s$  in  $\llcorner I \lrcorner$  is the object  $\llcorner \text{which } I(s) \lrcorner$ .  $\square$

Whether or not all sets and models are induced by LF-terms and LF-based models is a philosophical question. If we adopt a formalist or even a constructivist point of view, then the LF-terms are (representatives of) all the sets and thus the LF-based models are all the models. If we adopt a platonic point of view, only some models can be encoded, but these include – intuitively – all models whose components can be written down or named. Furthermore, we can always create variations of our signature  $ZFC$  to accommodate other perspectives. For example, we can add a choice operator to represent models obtained by applying the axiom of choice.

Then we have the following adequacy results for those models that can be represented:

**Definition 47.** A FOL-model  $M$  is *definable* if  $M = \llcorner I \lrcorner$  for some LF-based model  $I$ . In that case we also write  $I = \lceil M \rceil$ .

**Theorem 48** (Adequacy for Model Theory). *For every definable FOL-model  $M$  of  $\Sigma$ , and every term or formula  $E$  over  $\Sigma$ :*

$$\llbracket E \rrbracket^M = \llcorner \llbracket E \rrbracket^{\lceil M \rceil} \lrcorner$$

*Proof.* This is straightforward from the definitions. The only subtlety is to show that  $\llbracket \llbracket E \rrbracket^{\lceil M \rceil} \rrbracket$  does not depend on which LF-based model is chosen for  $\lceil M \rceil$ . But that is the case because any two possible choices must be provably equal.  $\square$

**Theorem 49** (Adequacy for Model Reduction). *For every FOL-signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$  and every definable FOL-model  $M'$  of  $\Sigma'$ :*

$$Mod(\sigma)(M') = \llbracket \sigma^{mod+} \lceil M' \rceil \rrbracket$$

*In particular, reducts of definable models are definable.*

*Proof.* This is straightforward from the definitions.  $\square$

Finally, even though we may not be able to encode all models, we can adequately encode the property of being a model theoretical theory morphism:

**Theorem 50** (Adequacy for Model Theoretical Theory Morphisms). *For a FOL-signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$ , we have  $\sigma : (\Sigma, \Theta) \xrightarrow{M} (\Sigma', \Theta')$  iff  $\sigma^{mod+} : \Sigma^{mod+} \rightarrow \Sigma'^{mod+}$  can be extended to an LF signature morphism  $\vartheta : (\Sigma, \Theta)^{mod+} \rightarrow (\Sigma', \Theta')^{mod+}$ .*

*Proof.* Consider the following commutative diagram, where as before  $\sigma^{syn} : \Sigma^{syn} \rightarrow \Sigma'^{syn}$  is the encoding of  $\sigma$  and all  $^{mod+}$ -nodes arise as pushouts of the corresponding  $^{syn}$ -node along  $\mu \mu' : L^{syn} \rightarrow L^{mod+}$ :

$$\begin{array}{ccccc} \Sigma^{syn} & \xrightarrow{\sigma^{syn}} & \Sigma'^{syn} & & \\ \downarrow & \searrow & \downarrow & & \\ (\Sigma, \Theta)^{syn} & & (\Sigma', \Theta')^{syn} & & \\ \downarrow & \searrow & \downarrow & & \\ \Sigma^{mod+} & \xrightarrow{\sigma^{mod+}} & \Sigma'^{mod+} & & \\ \downarrow & & \downarrow & & \\ (\Sigma, \Theta)^{mod+} & \xrightarrow{\vartheta} & (\Sigma', \Theta')^{mod+} & & \end{array}$$

The claim is that  $\sigma$  is a model theoretical theory morphism iff such a  $\vartheta$  exists.

The right-to-left direction is easy:  $\vartheta$  contains proof terms that show that the reduct of a  $(\Sigma', \Theta')$ -model satisfies all the axioms of  $\Theta$ .

To show the converse direction, recall that we only consider finite theories, and observe that given  $\Sigma, \Theta, \Sigma', \Theta'$ , and  $\sigma$ , we can write the following formula in the first-order language of ZFC.

$$\begin{aligned} f = & \forall U, s_1, \dots, s_n. (M'(U, s_1, \dots, s_n) \wedge \bigwedge_i A'_i(U, s_1, \dots, s_n)) \\ & \Rightarrow (M(U, \sigma_1, \dots, \sigma_m) \wedge \bigwedge_i A_i(U, \sigma_1, \dots, \sigma_m)) \end{aligned}$$

Here  $m$  is the number of function and predicate symbols declared in  $\Sigma$ ;  $\Sigma'$  declares function and predicate symbols named  $s_1, \dots, s_n$ ;  $M(x, y_1, \dots, y_m)$  expresses that  $(x, y_1, \dots, y_m)$  is a  $\Sigma$ -model with universe  $x$ ;  $A_i(x, y_1, \dots, y_m)$  expresses that said  $\Sigma$ -model satisfies the  $i$ -th axiom in  $\Theta$ ;  $M'(x, y_1, \dots, y_n)$  and  $A'_i(x, y_1, \dots, y_n)$  are defined accordingly for  $\Sigma'$  and  $\Theta'$ ; and  $\sigma_i \in \{s_1, \dots, s_n\}$  is the result of applying  $\sigma$  to  $s_i$ .

Then  $f$  is a theorem over ZFC iff  $\sigma$  is a model-theoretical theory morphism. So assume a proof of  $f$ , from which we obtain a corresponding LF proof term  $p$  over the signature  $ZFC$ .

Moreover, over the signature  $\Sigma'^{mod+}$ , which extends  $ZFC$ , we have a proof term  $q$  proving  $M'(U, s_1, \dots, s_n) \wedge \bigwedge_i A'_i(U, s_1, \dots, s_n)$ . From  $p$  and  $q$ , we obtain a  $\Sigma'^{mod+}$ -proof term  $r_i$  proving  $A_i(U, \sigma_1, \dots, \sigma_n)$  for the  $i$ -th axiom in  $\Theta$ . By putting  $\vartheta(a_i) = r_i$ , we obtain the needed LF signature morphism.  $\square$

The above proof rests on the philosophical assumption that a statement about ZFC – in this case, the statement  $\sigma : (\Sigma, \Theta) \xrightarrow{\text{M}} (\Sigma', \Theta')$  – can only be true if there is a proof of it in the first-order language of ZFC. Moreover, we assume that this first-order language is indeed the one that we encoded in  $ZFC$ . Researchers working with a different variant of set theory can apply Thm. 50 accordingly after modifying  $ZFC$ .

Finally, observe that the proof depends on the ability to switch between the internal representation of models – the tuples  $(U, s_1, \dots, s_n)$ , which can be encoded as LF terms over  $ZFC$  – and the external representation of models as LF signature morphisms into  $ZFC$ .

## 6. Related Work

There are formalizations of the semantics of formal languages in various frameworks. For example, in [BKV09], a simple functional programming language is formalized in Coq ([BC04]). In [MNvOS99], a simple while-language is formalized in Isabelle/HOL ([NPW02]). Both use domain-theoretical models formalized based on posets where we use set theoretical models. In [CD97], simple type theory is formalized in ALF ([MN94]) using the normalization-by-evaluation method. They use a glued model in which interpretations are paired with normal forms. A similar result was obtained in [Coq02] using Kripke models.

Although the settings of these results are very different from each other's and from ours, all approaches are quite similar in that they define syntax and models and give an interpretation function satisfying a soundness property. A novelty of our approach is to base the models on an explicitly formalized foundation: We formalize set theory and use sets as the universes of the models. In the cited formalizations, on the other hand, the universes are types of the framework's type theory, which thus acts as an implicit foundation. Our approach makes the foundation flexible and avoids such an implicit commitment. For example, we could represent the cited formalizations in LF using an LF signature for Coq, Isabelle/HOL, or ALF, respectively, instead of the one for set theory.

Another difference is that the cited approaches all represent the interpretation function as a function of the framework’s type theory. As this is impossible in LF, we use LF signature morphisms, which are less flexible but provide an elegant characterization of sound interpretation functions.

Dually, there are other formalizations of the semantics of formal languages in Twelf. In [App01], a formalization of HOL in Twelf is used to define the semantics of a machine language for proof-carrying code. This work does not focus on the separation of syntax, models, and interpretation. Instead, all notions are introduced as definitional extensions of HOL. From our perspective, they use HOL as the foundation and define the models by extending HOL whereas syntax and interpretation function are left implicit. In [LCH07], an SML-equivalent language is given, and state-transition systems are used to formalize the evaluation of expressions. These corresponds to our models, but they are not strictly separated from the syntax as in our case. Like our views, the interpretation function and soundness proof live on the meta-level: They are given by a number of logic programs formalized in the Twelf meta-theory. Despite their formidable sizes, both these Twelf developments are monolithic because they predate the module system.

Regarding our specific encodings, the encodings of first-order syntax and proof theory are straightforward and well-known (see, e.g., [HHP93]). Only the systematic use of modularity is novel. Our encoding of HOL is well-known, too, but note that there are two flavors of HOL, both based on simple type theory. The most common one based on [Chu40] treats propositions as terms of a special type *prop*. In LF, that would correspond to the declaration *prop : set*. This flavor is used in, e.g., [HST94], [App01], [Har96], [NPW02]. The advantage is that the connectives and quantifiers can be introduced as HOL-terms. Our variant with *prop : type* treats propositions as external to the type theory, i.e., propositions are not HOL-terms. This is more general and necessary to treat HOL as a fragment of first-order set theory where propositions and sets are strictly separated.

Our encoding of set theory is novel both in general and in the context of Twelf. The most advanced other formalizations of set theories are the ones in Mizar ([TB85]) and Isabelle/ZF ([PC93]). Scunak ([Bro06]) is a recent system developed specifically to exploit dependent type theory when encoding set theory. We discussed the differences between these and our encoding in Sect. 5.1. Other ways to encode set theory in dependently-typed frameworks use the framework’s type theory as the foundation of mathematics, as in [Acz78]. Such encodings can be mechanized in systems like Agda and Coq, see e.g., [Gri09].

The main advantage of these other systems over LF/Twelf is that they provide a stronger notion of definitional equality and (semi)-automated proof support. For example, Isabelle, Agda, and Coq permit the declaration of recursive functions that are evaluated automatically by the framework. Scunak implements the proof irrelevance we axiomatize in Sect. 5.1.3. All of them are connected to automated or semi-automated proof tools or provide tactic languages. LF, on the other hand, is ontologically much simpler, even minimalistic. Con-

sequently, proof terms are fully explicit and more complicated to construct by hand. But LF (as well as Scunak) can benefit from the use of higher-order abstract syntax, which simplifies the reasoning about adequacy relative to traditional mathematics.

Finally, the idea of encoding models as morphisms goes back to Lawvere’s work on functorial models [Law63] and the work on initial algebra semantics, e.g., in [GTW78]. While these have been developed in logical frameworks on paper before, e.g., in [MTP97] and [GMdP<sup>+</sup>07], our work marks the first time that they can be formalized and machine-checked in a logical framework.

## 7. Conclusion

We have given a comprehensive representation of first-order logic in a logical framework. Contrary to previous work, our representation covers both the proof and the model theoretical semantics given as provability and satisfaction, respectively. For example, the framework of institutions has been applied to the model theoretical semantics ([GB92]), and the framework LF to the proof theoretical semantics ([HHP93]), but a comprehensive representation has so far been lacking. This was due to the large ontological and philosophical differences between these two views on logic.

These differences are so big that we needed three major preliminary efforts to make this representation possible. In [Rab08, Rab10], we conceived the logical framework combining model and proof theory that we have built upon here. In [RS09], we gave the LF and Twelf module system that we used to implement the representation. In fact, our work is the largest case study in the Twelf module system to date. And finally, we needed a representation of a foundation of mathematics, which we have described in Sect. 5.1. These combine to a strong and flexible framework whose potential is exemplified by the representation of FOL we have given.

Our work will leverage future representations in two ways. Firstly, framework design and implementation are in place now, and this paper provides a detailed template how to represent logics. In fact, we have started this already ([KMR09]), and we expect further successes fast. Secondly, the Twelf module system permits the reuse of existing representation fragments. Our representation has separated all language features into independent and reusable components so that further logics can be represented by only adding individual language features such as sorted quantification or simple function types. Moreover, the meta-language for the model theory and its interpretation in ZFC set theory are composed modularly as well. Therefore, they cannot only be reused for many other logics but can also be refined flexibly if a more expressive meta-language or a different foundation of mathematics are needed. For example, we could easily extend the meta-language from HOL to a dependently-typed DHOL for a particular logic representation.

An important application of logical frameworks is to use the logic representations to reason about the represented logic. To that effect, we gave adequacy

results for syntax, proof theory, and model theory, and for the respective translations along morphisms. We gave a criterion to prove the soundness of a logic within the framework, and we used this to give a fully machine-verified soundness proof of first-order logic. A similar treatment of completeness remains future work.

Finally our work is part of a larger effort to obtain an atlas of logics and translations between them. Our work explains and exemplifies how logics, foundation, and models should be represented. A similar case study for logic translations is given recently in [Soj10]. Within the LATIN project ([KMR09]), our results will be integrated with the heterogeneous specification tool Hets ([MML07]) and the scalable Web infrastructure based on the markup language OMDoc ([Koh06]).

- [ACTZ06] A. Asperti, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. Crafting a Proof Assistant. In T. Altenkirch and C. McBride, editors, *TYPES*, pages 18–32. Springer, 2006.
- [Acz78] P. Aczel. The Type Theoretic Interpretation of Constructive Set Theory. In A. Macintyre, L. Pacholski, and J. Paris, editors, *Logic Colloquium '77*, pages 55–66. North-Holland, 1978.
- [AHMP98] A. Avron, F. Honsell, M. Miculan, and C. Paravano. Encoding modal logics in logical frameworks. *Studia Logica*, 60(1):161–208, 1998.
- [AHMS99] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an Evolutionary Formal Software-Development Using CASL. In D. Bert, C. Choppy, and P. Mosses, editors, *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1999.
- [App01] A. Appel. Foundational Proof-Carrying Code. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258. IEEE, 2001.
- [Bar92] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.
- [BC04] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.
- [Ber37] P. Bernays, 1937. Seven papers between 1937 and 1954 in the Journal of Symbolic Logic.
- [BKV09] N. Benton, A. Kennedy, and C. Varming. Some Domain Theory and Denotational Semantics in Coq. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2009.

- [Bro06] C. Brown. Combining Type Theory and Untyped Set Theory. In U. Furbach and N. Shankar, editors, *International Joint Conference on Automated Reasoning*, pages 205–219. Springer, 2006.
- [CD97] T. Coquand and P. Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7(1):75–94, 1997.
- [CF58] H. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.
- [CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988.
- [Chu40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.
- [Coq02] C. Coquand. A Formalised Proof of the Soundness and Completeness of a Simply Typed Lambda-Calculus with Explicit Substitutions. *Higher-Order and Symbolic Computation*, 15(1):57–90, 2002.
- [dB70] N. de Bruijn. The Mathematical Language AUTOMATH. In M. Laudet, editor, *Proceedings of the Symposium on Automated Demonstration*, volume 25 of *Lecture Notes in Mathematics*, pages 29–61. Springer, 1970.
- [Dia06] R. Diaconescu. Proof systems for institutional logic. *Journal of Logic and Computation*, 16(3):339–357, 2006.
- [FGT92] W. Farmer, J. Guttman, and F. Thayer. Little Theories. In D. Kapur, editor, *Conference on Automated Deduction*, pages 467–581, 1992.
- [Fra22] A. Fraenkel. The notion of 'definite' and the independence of the axiom of choice. 1922.
- [GB86] J. Goguen and R. Burstall. A study in the foundations of programming methodology: specifications, institutions, charters and parchments. In D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard, editors, *Workshop on Category Theory and Computer Programming*, pages 313–333. Springer, 1986.
- [GB92] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.
- [GMdP<sup>+</sup>07] J. Goguen, T. Mossakowski, V. de Paiva, F. Rabe, and L. Schröder. An Institutional View on Categorical Logic. *International Journal of Software and Informatics*, 1(1):129–152, 2007.

- [Göd40] K. Gödel. The Consistency of Continuum Hypothesis. *Annals of Mathematics Studies*, 3:33–101, 1940.
- [Gon06] G. Gonthier. A computer-checked proof of the four colour theorem, 2006. <http://research.microsoft.com/~gonthier/>.
- [Gor88] M. Gordon. HOL: A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer-Academic Publishers, 1988.
- [GR02] J. Goguen and G. Rosu. Institution morphisms. *Formal Aspects of Computing*, 13:274–307, 2002.
- [Gri09] J. Grimm. Implementation of Bourbaki’s Elements of Mathematics in Coq: Part One, Theory of Sets. Technical Report HAL:inria-00408143, INRIA, 2009.
- [GTW78] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice Hall, 1978.
- [Hal03] T. Hales. The flyspeck project, 2003. See <http://code.google.com/p/flyspeck/>.
- [Har96] J. Harrison. HOL Light: A Tutorial Introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 265–269. Springer, 1996.
- [HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
- [How80] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.
- [HR09] F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. In *Fourth Workshop on Logical and Semantic Frameworks, with Applications*, volume 256 of *Electronic Notes in Theoretical Computer Science*, pages 49–65, 2009.
- [HST94] R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.
- [KMR09] M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN Project, 2009. See <https://trac.omdoc.org/LATIN/>

- [Koh06] M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*. Number 4180 in Lecture Notes in Artificial Intelligence. Springer, 2006.
- [Lan98] S. Mac Lane. *Categories for the working mathematician*. Springer, 1998.
- [Law63] F. Lawvere. *Functional Semantics of Algebraic Theories*. PhD thesis, Columbia University, 1963.
- [LCH07] D. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of Standard ML. In M. Hofmann and M. Felleisen, editors, *Symposium on Principles of Programming Languages*, pages 173–184. ACM, 2007.
- [LP09] W. Lovas and F. Pfenning. Refinement Types as Proof Irrelevance. In P. Curien, editor, *Typed Lambda Calculi and Applications*, volume 5608 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2009.
- [Mes89] J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Proceedings, Logic Colloquium, 1987*, pages 275–329. North-Holland, 1989.
- [MGDT05] T. Mossakowski, J. Goguen, R. Diaconescu, and A. Tarlecki. What is a logic? In J. Béziau, editor, *Logica Universalis*, pages 113–133. Birkhäuser Verlag, 2005.
- [ML74] P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the '73 Logic Colloquium*, pages 73–118. North-Holland, 1974.
- [ML96] P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):3–10, 1996.
- [MML07] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522, 2007.
- [MN94] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*, pages 213–237. Springer, 1994.
- [MNvOS99] O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF=HOL+LCF. *Journal of Functional Programming*, 9(2):191–223, 1999.

- [MTP97] T. Mossakowski, A. Tarlecki, and W. Pawłowski. Combining and Representing Logical Systems. In E. Moggi and G. Rosolini, editors, *Category Theory and Computer Science*, pages 177–196. Springer, 1997.
- [Nip02] T. Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *TYPES conference*, pages 259–278. Springer, 2002.
- [Nor05] U. Norell. The Agda WiKi, 2005. <http://wiki.portal.chalmers.se/agda>.
- [NPW02] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.
- [NSM01] P. Naumov, M. Stehr, and J. Meseguer. The HOL/NuPRL proof translator - a practical approach to formal interoperability. In *14th International Conference on Theorem Proving in Higher Order Logics*. Springer, 2001.
- [Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
- [PC93] L. Paulson and M. Coen. Zermelo-Fraenkel Set Theory, 1993. Isabelle distribution, ZF/ZF.thy.
- [Pfe00] F. Pfenning. Structural cut elimination: I. intuitionistic and classical logic. *Information and Computation*, 157(1-2):84–141, 2000.
- [PS99] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.
- [Rab08] F. Rabe. *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen, 2008. Available at <http://kwarc.info/frabe/Research/phdthesis.pdf>.
- [Rab10] F. Rabe. A Logical Framework Combining Model and Proof Theory. To be submitted, see [http://kwarc.info/frabe/Research/rabe\\_combining\\_09.pdf](http://kwarc.info/frabe/Research/rabe_combining_09.pdf), 2010.
- [RS09] F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, pages 40–48. ACM Press, 2009.
- [Soj10] K. Sojakova. Toward Mechanically Verifying Logic Translations, 2010. Master’s thesis, Jacobs University Bremen.

- [SW83] D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. In M. Karpinski, editor, *Fundamentals of Computation Theory*, pages 413–427. Springer, 1983.
- [Tar33] A. Tarski. Pojęcie prawdy w językach nauk dedukcyjnych. *Prace Towarzystwa Naukowego Warszawskiego Wydział III Nauk Matematyczno-Fizycznych*, 34, 1933. English title: The concept of truth in the languages of the deductive sciences.
- [Tar38] A. Tarski. Über Unerreichbare Kardinalzahlen. *Fundamenta Mathematicae*, 30:176–183, 1938.
- [Tar96] A. Tarlecki. Moving between logical systems. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *Recent Trends in Data Type Specifications. 11th Workshop on Specification of Abstract Data Types*, volume 1130 of *Lecture Notes in Computer Science*, pages 478–502. Springer Verlag, 1996.
- [TB85] A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985.
- [Try89] A. Trybulec. Tarski Grothendieck Set Theory. *Journal of Formalized Mathematics*, Axiomatics, 1989.
- [TV56] A. Tarski and R. Vaught. Arithmetical extensions of relational systems. *Compositio Mathematica*, 13:81–102, 1956.
- [vN25] J. von Neumann. Eine Axiomatisierung der Mengenlehre. *Journal für die reine und angewandte Mathematik*, 154:219–240, 1925.
- [Wie67] N. Wiener. A Simplification of the Logic of Relations. In J. van Heijenoort, editor, *From Frege to Gödel*, pages 224–227. Harvard Univ. Press, 1967.
- [Wie07] F. Wiedijk. Mizar’s Soft Type System. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 383–399. Springer, 2007.
- [Zer08] E. Zermelo. Untersuchungen über die Grundlagen der Mengenlehre I. *Mathematische Annalen*, 65:261–281, 1908. English title: Investigations in the foundations of set theory I.

# Formalizing Foundations of Mathematics

Mihnea Iancu and Florian Rabe  
Computer Science, Jacobs University Bremen, Bremen, Germany

## Abstract

Over the recent decades there has been a trend towards formalized mathematics, and a number of sophisticated systems have been developed to support the formalization process and mechanically verify its result. However, each tool is based on a specific foundation of mathematics, and formalizations in different systems are not necessarily compatible. Therefore, the integration of these foundations has received growing interest. We contribute to this goal by using LF as a foundational framework in which the mathematical foundations themselves can be formalized and therefore also the relations between them. We represent three of the most important foundations – Isabelle/HOL, Mizar, and ZFC set theory – as well as relations between them. The relations are formalized in such a way that the framework permits the extraction of translation functions, which are guaranteed to be well-defined and sound. Our work provides the starting point of a systematic study of formalized foundations in order to compare, relate, and integrate them.

## 1 Introduction

The 20<sup>th</sup> century saw significant advances in the field of foundations of mathematics stimulated by the discovery of paradoxes in naive set theory, e.g., Russell’s paradox of unlimited set comprehension. Several seminal works have redeveloped and advanced large parts of mathematics based on one coherent choice of foundation, most notably the Principia (Whitehead & Russell 1913) and the works by Bourbaki (Bourbaki 1964). Today various flavors of axiomatic set theory and type theory provide a number of well-understood foundations.

Given a development of mathematics in one fixed foundation, it is possible to give a fully formal language in which every mathematical expression valid in that foundation can be written down. Then mathematics can – in principle – be reduced to the manipulation of these expression, an approach called *formalism* and most prominently expressed in Hilbert’s program. Recently this approach has gained more and more momentum due to the advent of computer technology: With machine support, the formidable effort of formalizing mathematics becomes feasible, and the trust in the soundness of an argument can be reduced to the trust in the implementation of the foundation.

However, compared to “traditional” mathematics, this approach has the drawback that it heavily relies on the choice of one specific foundation. Traditional mathematics, on

the other hand, frequently and often crucially abstracts from and moves freely between foundations to the extent that many mathematical papers do not mention the exact foundation used. This level of abstraction is very difficult to capture if every statement is rigorously reduced to a fixed foundation. Moreover, in formalized mathematics, different systems implementing different (or even the same or similar) foundations are often incompatible, and no reuse across systems is possible.

But the high cost of formalizing mathematics makes it desirable to join forces and integrate foundational systems. Currently, due to the lack of integration, significant overlap and redundancies exist between libraries of formalized mathematics, which slows down the progress of large projects such as the formal proofs of the Kepler conjecture (Hales 2003).

Our contribution can be summarized as follows. Firstly, we introduce a new methodology for the formal integration of foundations: Using a logical framework, we formalize not only mathematical theories but also the foundations themselves. This permits formally stating and proving relations between foundations. Secondly, we demonstrate our approach by formalizing three of the most widely-used important foundations as well as translations between them. Our work provides the starting point of a formal library of foundational systems that complements the existing foundation-specific libraries and provides the basis for the systematic and formally verified integration of systems for formalized mathematics.

We begin by describing our approach and reviewing related work in Sect. 2. Then, in Sect. 3, we give an overview of the logical framework we use in the remainder of the paper. We give a new formalization of traditional mathematics based on ZFC set theory in Sect. 4. Then we formalize two foundations with particularly large formalized libraries: Isabelle/HOL (Nipkow, Paulson & Wenzel 2002) in Sect. 5 and Mizar (Trybulec & Blair 1985) in Sect. 6. We also give a translation from Isabelle/HOL into ZFC and sketch a partial translation from Mizar (which is stronger than ZFC) to ZFC. We discuss our work and conclude in Sect. 7.

Our formalizations span several thousand lines of declarations, and their descriptions are correspondingly simplified. The full sources are available at (Iancu & Rabe 2010).

## 2 Problem Statement and Related Work

Automath (de Bruijn 1970) and the formalization of Landau’s analysis (Landau 1930, van Benthem Jutting 1977) were the first major success of formalized mathematics. Since then a number of computer systems have been put forward and have been adopted to varying degrees to formalize mathematics such as LCF (Gordon, Milner & Wadsworth 1979), HOL (Gordon 1988), HOL Light (Harrison 1996), Isabelle/HOL (Nipkow et al. 2002), IMPS (Farmer, Guttman & Thayer 1993), Nuprl (Constable, Allen, Bromley, Cleaveland, Cremer, Harper, Howe, Knoblock, Mendler, Panangaden, Sasaki & Smith 1986), Coq (Coquand & Huet 1988), Mizar (Trybulec & Blair 1985), Isabelle/ZF (Paulson & Coen 1993), and the body of peer-reviewed formalized mathematics is growing (T. Hales and G. Gonthier and J. Harrison and F. Wiedijk 2008, Matuszewski 1990, Klein,

Nipkow & (eds.) 2004). A comparison of some formalizations of foundations in Automath, including ZFC and Isabelle/HOL, is given in (Wiedijk 2006).

The problem of interoperability and integration between these systems has received growing attention recently, and a number of connections between them have been established. (Obua & Skalberg 2006) and (McLaughlin 2006) translate between Isabelle/HOL and HOL Light; (Keller & Werner 2010) from HOL Light to Coq; and (Krauss & Schropp 2010) from Isabelle/HOL to Isabelle/ZF. The OpenTheory format (Hurd 2009) was designed as an interchange format for different implementations of higher order logic.

We call these translations *dynamically verified* because they have in common that they translate theorems in such a way that the target system reproves every translated theorem. One can think of the source system’s proof as an oracle for the target system’s proof search. This approach requires no reasoning about or trust in the translation so that users of the target system can reuse translated theorems without making the source system or the translation part of their trusted code base. Therefore, such translations can be implemented and put to use relatively quickly. It is no surprise that such translations are advanced by researchers working with the respective target system.

Still, dynamically verified translations can be unsatisfactory. The proofs of the source theorems may not be available because they only exist transiently when the source system processes a proof script. The source system might not be able to export the proofs, or they may be too large to translate. In that case, it is desirable to translate only the theorems and appeal to a general result that guarantees the soundness of the theorem translation.

However, the statement of soundness naturally lives outside either of the two involved foundations. Therefore, stating, let alone proving, the soundness of a translation requires a third formal system in which source and target system and the translation are represented. We call the third system a *foundational framework*, and if the soundness of a translation is proved in a foundational framework, we speak of a *statically verified* translation.

Statically verified translations are theoretically more appealing because the soundness is proved once and for all. Of course, this requires the additional assumptions that the foundational framework is consistent and that the representations in the framework are adequate. If this is a concern, the soundness proof should be *constructive*, i.e., produce for every proof in the source system a translated proof in the target system. Then users of the target system have the option to recheck the translated proof.

The most comprehensive example of a statically verified translation – from HOL to Nuprl (Naumov, Stehr & Meseguer 2001) – was given in (Schürmann & Stehr 2004). HOL and Nuprl proof terms are represented as terms in the framework Twelf (Harper, Honsell & Plotkin 1993, Pfenning & Schürmann 1999) using the judgments-as-types methodology. The translation and a constructive soundness proof are formalized as type-preserving logic programs in Twelf. The soundness is verified by the Twelf type checker, and the well-definedness – i.e., the totality and termination of the involved logic programs – is proved using Twelf.

In this work, we demonstrate a general methodology for statically verified translations. We formalize foundations as signatures in the logical framework Twelf, and we use the

LF module system's (Rabe & Schürmann 2009) translations-as-morphisms methodology to formalize translations between them as signature morphisms. This yields translations that are well-defined and sound by design and which are verified by the Twelf type checker. Moreover, they are constructive, and the extraction of translation programs is straightforward.

Our work can be seen as a continuation of the Logosphere project (Pfenning, Schürmann, Kohlhase, Shankar & Owre. 2003), of which the above HOL-Nuprl translation was a part. Both Logosphere and our work use LF, and the main difference is that we use the new LF module system to reuse encodings and to encode translations. Logosphere had to use monolithic encodings and used programs to encode translations. The latter were either Twelf logic program or Delphin (Poswolsky & Schürmann 2008) functional programs, and their well-definedness and termination was statically verified by Twelf and Delphin, respectively. Using the module system, translations can be stated in a more concise and declarative way, and the well-definedness of translations is guaranteed by the LF type theory.

There are some alternative frameworks in which foundations can be formalized: other variants of dependent type theory such as Agda (Norell 2005), type theories such as Coq based on the calculus of inductive constructions, or the Isabelle framework (Paulson 1994) based on polymorphic higher-order logic. All of these provide roughly comparably expressive module systems. We choose LF because the judgments-as-types and relations-as-morphisms methodologies are especially appropriate to formalize foundations and their relations.

We discuss related work pertaining to the individual foundations separately below.

### 3 The Edinburgh Logical Framework

The Edinburgh Logical Framework (Harper et al. 1993) (LF) is a formal meta-language used for the formalization of deductive systems. It is related to Martin-Löf type theory and the corner of the lambda cube that extends simple type theory with dependent function types and kinds. We will work with the Twelf (Pfenning & Schürmann 1999) implementation of LF and its module system (Rabe & Schürmann 2009).

The central notion of the LF type theory is that of a *signature*, which is a list  $\Sigma$  of *kinded type family* symbols  $a : K$  or *typed constant* symbols  $c : A$ . It is convenient to permit those to carry optional definitions, e.g.,  $c : A = t$  to define  $c$  as  $t$ . (For our purposes, it is sufficient to assume that these abbreviations are transparent to the underlying type theory, which avoids some technical complications. Of course, they are implemented more intelligently.)

LF *contexts* are lists  $\Gamma$  of typed variables  $x : A$ , i.e., there is no polymorphism. Relative to a signature  $\Sigma$  and a context  $\Gamma$ , the expressions of the LF type theory are *kinds*  $K$ , *kinded type families*  $A : K$ , and *typed terms*  $t : A$ . *type* is a special kind, and type families of kind *type* are called *types*.

We will use the concrete syntax of Twelf to represent expressions:

- The dependent function type  $\Pi_{x:A} B(x)$  is written  $\{x : A\} B x$ , and correspondingly for dependent function kinds  $\{x : A\} K x$ . As usual we write  $A \rightarrow B$  when  $x$  does not occur free in  $B$ .
- The corresponding  $\lambda$ -abstraction  $\lambda_{x:A} t(x)$  is written  $[x : A] t x$ , and correspondingly for type families  $[x : A] (B x)$ .
- As usual, application is written as juxtaposition.

Given two signatures  $\text{sig } S = \{\Sigma\}$  and  $\text{sig } T = \{\Sigma'\}$ , a *signature morphism*  $\sigma$  from  $S$  to  $T$  is a list of assignments  $c := t$  and  $a := A$ . They are called *views* in Twelf and declared as  $\text{view } v : S \rightarrow T = \{\sigma\}$ . Such a view is well-formed if

- $\sigma$  contains exactly one assignment for every symbol  $c$  or  $a$  that is declared in  $\Sigma$  without a definition,
- each assignment  $c := t$  assigns to the  $\Sigma$ -symbol  $c : A$  a  $\Sigma'$ -term  $t$  of type  $\bar{\sigma}(A)$ ,
- each assignment  $a := K$  assigns to the  $\Sigma$ -symbol  $a : K$  a  $\Sigma'$ -type family  $K$  of type  $\bar{\sigma}(K)$ .

Here  $\bar{\sigma}$  is the homomorphic extension of  $\sigma$  that maps all closed expressions over  $\Sigma$  to closed expressions over  $\Sigma'$ , and we will write it simply as  $\sigma$  in the sequel. The central result about signature morphisms (see (Harper, Sannella & Tarlecki 1994)) is that they preserve typing and  $\alpha\beta\eta$ -equality: Judgments  $\vdash_\Sigma t : A$  imply judgments  $\vdash_{\Sigma'} \sigma(t) : \sigma(A)$  and similarly for kinding judgments and equality.

Finally, the Twelf module system permits inclusions between signatures and views. If a signature  $T$  contains the declaration `include S`, then all symbols declared in (or included into)  $S$  are available in  $T$  via qualified names, e.g.,  $c$  of  $S$  is available as  $S.c$ . Our inclusions will never introduce name clashes, and we will write  $c$  instead of  $S.c$  for simplicity. Correspondingly, if  $S$  is included into  $T$ , and we have a view  $v$  from  $S$  to  $T'$ , a view from  $T$  to  $T'$  may include  $v$  via the declaration `include v`.

This yields the following grammar for Twelf where gray color denotes optional parts.

|               |  |
|---------------|--|
| Toplevel      | $G ::= \cdot   G, \text{sig } T = \{\Sigma\}   G, \text{view } v : S \rightarrow T = \{\sigma\}$ |
| Signatures    | $\Sigma ::= \cdot   \Sigma, \text{include } S   \Sigma, c : A = t   \Sigma, a : K = t$           |
| Morphisms     | $\sigma ::= \cdot   \Sigma, \text{include } v   \sigma, c := t   \sigma, a := A$                 |
| Kinds         | $K ::= \text{type}   \{x : A\} K$  |
| Type families | $A ::= a   A t   [x : A] A   \{x : A\} A$  |
| Terms         | $t ::= c   t t   [x : A] t   x$  |

We will sometimes omit the type of a bound variable if it can be inferred from the context. Moreover, we will frequently use implicit arguments: If  $c$  is declared as  $c : \{x : A\} B$  and the value of  $s$  in  $c s$  can be inferred from the context, then  $c$  may alternatively be declared as  $c : B$  (with a free variable in  $B$  that is implicitly bound) and used as  $c$  (where the argument to  $c$  inferred). We will also use fixity and precedence declarations in the style of Twelf to make applications more readable.

*Example 1* (Representation of FOL in LF). The following is a fragment of an LF signature for first-order logic that we will use later to formalize set theory:

```

sig FOL = {
    set   : type
    prop  : type
    ded   : prop → type           prefix 0

    ⇒     : prop → prop → prop      infix 3
    ∧     : prop → prop → prop      infix 2
    ∀     : (set → prop) → prop
    ≡     : set → set → prop       infix 4
    ⇔     : prop → prop → prop      infix 1
    =     : [a] [b] (a ⇒ b) ∧ (b ⇒ a)
    ∘_I   : ded A → ded B → ded A ∧ B
    ∘_{E_l} : ded A ∧ B → ded A
    ∘_{E_r} : ded A ∧ B → ded B
    ⇒_I   : (ded A → ded B) → ded A ⇒ B
    ⇒_E   : ded A ⇒ B → ded A → ded B
    ∃_I   : ({x : i} ded (F x)) → ded (∀ [x] F x)
    ∃_E   : ded (∀ [x] F x) → {c : i} ded (F c)
}

```

This introduces two types *set* and *prop* for sets and propositions, respectively, and a type family *ded* indexed by propositions. Terms *p* of type *ded F* represents proofs of *F*, and the inhabitation of *ded F* represents the provability of *F*. Higher-order abstract syntax is used to represent binders, e.g.,  $\forall([x : \text{set}] F x)$  represents the formula  $\forall x : \text{set}. F(x)$ . Equivalence is introduced as a defined connective.

Note that the argument of *ded* does not need brackets as *ded* has the weakest precedence. Moreover, by convention, the Twelf binders  $[]$  and  $\{\}$  always bind as far to the right as is consistent with the placement of brackets.

As examples for inference rules, we give natural deduction introduction and elimination rules conjunction and implication. Here *A* and *B* of type *prop* are implicit arguments whose types and values are inferred. For example, the theorem of commutativity of conjunction can now be stated as

$$\begin{aligned}
\text{comm\_conj} &: \text{ded}(A \wedge B) \Leftrightarrow (B \wedge A) \\
&= \wedge_I (\Rightarrow_I [p] \wedge_I (\wedge_{E_r} p) (\wedge_{E_l} p)) \\
&\quad (\Rightarrow_I [p] \wedge_I (\wedge_{E_r} p) (\wedge_{E_l} p))
\end{aligned}$$

The LF type system guarantees that the proof is correct.

## 4 Zermelo-Fraenkel Set Theory

In this section, we present our formalization of Zermelo-Fraenkel set theory. We give an overview over our variant of ZFC in Sect. 4.1 and describe its encoding in Sect. 4.2

and [4.3] Finally, we discuss related formalizations in Sect. [4.4].

## 4.1 Preliminaries

Zermelo-Fraenkel set theory (Zermelo 1908, Fraenkel 1922) (with or without choice) is the most common implicitly or explicitly assumed foundation of mathematics. It represents all mathematical objects as sets related by the binary  $\in$  predicate. Propositions are stated using an untyped first-order logic. The logic is classical, but we will take care to reason intuitionistically whenever possible.

There are a number of equivalent choices for the axioms of ZFC. Our axioms are

- Extensionality:  $\forall x \forall y (\forall z (z \in x \Leftrightarrow z \in y) \Rightarrow x = y)$
- Set existence:  $\exists x \text{ true}$  (This could be derived from the axiom of infinity, but we add it explicitly here to reduce dependence on infinity.),
- Unordered pairing:  $\forall x \forall y \exists a (\forall z (z = x \vee z = y) \Rightarrow z \in a)$
- Union:  $\forall X \exists a \forall z (\exists x (x \in X \wedge z \in x) \Rightarrow z \in a)$
- Power set:  $\forall x \exists a \forall z ((\forall t (t \in z \Rightarrow t \in x)) \Rightarrow z \in a)$
- Specification:  $\forall X \exists a (\forall z ((z \in X \wedge \varphi(z)) \Leftrightarrow z \in a))$  for a unary predicate  $\varphi$  (possibly containing free variables)
- Replacement:  $\forall a (\forall x (x \in a) \Rightarrow \exists^! y (\varphi(x, y)) \Rightarrow \exists b (\forall y (\exists x (x \in a \wedge \varphi(x, y)) \Leftrightarrow y \in b))$  for a binary predicate  $\varphi$  (possibly containing free variables) where  $\exists^!$  abbreviates the easily definably quantifier of unique existence
- Regularity:  $\forall x (\exists t (t \in x)) \Rightarrow (\exists y (y \in x \wedge \neg(\exists z (z \in x \wedge z \in y))))$ .
- Choice and infinity, which we omit here.

It is important to note that there are no first-order terms except for the variables. Specific sets (i.e., first-order constant symbols) and operations on sets (i.e., first-order function symbols) are introduced only as derived notions: A new symbol may be introduced to abbreviate a uniquely determined set. For example, the empty set  $\emptyset$  abbreviates the unique set  $x$  satisfying  $\forall y. \neg y \in x$ . Adding such abbreviations is conservative over first-order logic but cannot be formalized within the language of first-order.

## 4.2 Untyped Set Theory

Our Twelf formalization of ZFC uses three main signatures: *ZFC\_FOL* encodes first-order logic, *ZFC* encodes the first-order theory of ZFC, and finally *Operations* introduces the basic operations and their properties, most notably products and functions. The actual encodings (Iancu & Rabe 2010) comprise several hundred lines of Twelf declarations and are factored into a number of smaller signatures to enhance maintainability

and reuse. Therefore, our presentation here is only a summary. Moreover, to enhance readability, we will use more Unicode characters in identifiers here than in the actual encodings.

**First-Order Logic**  $ZFC\_FOL$  is an extension of the signature  $FOL$  given in Ex. 1. Besides the usual components of FOL encodings in LF (see e.g., (Harper et al. 1993)), we use two special features.

Firstly, we add the (definite) *description operator*  $\delta : \{F : set \rightarrow prop\} ded \exists^! ([x]F x) \rightarrow set$ , which encodes the mathematical practice of giving a name to a uniquely determined object. Here  $\exists^!$  is the quantifier of unique existence which is easily definable. Thus  $\delta$  takes a formula  $F(x)$  with a free variable  $x$  and a proof of  $\exists^! x.F(x)$  and returns a new set. The LF type system guarantees that  $\delta$  can only be applied after showing unique existence.  $\delta$  is axiomatized using the axiom scheme  $ax_\delta : ded F (\delta F P)$ ; from this we can derive irrelevance, i.e.,  $\delta F P$  returns the same object no matter which proof  $P$  is used.

Secondly, we add *sequential connectives* for conjunction and implication. In a sequential implication  $F \Rightarrow' G$ ,  $G$  is only considered if  $F$  is true, and similarly for conjunction. This is very natural in mathematical practice – for example, mathematicians do not hesitate to write  $x \neq 0 \Rightarrow' x/x = 1$  when  $/$  is only defined for non-zero dividers. All other connectives remain as usual.

Sequential implication and conjunction are formalized in LF as follows:

$$\begin{aligned}\wedge' &: \{F : prop\} (ded F \rightarrow prop) \rightarrow prop \\ \Rightarrow' &: \{F : prop\} (ded F \rightarrow prop) \rightarrow prop \\ \wedge'_I &: \{p : ded F\} ded G p \rightarrow F \wedge' [p] G p \\ \wedge'_{E_l} &: ded F \wedge' [p] G p \rightarrow ded F \\ \wedge'_{E_r} &: \{q : ded F \wedge' [p] G p\} ded G (\wedge' E_l q) \\ \Rightarrow'_I &: (\{p : ded F\} ded G p) \rightarrow ded F \Rightarrow' [p] G p \\ \Rightarrow_E &: ded F \Rightarrow' [p] G p \rightarrow \{p : ded F\} ded G p\end{aligned}$$

$\wedge'$  and  $\Rightarrow'$  are applied to two arguments, first a formula  $F$ , and then a formula  $G$  stated in a context in which  $F$  is true. This is written as, e.g.,  $F \wedge' [p] G p$  where  $p$  is an assumed proof of  $F$  that may occur in  $G$ . We will use  $F \wedge' G$  and  $F \Rightarrow' G$  as abbreviations when  $p$  does not occur in  $G$ , which yield the non-sequential cases. The introduction and elimination rules are generalized accordingly. Note that these sequential connectives do not rely on classicality.

In plain first-order logic, such sequential connectives would be useless as a proof cannot occur in a formula. But in the presence of the description operator, the proofs frequently occur in terms and thus in formulas.

**Set Theory** The elementhood predicate is encoded as  $\in : set \rightarrow set \rightarrow prop$  together with a corresponding infix declaration. The formalization of the axioms is straightforward, for example, the axiom of extensionality is encoded as:

$$ax\_exten : ded \forall [x] \forall [y] (\forall [z] z \in x \Leftrightarrow z \in y) \Rightarrow x \doteq y$$

It is now easy to establish the adequacy of our encoding in the following sense: Every well-formed closed LF-term  $s : set$  over  $ZFC$  encodes a unique set satisfying a certain predicate  $F$ . This is obvious because  $s$  must be of the form  $\delta F P$ . The inverse does not hold as there are models of set theory with more sets than can be denoted by closed terms.

**Basic Operations** We can now derive the basic notions of set theory and their properties: Using the description operator and the respective axioms, we can introduce defined Twelf symbols

$$\begin{aligned} empty &: set = \dots \\ uopair &: set \rightarrow set = \dots \\ bigunion &: set \rightarrow set = \dots \\ powerset &: set \rightarrow set = \dots \\ image &: (set \rightarrow set) \rightarrow set \rightarrow set = \dots \\ filter &: set \rightarrow (set \rightarrow prop) \rightarrow set = \dots \end{aligned}$$

such that  $empty$  encodes  $\emptyset$ ,  $uopair x y$  encodes  $\{x, y\}$ ,  $bigunion X$  encodes  $\bigcup X$ ,  $powerset X$  encodes  $\mathcal{P}X$ ,  $image f A$  encodes  $\{f(x) : x \in A\}$ , and  $filter A F$  encodes  $\{x \in A \mid F(x)\}$ .

For example, to define  $uopair$  we proceed as follows:

$$\begin{aligned} is\_uopair &: set \rightarrow set \rightarrow set \rightarrow prop \\ &= [x][y][a] (\forall [z] (z \doteq x \vee z \doteq y) \Leftrightarrow z \in a) \\ p\_uopair &: ded \exists^! (is\_uopair A B) \\ &= spec\_unique (shrink (\forall_E (\forall_E ax\_pairing A) B)) \\ uopair &: set \rightarrow set \rightarrow set = [x][y] \delta (is\_uopair x y) p\_uopair \end{aligned}$$

Here  $is\_uopair x y a$  formalizes the defining property  $a \doteq \{x, y\}$  of the new function symbol, and  $p\_uopair$  shows unique existence. The above uses two lemmas

$$\begin{aligned} shrink &: ded (\exists [X] \forall ([z] (\varphi z) \Rightarrow z \in X)) \\ &\rightarrow ded (\exists [x] \forall ([z] (\varphi z) \Leftrightarrow z \in x)) = \dots \\ spec\_unique &: ded (\exists [x] \forall ([z] (\varphi z) \Leftrightarrow z \in x)) \\ &\rightarrow ded \exists^! [x] \forall ([z] (\varphi z) \Leftrightarrow z \in x) = \dots \end{aligned}$$

$shrink$  expresses that if there is a set  $X$  that contains all the elements for which the predicate  $\varphi : set \rightarrow prop$  holds, then the set described by  $\varphi$  exists.  $spec\_unique$  expresses that if a predicate  $\varphi : set \rightarrow prop$  describes a set then that set exists uniquely. They can be proved easily using extensionality and specification.

**Advanced Operations** Then we can define the advanced operations on sets in the usual way. For example, the definition of binary union  $x \cup y = \bigcup \{x, y\}$  can be directly formalized as

$$\text{union} : \text{set} \rightarrow \text{set} \rightarrow \text{set} = [x][y] \text{ bigunion } (\text{uopair } x y)$$

We omit the definitions of singleton sets, ordered pairs, cartesian products, relations, partial functions, and functions. Our definitions are standard except for the ordered pair. We define  $(x, y) = \{\{x\}, \{\{y\}, \emptyset\}\}$ , which is similar to Wiener's definition (Wiener 1967) and different from the more common  $(x, y) = \{\{x, y\}, \{x\}\}$  due to Kuratowski. Our definition is a bit simpler to work with than Kuratowski pairs because it avoids the special case  $(x, x) = \{\{x\}\}$ .

$$\begin{aligned} \text{pair} &: \text{set} \rightarrow \text{set} \rightarrow \text{set} \\ &= [a][b] \text{ uopair } (\text{singleton } a) (\text{uopair } (\text{singleton } b) \text{ empty}) \end{aligned}$$

The difference with Kuratowski pairs is not significant as we immediately prove the characteristic properties of pairing and then never appeal to the definition anymore.

$$\begin{aligned} \text{conv}_{pi1} &: \text{ded pil } (\text{pair } X Y) \doteq X = \dots \\ \text{conv}_{pi2} &: \text{ded pi2 } (\text{pair } X Y) \doteq X = \dots \\ \text{conv}_{pair} &: \text{ded ispair } X \rightarrow \text{ded pair } (\text{pi1 } X) (\text{pi2 } X) \doteq X = \dots \end{aligned}$$

The proofs are technical but straightforward.

Finally, we can define function construction  $X \ni x \mapsto f(x)$  and application  $f(x)$  as

$$\begin{aligned} \lambda &: \text{set} \rightarrow (\text{set} \rightarrow \text{set}) \rightarrow \text{set} = [a][f] \text{ image } ([x] \text{ pair } x (f x)) a \\ @ &: \text{set} \rightarrow \text{set} \rightarrow \text{set} \\ &= [f][a] \text{ bigunion } (\text{image } \text{pi2 } (\text{filter } f ([x] (\text{pi1 } x) \doteq a))) \end{aligned}$$

where  $\lambda A f$  encodes  $\{(x, f(x)) : x \in A\}$ , and  $@ f x$  yields “the  $b$  such that  $(a, b) \in f$ ”. Application is defined for all sets: for example, it returns  $\emptyset$  if  $f$  is not defined for  $x$ .

Like for pairs, we immediately prove the characteristic properties, which are known as  $\beta\eta$ -conversion and extensionality in computer science. We never use other properties than these later on:

$$\begin{aligned} \text{conv}_{apply} &: \text{ded } X \in A \rightarrow \text{ded } @(\lambda A F) X \doteq F X = \dots \\ \text{conv}_{lambda} &: \text{ded } F \in (\Rightarrow A B) \rightarrow \text{ded } \lambda A ([x] @ F x) \doteq F = \dots \\ \text{func}_{ext} &: \text{ded } F \in (\Rightarrow A B) \rightarrow \text{ded } G \in (\Rightarrow A B) \rightarrow \\ &\quad (\{a\} \text{ ded } a \in A \rightarrow \text{ded } @F a \doteq @G a) \rightarrow \text{ded } F \doteq G = \dots \end{aligned}$$

Again we omit the straightforward proofs.

### 4.3 Typed Set Theory

**Classes as Types** A major drawback of formalizations of set theory is the complexity of reasoning about elementhood and set equality. It is well-known how to overcome these using typed languages, but in mathematical accounts of set theory, types are not primitive but derived notions. We proceed accordingly: The central idea is to use the predicate subtype  $\text{Elem } A = \{x : \text{set} \mid \text{ded } x \in A \text{ inhabited}\}$  to represent the set  $A$ . In fact, we can use the same approach to recover classes as a derived notion:  $\text{Class } F = \{x : \text{set} \mid \text{ded } F x \text{ inhabited}\}$  for any unary predicate  $F : \text{set} \rightarrow \text{prop}$ .

However, LF does not support predicate subtypes (for the good reason that it would make the typing relation undecidable). Therefore, we think of elements  $x$  of the class  $\{x \mid F(x)\}$  as pairs  $(x, P)$  where  $P : \text{ded } F x$  is a proof that  $x$  is indeed in that class. We encode this in LF as follows:

$$\begin{aligned}
\text{Class} &: (\text{set} \rightarrow \text{prop}) \rightarrow \text{type} \\
\text{celem} &: \{a : \text{set}\} \text{ ded } F a \rightarrow \text{Class } F \\
\text{cwhich} &: \text{Class } F \rightarrow \text{set} \\
\text{cwhy} &: \{a : \text{Class } F\} \text{ ded } (F (\text{cwhich } a)) \\
\\
\text{Elem} &: \text{set} \rightarrow \text{type} = [a] \text{ Class } [x] x \in a \\
\text{elem} &: \{a : \text{set}\} \text{ ded } a \in A \rightarrow \text{elem } A = [a] [p] \text{ celem } a p \\
\text{which} &: \text{elem } A \rightarrow \text{set} = [a] \text{ cwhich } a \\
\text{why} &: \{a : \text{elem } A\} \text{ ded } (\text{which } a) \in A = [a] \text{ cwhy } a
\end{aligned}$$

$\text{Class } F$  encodes  $\{x \mid F(x)\}$ ,  $\text{celem } x P$  produces an element of a class, and  $\text{cwhich } x$  and  $\text{cwhy } x$  return the set and its proof. The remaining declarations specialize these notions to the classes  $\{x \mid x \in A\}$ .

To axiomatize these, we use the additional axiom  $\text{eqwhich} : \text{ded } \text{cwhich } (\text{elem } X P) \doteq X$  as well as the following axiom for proof irrelevance

$$\text{proofirrel} : \{f : \text{ded } G \rightarrow \text{Class } A\} \text{ ded } \text{cwhich } (f P) \doteq \text{cwhich } (f Q)$$

which formalizes that two sets are equal if they only differ in a proof.

**Typed Operations** Using the types  $\text{Elem } A$ , we can now lift all the basic untyped operations introduced above to the typed level. In particular, we define typed quantifiers  $\forall^*$ ,  $\exists^*$ , typed equality  $\doteq^*$ , typed function spaces  $\Rightarrow^*$ , and booleans  $\text{bool}$  as follows.

Firstly, we define *typed quantifiers* such as  $\forall^* : (\text{elem } A \rightarrow \text{prop}) \rightarrow \text{prop}$ . In higher-order logic (Church 1940), such typed quantification can be defined easily using abstraction over the booleans. This is not possible in ZFC because the type  $\text{prop}$  is not a set itself, i.e., we have  $\text{prop} : \text{type}$  and not  $\text{prop} : \text{set}$ . If we committed to classical logic, we could use the set  $\text{bool} : \text{set}$  from below.

A natural solution is relativization as in  $\forall^* F := \forall[x] x \in A \Rightarrow' F x$  for  $F : \text{elem } A \rightarrow \text{prop}$ . However, an attempt to define typed quantification like this meets a subtle difficulty: In  $\forall^* F$ ,  $F$  only needs to be defined for elements of  $A$  whereas in  $\forall[x] x \in A \Rightarrow' F x$ ,  $F$  must be defined for all sets even though  $F x$  is intended to be ignored if  $x \notin A$ . Therefore, we use sequential connectives:

$$\begin{aligned}
\forall^* &: (\text{Elem } A \rightarrow \text{prop}) \rightarrow \text{prop} = [F] (\forall[x] x \in A \Rightarrow' [p] (F (\text{elem } x p))) \\
\exists^* &: (\text{Elem } A \rightarrow \text{prop}) \rightarrow \text{prop} = [F] (\exists[x] x \in A \wedge' [p] (F (\text{elem } x p)))
\end{aligned}$$

It is easy to derive the expected introduction and elimination rules for  $\forall^*$  and  $\exists^*$ .

Secondly, *typed equality* is easy to define:

$$\doteq^* : \text{Elem } A \rightarrow \text{Elem } A \rightarrow \text{prop} = [a] [b] (\text{which } a) \doteq (\text{which } b)$$

It is easy to see that all rules for  $\doteq$  can be lifted to  $\doteq^*$ .

Then, thirdly, we can define *function types* in the expected way:

$$\begin{aligned}
\Rightarrow^* & : \text{set} \rightarrow \text{set} \rightarrow \text{set} = [x][y] \text{ Elem } (x \Rightarrow y) \\
\lambda^* & : (\text{Elem } A \rightarrow \text{Elem } B) \rightarrow \text{Elem } (A \Rightarrow^* B) = \dots \\
@^* & : \text{Elem } (A \Rightarrow^* B) \rightarrow \text{Elem } A \rightarrow \text{Elem } B \\
& = [F][x] \text{ elem } (@(\text{which } F)(\text{which } x)) (\text{funcE } (\text{why } F)(\text{why } x)) \\
\text{beta} & : \text{ded } (@^* (\lambda^* [x] F x) A) \doteq^* F A = \dots \\
\text{eta} & : \text{ded } (\lambda^* [x] (@^* F x)) \doteq^* F = \dots
\end{aligned}$$

We omit the quite involved definitions and only mention that the typed quantifiers and thus the sequential connectives are needed in the definitions.

Finally, we introduce the set  $\{\emptyset, \{\emptyset\}\}$  of booleans and derive some important operations for them. In particular, these are the constants 0 and 1, a supremum operation on families of booleans, a variant of if-then-else where the then-branch (else-branch) may depend on the truth (falsity) of the condition, and a reflection function mapping propositions to booleans.

$$\begin{aligned}
\text{bool} & : \text{set} = \text{uopair } \text{empty } (\text{singleton } \text{empty}) \\
0 & : \text{Elem } \text{bool} = \dots \\
1 & : \text{Elem } \text{bool} = \dots \\
\text{sup} & : (\text{Elem } A \rightarrow \text{Elem } \text{bool}) \rightarrow \text{Elem } \text{bool} \\
\text{ifte} & : \{F : \text{prop}\} (\text{ded } F \rightarrow \text{Elem } A) \rightarrow (\text{ded } \neg F \rightarrow \text{Elem } A) \\
& \quad \rightarrow \text{Elem } A = \dots \\
\text{reflect} & : \text{Elem } \text{prop} \rightarrow \text{Elem } \text{bool}
\end{aligned}$$

The definition of the supremum operation is only possible after proving that  $\{\emptyset, \{\emptyset\}\} = \mathcal{P}\{\emptyset\}$ , which requires the use of excluded middle. (In fact, it is equivalent to it.) Similarly, *reflect* and *ifte* can only be defined in the presence of excluded middle. All other definitions in our formalization of ZFC are also valid intuitionistically.

#### 4.4 Related Work

Several formalizations of set theory have been proposed and developed quite far. Most notable are the encodings of Tarski-Grothendieck set theory in Mizar (Trybulec & Blair 1985, Trybulec 1989) and of ZF in Isabelle (Paulson 1994, Paulson & Coen 1993). The most striking difference with our formalization is that these employ sophisticated machine support with structured proof languages. Since there is no comparable machine support for Twelf, our encoding uses hand-written proof terms.

We chose LF because it permits a more elegant formalization: We use only  $\in$  as a primitive symbol and use a description operator to introduce names for derived concepts. This deviates from standard accounts of formalized mathematics and is in contrast to Mizar where primitive function symbols are used for singleton, unordered pair, and union, and to Isabelle/ZF where primitive function symbols are used for empty set, power set, union, infinite set, and replacement. But it corresponds more closely to mathematical practice, where the implicit use of a description operator is prevalent.

Our encoding depends crucially on dependent types. Description operators are also used in typed formalizations of mathematics such as HOL (Church 1940). They differ from ours by not taking a proof of unique existence as an argument. Consequently, they must assume the non-emptiness of all types and a global choice function. Other language features only possible in a dependently-typed framework are sequential connectives and our *ifte* construct. Connectives similar to our sequential ones are also used in PVS (Owre, Rushby & Shankar 1992) and in (de Nivelle 2010), albeit without proof terms occurring explicitly in formulas.

Moreover, using dependent types, we can recover typed reasoning as a derived notion. Here, our approach is similar to the one in Scunak (Brown 2006), and in fact our formalization of classes and typed reasoning is inspired by the one used in Scunak. Scunak uses a variant of dependent type theory specifically developed for this purpose: The symbols *set*, *prop*, and *Class*, and the axioms *eqwhich* and *proofirrel* are primitives of the type theory. This renders the formalization much simpler at the price of using a less elegant framework.

A compromise between our encoding and Scunak's would be an extension of the LF framework. For example, the dependent sum type  $\Sigma_{x:\text{set}}(\text{ded } F x)$  could be used instead of our *Class F*. Moreover, in (Lovas & Pfenning 2009) a variant of proof irrelevance is introduced for LF that might make our encoding more elegant.

## 5 Isabelle and Higher-Order Logic

### 5.1 Preliminaries

**Isabelle** Isabelle is a logical framework and generic LCF-style interactive theorem prover based on polymorphic higher-order logic (Paulson 1994). We will only consider the core language of Isabelle here – the *Pure* logic and basic declarations – and omit the module system and the structured proof language. We gave a comprehensive formalization of *Pure* and the Isabelle module system in (Rabe 2010).

The grammar for Isabelle is given in Fig. 1, which is a simplified variant of the one given in (Wenzel 2009).

|                 |  |
|-----------------|--|
| <i>con</i>      | $::= c :: \tau$  |
| <i>ax</i>       | $::= a : \varphi$  |
| <i>lem</i>      | $::= l : \varphi \text{ proof}$  |
| <i>typedecl</i> | $::= (\alpha_1, \dots, \alpha_n)t$   |
| <i>types</i>    | $::= (\alpha_1, \dots, \alpha_n)t = \tau$  |
| $\tau$          | $::= \alpha \mid (\tau, \dots, \tau)t \mid \tau \Rightarrow \tau \mid \text{prop}$                     |
| <i>term</i>     | $::= x \mid c \mid \text{term term} \mid \lambda x :: \tau.\text{term}$                                |
| $\varphi$       | $::= \varphi \Rightarrow \varphi \mid \bigwedge x :: \tau.\varphi \mid \text{term} \equiv \text{term}$ |
| <i>proof</i>    | $::= \dots$  |

Figure 1: Isabelle Grammar

An Isabelle theory is a list of declarations of typed constants  $c :: \tau$ , axioms  $a : \varphi$ , lemmas  $a : \varphi P$  where  $P$  proves  $\varphi$ , and  $n$ -ary type operators  $(\alpha_1, \dots, \alpha_n)t$  which may carry a definition in terms of the  $\alpha_i$ . Definitions for constants can be introduced as special cases of axioms, and we consider base types as nullary type operators.

Types  $\tau$  are formed from type variables  $\alpha$ , type operator applications  $(\tau_1, \dots, \tau_n)t$ , function types, and the base type  $prop$  of propositions. Terms are formed from variables, constants, application, and lambda abstraction. Propositions are formed from implication  $\Rightarrow$ , universal quantification  $\Lambda$  at any type, and equality on any type. (Wenzel 2009) does not give a grammar for proofs but lists the inference rules; they are  $\Lambda$  introduction and elimination,  $\Rightarrow$  introduction and elimination, reflexivity and substitution for equality,  $\beta$  and  $\eta$  conversion, and functional extensionality.

Constants may be polymorphic in the sense that their types may contain free type variables. When a polymorphic constant is used, Isabelle automatically infers the type arguments.

**HOL** The most advanced logic formalized in Isabelle is HOL (Nipkow et al. 2002). Isabelle/HOL is a classical higher-order logic with shallow polymorphism, non-empty types, and choice operator (Church 1940, Gordon & Pitts 1993).

Isabelle/HOL uses the same types and function space as Isabelle. But it introduces a type  $bool$  for HOL-propositions (i.e., booleans since HOL is classical) that is different from the type  $prop$  of Isabelle-propositions. The coercion  $Trueprop : bool \Rightarrow prop$  is used as the Isabelle truth judgment on HOL propositions. HOL declares primitive constants for implication, equality on all types, definite and indefinite description operator  $some x : \tau.P$  and  $the x : \tau.P$  for a predicate  $P : \tau \Rightarrow bool$ . Furthermore, HOL declares a polymorphic constant *undefined* of any type and an infinite base type  $ind$ , which we omit in the following. Based on these primitives and their axioms, simply-typed set theory is developed by purely definitional means.

Going beyond the Isabelle framework, Isabelle/HOL also supports Gordon/HOL-style type definitions using representing sets. A set  $A$  on the type  $\tau$  is given by its characteristic function, i.e.,  $A : \tau \Rightarrow bool$ . An Isabelle/HOL type definition is of the form  $(\alpha_1, \dots, \alpha_n)t = A P$  where  $P$  and  $A$  contain the variables  $\alpha_1, \dots, \alpha_n$  and  $P$  proves that  $A$  is non-empty. If such a definition is in effect,  $t$  is an additional type that is axiomatized to be isomorphic to the set  $A$ .

## 5.2 Formalizing Isabelle/HOL

**Isabelle** Our formalization of Isabelle follows the one we gave in (Rabe 2010). We declare an LF signature *Pure* for the inner syntax of Isabelle, which declares symbols for all primitives that can occur in expressions. *Pure* is given in Fig. 2.

This yields a straightforward structural encoding function  $\Gamma \dashv$  that acts as described in Fig. 3. Similar encodings are well-known for LF, see e.g., (Harper et al. 1993). The only subtlety is the case of polymorphic constant applications  $c t_1 \dots t_n$  where the type of  $c$  contains type variables  $\alpha_1, \dots, \alpha_m$ . Here we need to infer the

```
sig S = {
  include Pure
  ΓΣ
}
```

```

sig Pure = {
  tp      : type
  ⇒      : tp → tp → tp
          infix 0
  tm      : tp → type
          prefix 0
  λ       : (tm A → tm B) → tm (A ⇒ B)
  @       : tm (A ⇒ B) → tm A → tm B
          infix 1000

  prop   : tp
  ∧      : (tm A → tm prop) → tm prop
          infix 1
  ⇒⇒    : tm prop → tm prop → tm prop
          infix 2
  ≡      : tm A → tm A → tm prop
          infix 2

  ⊢      : tm prop → type
          prefix 0
  ∧I     : (x : tm A ⊢ (B x)) → ⊢ ∧([x] B x)
  ∧E     : ⊢ ∧([x] B x) → {x : tm A} ⊢ (B x)
  ⇒I     : (⊢ A → ⊢ B) → ⊢ A ⇒ B
  ⇒E     : ⊢ A ⇒ B → ⊢ A → ⊢ B
  refl   : ⊢ X ≡ X
  subs   : {F : tm A → tm B} ⊢ X ≡ Y → ⊢ F X ≡ F Y
  exten  : ({x : tm A} ⊢ (F x) ≡ (G x)) → ⊢ λF ≡ λG
  beta   : ⊢ (λ[x : tm A] F x) @ X ≡ F X
  eta    : ⊢ λ ([x : tm A] F @ x) ≡ F
}

}

```

Figure 2: LF Signature for Isabelle

types  $\tau_1, \dots, \tau_m$  at which  $c$  is applied, and put  $\lceil c t_1 \dots t_n \rceil = (c \lceil \tau_1 \rceil \dots \lceil \tau_m \rceil) @ \lceil t_1 \rceil \dots @ \lceil t_n \rceil$ . Polymorphic axioms and lemmas occurring in proofs are treated accordingly. Finally, an Isabelle theory  $S = \Sigma$  is represented as shown on the right where  $\lceil \Sigma \rceil$  is defined declaration-wise according to Fig. 4

```

sig HOL = {
  include Pure
  bool      : tp
  trueprop  : tm bool ⇒ prop
  eps       : tm (A ⇒ bool) ⇒ A
  ≡         : tm A ⇒ A ⇒ bool
  set       : tp → tp = [a] a ⇒ bool
  nonempty  : (tm set A) → type = ...
  typedef   : {s : tm set A} nonempty s → tp
  Rep       : tm (typedef S P) ⇒ A
  Abs       : tm A ⇒ (typedef (S : tm set A) P)
}

```

Figure 5: LF Signature for HOL

| Expression | Isabelle   | LF  |
|------------|--|---|
| type       | $\tau$   | $\Gamma t^\neg : tp$                                |
| term       | $t :: \tau$  | $\Gamma t^\neg : tm \Gamma \tau^\neg$               |
| proof      | $P$ proving $\varphi$                                    | $\Gamma P^\neg : \vdash \Gamma \varphi^\neg$        |
|            | containing type variables<br>$\alpha_1, \dots, \alpha_m$ | in context<br>$\alpha_1 : tp, \dots, \alpha_m : tp$ |

Figure 3: Encoding of Expressions

| Declaration     | Isabelle  | LF  |
|-----------------|---|---|
| type operator   | $(\alpha_1, \dots, \alpha_n) t$   | $t : tp \rightarrow \dots \rightarrow tp \rightarrow tp$  |
| type definition | $(\alpha_1, \dots, \alpha_n) t = \tau$                                  | $t : tp \rightarrow \dots \rightarrow tp \rightarrow tp$<br>$= [\alpha_1] \dots [\alpha_n] \tau$                                  |
| constant        | $c :: \tau, \quad \alpha_1, \dots, \alpha_m \text{ in } \tau$           | $c : tp \rightarrow \dots \rightarrow tp \rightarrow tm \Gamma \tau^\neg$   |
| axiom           | $a : \varphi, \quad \alpha_1, \dots, \alpha_m \text{ in } \tau$         | $a : tp \rightarrow \dots \rightarrow tp \rightarrow \vdash \Gamma \varphi^\neg$  |
| lemma           | $l : \varphi P, \quad \alpha_1, \dots, \alpha_m \text{ in } \varphi, P$ | $l : tp \rightarrow \dots \rightarrow tp \rightarrow \vdash \Gamma \varphi^\neg$<br>$= [\alpha_1] \dots [\alpha_m] \Gamma P^\neg$ |

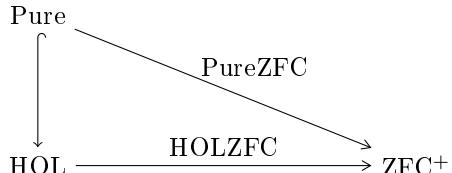
Figure 4: Encoding of Declarations

**HOL** Since HOL is an Isabelle theory, its LF-encoding follows immediately from the definition above. The fragment arising from translating some of the primitive declarations of HOL is given in the upper part of the signature *HOL* in Fig. 5. For example, *eps* is the choice operator. The lower part gives some of the additional declarations needed to encode HOL-style type definitions. The central declaration is *typedef*, which takes a set *S* on the type *A* and a proof that *S* is nonempty and returns a new type, say *T*. *Rep* and *Abs* translate between *A* and *T*, we refer to (Wenzel 2009) for details.

### 5.3 Interpreting Isabelle/HOL in ZFC

We formalize the relation between Isabelle/HOL and ZFC by giving two views *PureZFC* and *HOLZFC* from *Pure* and *HOL*, respectively, to *ZFC* as shown on the right. These formalize the standard set-theoretical semantics of higher-order logic.

*ZFC* arises from *ZFC* by adding a global choice function  $choice : \{A : Class \text{ nonempty}\} (Elem (chwich A))$  that produces an element of a non-empty set *A*. This is stronger than the axiom of choice (which merely states the existence of such an element) but needed to interpret the choice operators of HOL.



**Isabelle** The general structure of the translation is given in Fig. 6 and the view in Fig. 7. Types are mapped to non-empty sets, terms to elements, in particular propositions to booleans, and proofs of  $\varphi$  to proofs of  $PureZFC(\varphi) \doteq^* 1$ . These invariants are encoded (and guaranteed) by the assignments to  $tp$ ,  $tm$ ,  $prop$ , and  $\vdash$  in  $PureZFC$ . It is tempting to map Isabelle propositions to ZFC propositions rather than to booleans. However, in Isabelle,  $prop$  is a normal type and thus must be interpreted as a set. An alternative would be to map  $prop$  to a set representing intuitionistic truth values rather than classical ones, but we omit that for simplicity. (Due to our use of a standard model, we cannot expect completeness anyway.)

| Isabelle/HOL          | ZFC  |
|-----------------------|--|
| $\tau : tp$           | $PureZFC(\tau) : Class\ nonempty.$                       |
| $t : tm\ \tau$        | $PureZFC(t) : Elem\ (cwhich\ PureZFC(\tau)).$            |
| $\varphi : tm\ prop$  | $PureZFC(\varphi) : Elem\ (cwhich\ boolne).$             |
| $P : \vdash\ \varphi$ | $PureZFC(P) : ded\ PureZFC(\varphi) \doteq^* (bbne\ 1).$ |

Figure 6: Isabelle/HOL Declarations in ZFC

The case for terms  $t$  is a bit tricky: Since  $\tau$  is interpreted as an element of *Class nonempty*, we first have to apply *cwhich* to obtain a set. Then we apply *Elem* to this set to obtain the type of its elements. Similarly, *prop* cannot be mapped directly to *Elem bool*. Instead, we have to introduce *boolne* : *Class nonempty* which couples *bool* with the proof that it is a non-empty set. Therefore, we also have to define the auxiliary functions *bbne* : *Elem bool*  $\rightarrow$  *Elem* (*cwhich boolne*) and *bneb* : *Elem* (*cwhich boolne*)  $\rightarrow$  *Elem* *bool* to convert back and forth. These technicalities indicate a drawback of our – otherwise perfectly natural – representation of classes. Different representations that separate the mapping of types to sets from the proofs of non-emptiness may prove more scalable, but would require a more sophisticated framework.

The remaining cases are straightforward. For example,  $\Rightarrow$  must be mapped to a *ZFC* expression that takes two arguments of type *Class nonempty* and returns another, i.e., must respect the invariants above.

**HOL** Similarly, we obtain a view from *HOL* to *ZFC*, a fragment of which is shown in Fig. 8. *HOL* booleans are mapped to *ZFC* booleans so that *trueprop* is mapped to the identity. The choice operator *eps* is interpreted using *ifte* and *choice*. Note that

view  $PureZFC : Pure \rightarrow ZFC = \{$   
 $tp := Class\ nonempty$   
 $tm := elem$   
 $prop := boolne$   
 $\vdash := [x] ded\ x \doteq^* 1$   
 $\Rightarrow := \Rightarrow^*$   
 $\lambda := \lambda^*$   
 $@ := @_*$   
 $\wedge := \forall^*$   
 $\implies := \Rightarrow$   
 $\equiv := \doteq^*$   
 $:$   
 $\}$

Figure 7: Interpreting Pure in ZFC

in the given Twelf terms we elide some bookkeeping proof steps. The *then*-branch uses *elem* (*filter f*) *P* to construct an element of *Class nonempty*, to which then *choice* is applied. In both cases, *P* must use the assumption *p* that the condition of the *ifte*-split is true.

*typedef s p* is interpreted using *filter* according to *s*. Thus, type definitions using sets on *A* are interpreted as subsets of *A* in the expected way. The proof *p* is used to obtain an element of *Class nonempty*.

```

view HOLZFC : HOL → ZFC = {
  include PureZFC
  bool      :=  bool
  trueprop  :=  [x] x
  ≡         :=  λ*([x](λ*([y]bbne(reflect(x ≡* y)))))
  eps       :=  [f : Elem (A ⇒* bool)] ifte (nonempty (filter f))
               ([p] (choice (elem (filter f) p)))
               ([p] choice A)
  :
  typedef   :=  [s : Elem (A ⇒ bool)] [p] celem (filter ([x] s @ x ≡* 1)) p
  :
}

```

Figure 8: Interpreting HOL in ZFC

## 5.4 Related Work

Our formalization of Isabelle is a special case of the one we gave in (Rabe 2010). There we also cover the Isabelle module system. Together with the formalization of HOL given here, we now cover interpretations of Isabelle locales in terms of Isabelle/HOL. This is interesting because if Isabelle locales are seen as logical theories and HOL as a foundation of mathematics, then interpretations can be seen as models.

Formalizations of HOL in logical frameworks have been given in (Pfenning et al. 2003) using LF and of course in Isabelle itself (Nipkow et al. 2002). Ours appears to be the first formalization of Isabelle and HOL and the meta-relation between them. Moreover, we do not know any other formalizations of HOL-style type definitions in a formal framework – even in the Isabelle/HOL formalization, the type definitions are not expressed exclusively in terms of the Pure meta-language.

Our semantics of Isabelle/HOL does not quite follow the one given in (Gordon & Pitts 1993). There, individual models provide a set  $\mathcal{U}$  of sets, and every type is interpreted as an element of  $\mathcal{U}$ . Models must provide further structure to interpret HOL type constructors, in particular a choice function on  $\mathcal{U}$ . Our semantics can be seen as a single model where the set theoretical universe is used instead of  $\mathcal{U}$ . Consequently, our model is not a set itself and thus not a model in the sense of (Gordon & Pitts 1993), but every individual model in that sense is subsumed by ours.

Independent of our work, a similar semantics of Isabelle/HOL is given in (Krauss & Schropp 2010). They translate Isabelle/HOL to Isabelle/ZF where the interpretation of Pure is simply the identity. Their semantics is given as a target-trusting implementation rather than formalized in a framework. They also use the full set-theoretical universe and a global choice function. An important difference is the treatment of non-emptiness: They assume that interpretations for all type constructors are given that respect non-emptiness; then they can interpret all types as sets (which will always be non-empty) and only have to relativize universal quantifiers over types to quantifiers over non-empty sets. Our translation is more complicated in that respect because it uses *Class nonempty* to guarantee the non-emptiness.

## 6 Mizar and Tarski-Grothendieck Set Theory

### 6.1 Preliminaries

**Mizar** At its core, Mizar (Trybulec & Blair 1985) is an implementation of classical first-order logic. However, it is designed to be used with a single theory: set theory following the Tarski-Grothendieck axiomatization (Tarski 1938, Bourbaki 1964) (TG). Consequently, Mizar is strongly influenced by its representation of TG. Like Isabelle, it includes a semi-automated theorem prover and a structured proof language.

Mizar/TG is notable for being the only major system for the formalization of mathematics that is based on set theory. Types are only introduced as a means of efficiency and clarity but not as a foundational commitment. Moreover, the Mizar Mathematical Library is one of the largest libraries of formalized mathematics containing over 50000 theorems and 9500 definitions.

Mizar’s logic is an extension of classical first-order logic with second-order axiom schemes. The proof system is Jaskowski-style natural deduction (Jaśkowski 1934). Contrary to the LCF style implementations of HOL and to our ZFC, which try to use a small set of primitives, Mizar features a rich ontology of primitive mathematical objects, types, and proof principles.

In particular, the type *set* of terms (i.e., sets in Mizar/TG) can be refined using a complex type system, see, e.g., (Wiedijk 2007). The basic types are called *modes*, and while they are semantically predicate subsorts (i.e., classes in Mizar/TG), they are technically primitive in Mizar. Modes can be further refined by *attributes*, which are predicates on a type. These two refinement relations generate a subtype relation between type expressions, called type *expansion*. Both modes and attributes may take arguments, which makes Mizar dependently-typed. Mizar enforces the non-emptiness of all types, and all mode definitions and attribute applications induce the respective proof obligations.

The notion of typed first-order functions between types, called *functors*, is primitive. Function definitions may be implicit, in which case they induce proof obligations for well-definedness.

This expressivity makes theorems and proofs written in Mizar relatively easy to read

but makes it hard to represent Mizar itself in a logical framework. We will use the grammar given in Fig. 9, which is a substantially simplified variant of the one given in (Mizar 2009). Here  $\dots$  and  $*$  denote possibly empty repetition.

|                    |       |   |
|--------------------|-------|---|
| <i>Article</i>     | $::=$ | <i>Article-Name</i> $^*$ <i>Text-Proper</i>   |
| <i>Text-Proper</i> | $::=$ | ( <i>Block</i>   <i>Theorem</i> ) $^*$  |
| <i>Block</i>       | $::=$ | <i>definition let</i> ( $x$ be $\vartheta$ ) $^*$ <i>Definition end</i>   |
| <i>Definition</i>  | $::=$ | <i>Mode</i>   <i>Functor</i>   <i>Attribute</i>   |
| <i>Mode</i>        | $::=$ | <i>mode</i> $M$ of $x_1, \dots, x_n$ is $\vartheta$<br>  <i>mode</i> $M$ of $x_1, \dots, x_n \rightarrow \vartheta$ means $\alpha$<br><i>existence proof</i>      |
| <i>Attribute</i>   | $::=$ | <i>attr</i> $x$ is $(x_1, \dots, x_n)V$ means $\alpha$  |
| <i>Functor</i>     | $::=$ | <i>func</i> $f(x_1, \dots, x_n)$ equals $t$<br>  <i>func</i> $f(x_1, \dots, x_n) \rightarrow \vartheta$ means $\alpha$<br><i>existence proof uniqueness proof</i> |
| <i>Theorem</i>     | $::=$ | <i>theorem</i> $T : \alpha$ <i>proof</i>  |
| $t$                | $::=$ | $x$   $f(t_1, \dots, t_n)$  |
| $\alpha$           | $::=$ | $t$ is $\vartheta$   $t$ in $t$   $\alpha \& \alpha$   $\text{not } \alpha$   $t = t$<br>  <i>for</i> $x$ be $\vartheta$ holds $\alpha$                           |
| $\vartheta$        | $::=$ | <i>Adjective</i> $^*$ <i>Radix</i>  |
| <i>Adjective</i>   | $::=$ | $(t_1, \dots, t_n)V$   <i>non</i> $(t_1, \dots, t_n)V$  |
| <i>Radix</i>       | $::=$ | $M$ of $t_1, \dots, t_n$  |
| <i>proof</i>       | $::=$ | $\dots$   |

Figure 9: Mizar Grammar

A Mizar article starts with one import clause for every kind of declaration to import from other articles. Instead, we only permit cumulative imports of whole articles. This is followed by a list of definitions and theorems. We only permit mode, functor, and attribute definitions. Predicate definitions and schemes could be added easily.

All three kinds of definitions introduce a new symbol, which takes a list of typed term arguments  $x_i$ . The type  $\vartheta_i$  of  $x_i$  must be given by a *let* declaration or defaults to the type *set*. Mode definitions define  $M$  of  $x_1, \dots, x_n$  either explicitly as the type  $\vartheta(x_1, \dots, x_n)$  or implicitly as the type of sets *it* of type  $\vartheta$  satisfying  $\alpha(it, x_1, \dots, x_n)$ . In the latter case, non-emptiness must be proved. Similarly, functor definitions define  $f(x_1, \dots, x_n)$  either explicitly as  $t(x_1, \dots, x_n)$  or implicitly as the object *it* of type  $\vartheta$  satisfying  $\alpha(it, x_1, \dots, x_n)$ . In the latter case, well-definedness, i.e., existence and uniqueness, must be proved. Finally, attribute definitions define  $(x_1, \dots, x_n)V$  as the unary predicate on the type of  $x$  given by  $\alpha(x, x_1, \dots, x_n)$ .

Terms  $t$  and formulas  $\alpha$  are formed in the usual way, and we omit the productions for *proof* terms. *in* and *is* are used for elementhood in a set or a type, respectively. Finally, types are formed by providing a list of possibly negated adjectives on a mode. In Mizar, these types must be proved to be non-empty before they can be used, which we will omit here.

**Tarski-Grothendieck Set Theory** TG is similar to ZFC but uses Tarski’s axiom asserting that for every set there is a universe containing it. It implies the axioms of infinity, choice, power set, and large cardinals. Mizar/TG is defined in the Mizar article [Tarski \(Trybulec 1989\)](#), which contains primitives for elementhood, singleton, unordered pair, union, the Fraenkel scheme, the Tarski axiom, as well as a definition of ordered pairs following Kuratowski.

## 6.2 Formalizing Mizar and TG Set Theory

```

sig Mizar = {
  tp      : type
  prop    : type
  proof   : prop → type
  be      : tp → type
  set     : tp
  is      : be T → tp → prop
  in      : be T → be T' → prop
  not    : prop → prop
  and    : prop → prop → prop
  eq     : be T → be T' → prop
  :
  for    : (be T → prop) → prop
  :
  func   : {f : be T → prop} (proof ex [x] f x) →
            proof for [x] for [y] (f x and f y) implies x eq y → be T
  funcprop : {F} {Ex} {Unq} proof F (func F Ex Unq)
  attr   : tp → type = [t] (be t → prop)
  adjective : {t : tp} attr t → tp
  adjI   : {x : be X} (proof A x) → be (adjective X A)
  adjE   : {x : be (adjective X A)} be X
  adjE'  : {x : be (adjective X A)} proof A (adjE x)
  :
}
  
```

Figure 10: LF Signature for Mizar

**Mizar** The LF signature that encodes Mizar’s logic is given in Fig. 10, where we omit the declarations of definable constants, such as equivalence *iff* and existential quantifier *ex*. The general form of the encoding of Mizar expressions in LF is given in Fig. 11. Mizar types, formulas, and proofs of  $F$  are represented as LF terms of the types  $tp$ ,  $prop$ , and  $proof \sqcap F \sqcup$ , respectively. The judgment *expand* encodes Mizar’s type expansion relation.

Mizar’s use of a type system within an untyped foundation is hard to represent in a logical framework. We mimic it by using an auxiliary type constructor *be* with the intended meaning that  $t : be T$  encodes a Mizar term  $t$  of type  $T$ . Consequently, if  $T$

expands to  $T'$ , terms of type  $T$  must be explicitly cast to obtain terms of type  $T'$  by applying *cast*.

Attributes on a type  $\vartheta$  are represented as LF terms of type  $attr \vartheta$ . In effect, they are represented as LF functions  $be \vartheta \rightarrow prop$ . A type  $\vartheta = A_1 \dots A_m R$  is encoded as  $adjective(\dots(adjective R A_m) \dots) A_1$ . Attributes  $A = (t_1, \dots, t_n)V$  are encoded as  $V \lceil t_1 \rceil \dots \lceil t_n \rceil$ . Finally types  $M$  of  $t_1, \dots, t_n$  (radix types in Mizar) are encoded as  $M \lceil t_1 \rceil \dots \lceil t_n \rceil$ .

To that, we add LF constant declarations that represent the primitive formula and proof constructors of Mizar's first-order logic. For formulas and proofs, this is straightforward, and the only subtlety is to identify exactly which constructors are primitive. For example, *or* and *imp* are defined using *and* and *not*. We omit the constructors for type expansion. This induces an encoding of Mizar terms, types, formulas, and proofs as LF terms. The only remaining subtlety is that applications of *cast* must be inserted whenever the well-formedness of a type depends on the type expansion relations.

| Expression     | Mizar                               | LF  |
|----------------|-------------------------------------|---|
| type           | $\vartheta$                         | $\lceil \vartheta \rceil : tp$                                      |
| formula        | $\alpha$                            | $\lceil \alpha \rceil : prop$                                       |
| proof          | $P$ proving $\alpha$                | $\lceil P \rceil : proof \lceil \alpha \rceil$                      |
| typed term     | $t$ be $\vartheta$                  | $\lceil t \rceil : be \lceil \vartheta \rceil$                      |
| type expansion | $\vartheta$ expands to $\vartheta'$ | $expand \lceil \vartheta \rceil \lceil \vartheta' \rceil$ inhabited |

Figure 11: Encoding of Expressions

Then we can represent Mizar declarations according to Fig. 12. Explicit functor and mode definitions are represented easily as defined LF constants. Implicit definitions are represented using special constants *func* and *mode*. *func* ( $[x : be \vartheta] \alpha x$ ) *ex unq* encodes the uniquely existing object of type  $\vartheta$  that satisfies  $\alpha$ . Similar to  $\delta$  in our ZFC encodings, it takes proofs of existence and uniqueness as arguments. *mode* ( $[x : be \vartheta] \alpha x$ )  $P$  encodes the necessarily non-empty subtype of  $\vartheta$  containing the objects satisfying  $\alpha$ . Attribute definitions are encoded easily. In all three cases, the arguments  $x_1, \dots, x_n$  of Mizar functors/modes/attributes are represented directly as LF arguments. Finally, theorems are encoded in the same way as for Isabelle.

Finally, we can encode a Mizar article  $Art_1, \dots, Art_n$  *TP* in file *A* as the following LF signature where the *Text-Proper* part *TP* is encoded declaration-wise.

```
sig A = {
  include Mizar
  include Art1
  :
  include Artn
  "TP"
}
```

**Adequacy** Intuitively, our Mizar encoding should be adequate in the sense that Mizar articles that stay within our simplified grammar are well-formed in Mizar iff their encoding is well-formed in LF.

We cannot state or even prove the adequacy because there is no reference semantics of Mizar that would be rigorous and complete

| Mizar  | LF   |
|--|--|
| <code>let <math>x_i</math> be <math>\vartheta_i</math></code>  |  |
| <code>mode <math>M</math> of <math>x_1, \dots, x_n</math> is <math>\vartheta</math></code>                       | $M : \{x_1 : be^\top \vartheta_1\} \dots \{x_n : be^\top \vartheta_n\} tp$<br>$= [x_1] \dots [x_n]^\top \vartheta^\top$  |
| <code>mode <math>M</math> of <math>x_1, \dots, x_n \rightarrow \vartheta</math> means <math>\alpha</math></code> | $M : \{x_1 : be^\top \vartheta_1\} \dots \{x_n : be^\top \vartheta_n\} tp$<br>$= [x_1] \dots [x_n] mode ([it]^\top \alpha^\top)^\top P^\top$                       |
| <code>func <math>f(x_1, \dots, x_n)</math> equals <math>t</math></code>  | $f : \{x_1 : be^\top \vartheta_1\} \dots \{x_n : be^\top \vartheta_n\} be^\top \vartheta$<br>$= [x_1] \dots [x_n]^\top t^\top$                                     |
| <code>func <math>f(x_1, \dots, x_n) \rightarrow \vartheta</math> means <math>\alpha</math></code>                | $f : \{x_1 : be^\top \vartheta_1\} \dots \{x_n : be^\top \vartheta_n\} be^\top \vartheta$<br>$= [x_1] \dots [x_n] func ([it]^\top \alpha^\top)^\top P^\top Q^\top$ |
| <code>let <math>x</math> be <math>\vartheta</math></code>  | $V : \{x_1 : be^\top \vartheta_1\} \dots \{x_n : be^\top \vartheta_n\} attr^\top \vartheta$<br>$= [x_1] \dots [x_n] ([x]^\top \alpha^\top)$                        |
| <code>attr <math>x</math> is <math>(x_1, \dots, x_n)V</math> means <math>\alpha</math></code>                    | $T : proof^\top \alpha^\top =^\top P^\top$   |
| <code>theorem <math>T : \alpha</math></code>   |  |

Figure 12: Encoding of Declarations

enough for that. This is partially due to the fact that Mizar is justified more through mathematical intuition than through a formal semantics.

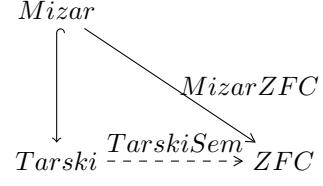
**TG Set Theory** The encoding of TG set theory given in Fig. 13 is rather straightforward. The use of LF’s  $\{\}$  binder for Mizar’s axiom schemes is the only subtlety. The definitions for *singleton*, *uopair*, and *union* are given using *func*, and their existence and uniqueness conditions are stated as axioms. We only give the case for *singleton*. The Tarski axiom is easy to encode but requires some auxiliary definitions.

### 6.3 Interpreting Mizar/Tarski in ZFC

Similar to the interpretation of Isabelle/HOL in ZFC, we give corresponding views for Mizar. Here the view from *Tarski* to *ZFC* is dashed because it is partial: It omits the Tarski axiom, which goes beyond ZFC.

**Mizar** The general idea of the interpretation of Mizar in ZFC is given in Fig. 14. In particular, a type  $\vartheta$  is interpreted as a unary predicate (the intensional description of  $\vartheta$ ), and the auxiliary type  $be \vartheta$  as the class of sets in  $\vartheta$  (the extensional description of  $\vartheta$ ). Technically, we should interpret types as non-empty predicates, i.e., as pairs of a predicate and an existence proof. We avoid that because it would complicate the encoding even more than in the case of Isabelle/HOL. This is possible because no part of our restricted Mizar language relies on the non-emptiness of type.

Type expansion is interpreted as a subclass relationship, and the interpretation of *cast* maps a set to itself but treated as an element of a different class. This is formalized by the first declarations in the view *MizarZFC* in Fig. 15.



```

sig Tarski = {
  include Mizar
  :
  singletonex : {y : be set} proof ex [it : be set] (for [x : be set]
    (x in it) iff (x eq y))
  singletonunq : {y} proof for [it] for [it'] (for [x] (x in it iff x eq y)
    and for [x] (x in it' iff x eq y)) implies it eq it'
  singleton : be set → be set = [y] func ([it] for [x] x in it iff x eq y)
    (singletonex y) (singletonunq y)
  :
  fraenkel : {A : be set} {P : be set → be set → prop}
    proof (for [x : be set] for [y : be set] for [z : be set]
      ((P x y) and (P x z)) implies y eq z) → proof (ex [X]
        for [x] ((x in X) iff (ex [y] y in A and (P y x))))
  :
  subsetclosed : {m} prop = [m] for [x] (for [y]
    (((x in m) and (y ⊆ x)) implies (y in m)))
  powersetclosed : {m} prop = [m] for [x] (x in m implies (ex [z] z in m
    and (for [y] y ⊆ x implies y in z)))
  tarski_ax : proof for [n] (ex [m] (
    n in m and subsetclosed m and
    powersetclosed m and for [x]
    (x ⊆ m implies ((isomorphic x m) or x in m))))
  :
}

```

Figure 13: Encoding TG Set Theory

*func* is interpreted using the description operator from ZFC, and the interpretation of *mode* is trivial.

Finally, attributed modes *adjective*  $\vartheta$   $A$  are interpreted using the conjunction of the interpretations  $P : \text{set} \rightarrow \text{prop}$  of  $\vartheta$  and  $Q : \text{Class } P \rightarrow \text{prop}$  of  $A$ . Note how sequential conjunction is needed to use the truth of  $P x$  in the second conjunct. This is necessary because in Mizar  $A$  only has to be defined for terms of type  $\vartheta$ , which corresponds to  $Q$  only being applicable to sets satisfying  $P$ .

We omit the straightforward but technical remaining cases for formula and proof constructors.

**TG** The view *TarskiZFC* from *Tarski* to *ZFC* is straightforward, and we omit the details. However, the view is only partial because it omits the Tarski axiom.

Partial views in LF simply omit cases. Consequently, their homomorphic extensions are partial functions. For our view, that means that every definition or theorem that

| Mizar                   | ZFC  |
|-------------------------|--|
| $\vartheta : tp$        | $MizarZFC(\vartheta) : set \rightarrow prop$   |
| $\alpha : prop$         | $MizarZFC(\alpha) : prop$                      |
| $P : proof \alpha$      | $MizarZFC(P) : ded MizarZFC(P)$                |
| $\alpha : be \vartheta$ | $MizarZFC(\alpha) : Class MizarZFC(\vartheta)$ |

Figure 14: Mizar/TG Declarations in ZFC

```

view MizarZFC : Mizar → ZFC = {
  tp      :=  set → prop
  prop    :=  prop
  proof   :=  ded
  be      :=  [f] Class f
  set     :=  [x] ⊤
  is      :=  [a] [F] F (cwhich a)
  in      :=  [a] [b] (cwhich a) ∈ (cwhich b)
  :
  func    :=  [F] [EX] [UNQ] δ F (andI EX UNQ)
  mode    :=  [F : Class A → prop] [EX] ([x] (A x) ∧' [p] F (celem x p))
  adjective :=  [P : set → prop] [Q : Class T → prop]
                [x] (P x) ∧' [p] (Q (celem x p))
  :
}

```

Figure 15: Interpreting *Mizar* in *ZFC*

depends on the Tarski axiom cannot be translated to ZFC. This is more harmful than it sounds: Since the Tarski axiom is used in Mizar to prove the existence of power set, infinity, and choice, almost all definitions depend on it.

However, we have already designed an elegant extension of the notion of Twelf views that solves this problem in (Dumbrava & Rabe 2010). With this extension, it is possible to make *TarskiZFC* undefined for the Tarski axiom, but map Mizar’s theorems of power set, infinity, and choice, which depend on the Tarski axiom, to their counterparts in ZFC. We say that power set, infinity, and choice are *recovered* by the view. Then Mizar expressions that are stated in terms of the recovered constants can still be translated to ZFC, and the preservation of truth is still guaranteed. With this amendment, most theorems in the Mizar library can be translated. Only theorems that directly appeal to the Tarski axiom remain untranslatable, and that is intentional because they are likely to be unprovable over ZFC.

## 6.4 Related Work

Mizar is infamous for being impenetrable as a logic, and previous work has focused on making the syntax and semantics of Mizar more accessible. The main source of complexity is the type system.

(Wiedijk 2007) gives a comprehensive account on the syntax and semantics of the Mizar type system. It interprets types as predicates in the same way as we did here. A translation to first-order logic is given that is similar in spirit to our translation to ZFC. An alternative approach using type algebras was given in (Bancerek 2003).

In (Urban 2003), a translation of Mizar into TPTP-based first-order logic is given. It also interprets types as predicates.

## 7 Conclusion and Future Work

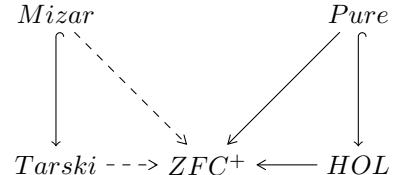
We have represented three foundations of mathematics and two translations between them in a formal framework, namely Twelf. The most important feature is that the well-definedness and soundness of the translations are verified statically and automatically by the Twelf type checker. In particular, the LF type system guarantees that the translation functions preserve provability. Our work is the first systematic case study of statically verified translations between foundations.

Our foundations are ZFC, Mizar’s Tarski-Grothendieck set theory (TG) and Isabelle’s higher-order logic (HOL). We chose ZFC as the most widespread foundation of non-formalized mathematics, and our formalization stays notably close to textbook developments of ZFC. (We have to add a global choice function though both for Isabelle/HOL and for Mizar/TG.) We chose Isabelle/HOL and Mizar because they are two of the most advanced foundations of formalized mathematics in terms of library size and (semi-)automated proof support. They are also foundationally very different – higher-order logic and untyped set theory, respectively – and represent the whole spectrum of foundations. Moreover, our formalizations make the foundational assumptions of these systems explicit and thus contribute to their documentations and systematic comparison.

We have formalized translations from Isabelle/HOL and Mizar/TG into ZFC as indicated on the right. These translations can be seen as giving two foundations used in formalized mathematics a semantics in terms of the foundation dominant in traditional mathematics. Actually, the translation from Mizar/TG to  $ZFC^+$  is only partial because the former is stronger than the latter, but this is no serious concern as we discussed in Sect. 6.3. We did not give the inverse translation from ZFC to Mizar/TG, but that would be straightforward. However, a corresponding translation from ZFC to Isabelle/HOL remains a challenge. (Translations such as the one in (Aczel 1998) would not be inverse to ours.)

Future work will focus on two research directions.

Firstly, we will formalize more foundations and translations. This is an on-going effort



in the LATIN project (Kohlhase, Mossakowski & Rabe 2009), which will provide a large library of statically verified foundation translations and for which this work provides the theoretical bases and seed library. Examples of further systems are Coq (Coquand & Huet 1988) or PVS (Owre et al. 1992).

Secondly, a major drawback of statically verified translations is that the extracted translation functions cannot be directly applied to the libraries of the foundations: First those libraries must be represented in the foundational framework. This is a conceptually trivial but practically long-term research effort that is still under way.

**Acknowledgements** This work was supported by grant KO 2428/9-1 (LATIN) by the Deutsche Forschungsgemeinschaft.

## References

- Aczel, P. (1998), On Relating Type Theories and Set Theories, *in* T. Altenkirch, W. Naraschewski & B. Reus, eds, ‘Types for Proofs and Programs’, pp. 1–18.
- Bancerek, G. (2003), On the structure of Mizar types, Vol. 85 of *Electronic Notes in Theoretical Computer Science*, pp. 69–85.
- Bourbaki, N. (1964), Univers, *in* ‘Séminaire de Géométrie Algébrique du Bois Marie - Théorie des topos et cohomologie étale des schémas’, Springer, pp. 185–217.
- Brown, C. (2006), Combining Type Theory and Untyped Set Theory, *in* U. Furbach & N. Shankar, eds, ‘International Joint Conference on Automated Reasoning’, Springer, pp. 205–219.
- Church, A. (1940), ‘A Formulation of the Simple Theory of Types’, *Journal of Symbolic Logic* 5(1), 56–68.
- Constable, R., Allen, S., Bromley, H., Cleaveland, W., Cremer, J., Harper, R., Howe, D., Knoblock, T., Mendler, N., Panangaden, P., Sasaki, J. & Smith, S. (1986), *Implementing Mathematics with the Nuprl Development System*, Prentice-Hall.
- Coquand, T. & Huet, G. (1988), ‘The Calculus of Constructions’, *Information and Computation* 76(2/3), 95–120.
- de Bruijn, N. (1970), The Mathematical Language AUTOMATH, *in* M. Laudet, ed., ‘Proceedings of the Symposium on Automated Demonstration’, Vol. 25 of *Lecture Notes in Mathematics*, Springer, pp. 29–61.
- de Nivelle, H. (2010), Classical Logic with Partial Functions, *in* J. Giesl & R. Hähnle, eds, ‘Automated Reasoning’, Springer, pp. 203–217.
- Dumbrava, S. & Rabe, F. (2010), ‘Structuring Theories with Partial Morphisms’. Workshop on Abstract Development Techniques.

- Farmer, W., Guttman, J. & Thayer, F. (1993), ‘IMPS: An Interactive Mathematical Proof System’, *Journal of Automated Reasoning* **11**(2), 213–248.
- Fraenkel, A. (1922), ‘The notion of ’definite’ and the independence of the axiom of choice’.
- Gordon, M. (1988), HOL: A Proof Generating System for Higher-Order Logic, in G. Birtwistle & P. Subrahmanyam, eds, ‘VLSI Specification, Verification and Synthesis’, Kluwer-Academic Publishers, pp. 73–128.
- Gordon, M., Milner, R. & Wadsworth, C. (1979), *Edinburgh LCF: A Mechanized Logic of Computation*, number 78 in ‘LNCS’, Springer Verlag.
- Gordon, M. & Pitts, A. (1993), The HOL Logic, in M. Gordon & T. Melham, eds, ‘Introduction to HOL, Part III’, Cambridge University Press, pp. 191–232.
- Hales, T. (2003), ‘The flyspeck project’. See <http://code.google.com/p/flyspeck/>.
- Harper, R., Honsell, F. & Plotkin, G. (1993), ‘A framework for defining logics’, *Journal of the Association for Computing Machinery* **40**(1), 143–184.
- Harper, R., Sannella, D. & Tarlecki, A. (1994), ‘Structured presentations and logic representations’, *Annals of Pure and Applied Logic* **67**, 113–160.
- Harrison, J. (1996), HOL Light: A Tutorial Introduction, in ‘Proceedings of the First International Conference on Formal Methods in Computer-Aided Design’, Springer, pp. 265–269.
- Hurd, J. (2009), OpenTheory: Package Management for Higher Order Logic Theories, in G. D. Reis & L. Théry, eds, ‘Programming Languages for Mechanized Mathematics Systems’, ACM, pp. 31–37.
- Iancu, M. & Rabe, F. (2010), ‘Formalizing Foundations of Mathematics, LF Encodings’. see <https://latin.omdoc.org/wiki/FormalizingFoundations>.
- Jaśkowski, S. (1934), ‘On the rules of suppositions in formal logic’, *Studia Logica* **1**, 5–32.
- Keller, C. & Werner, B. (2010), Importing HOL Light into Coq, in M. Kaufmann & L. Paulson, eds, ‘Proceedings of the Interactive Theorem Proving conference’. to appear in LNCS.
- Klein, G., Nipkow, T. & (eds.), L. P. (2004), ‘Archive of Formal Proofs’. <http://afp.sourceforge.net/>.
- Kohlhase, M., Mossakowski, T. & Rabe, F. (2009), ‘The LATIN Project’. See <https://trac.omdoc.org/LATIN/>.
- Krauss, A. & Schropp, A. (2010), A Mechanized Translation from Higher-Order Logic to Set Theory, in M. Kaufmann & L. Paulson, eds, ‘Proceedings of the Interactive Theorem Proving conference’. to appear in LNCS.

- Landau, E. (1930), *Grundlagen der Analysis*, Akademische Verlagsgesellschaft.
- Lovas, W. & Pfenning, F. (2009), Refinement Types as Proof Irrelevance, in P. Curien, ed., ‘Typed Lambda Calculi and Applications’, Vol. 5608 of *Lecture Notes in Computer Science*, Springer, pp. 157–171.
- Matuszewski, R. (1990), ‘Formalized Mathematics’. <http://mizar.uwb.edu.pl/fm/>.
- McLaughlin, S. (2006), An Interpretation of Isabelle/HOL in HOL Light, in N. Shankar & U. Furbach, eds, ‘Proceedings of the 3rd International Joint Conference on Automated Reasoning’, Vol. 4130 of *Lecture Notes in Computer Science*, Springer.
- Mizar (2009), ‘Grammar, version 7.11.02’. <http://mizar.org/language/mizar-grammar.xml>.
- Naumov, P., Stehr, M. & Meseguer, J. (2001), The HOL/NuPRL proof translator - a practical approach to formal interoperability, in ‘14th International Conference on Theorem Proving in Higher Order Logics’, Springer.
- Nipkow, T., Paulson, L. & Wenzel, M. (2002), *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Springer.
- Norell, U. (2005), ‘The Agda WiKi’ <http://wiki.portal.chalmers.se/agda>
- Obua, S. & Skalberg, S. (2006), Importing HOL into Isabelle/HOL, in N. Shankar & U. Furbach, eds, ‘Proceedings of the 3rd International Joint Conference on Automated Reasoning’, Vol. 4130 of *Lecture Notes in Computer Science*, Springer.
- Owre, S., Rushby, J. & Shankar, N. (1992), PVS: A Prototype Verification System, in D. Kapur, ed., ‘11th International Conference on Automated Deduction (CADE)’, Springer, pp. 748–752.
- Paulson, L. (1994), *Isabelle: A Generic Theorem Prover*, Vol. 828 of *Lecture Notes in Computer Science*, Springer.
- Paulson, L. & Coen, M. (1993), ‘Zermelo-Fraenkel Set Theory’. Isabelle distribution, ZF/ZF.thy.
- Pfenning, F. & Schürmann, C. (1999), ‘System description: Twelf - a meta-logical framework for deductive systems’, *Lecture Notes in Computer Science* **1632**, 202–206.
- Pfenning, F., Schürmann, C., Kohlhase, M., Shankar, N. & Owre., S. (2003), ‘The Logosphere Project’. <http://www.logosphere.org/>
- Poswolsky, A. & Schürmann, C. (2008), System Description: Delphin - A Functional Programming Language for Deductive Systems, in A. Abel & C. Urban, eds, ‘International Workshop on Logical Frameworks and Metalanguages: Theory and Practice’, ENTCS, pp. 135–141.

- Rabe, F. (2010), Representing Isabelle in LF, *in* K. Crary & M. Miculan, eds, ‘Logical Frameworks and Meta-languages: Theory and Practice’, Vol. 34 of *EPTCS*.
- Rabe, F. & Schürmann, C. (2009), A Practical Module System for LF, *in* J. Cheney & A. Felty, eds, ‘Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)’, ACM Press, pp. 40–48.
- Schürmann, C. & Stehr, M. (2004), An Executable Formalization of the HOL/Nuprl Connection in the Metalogical Framework Twelf, *in* ‘11th International Conference on Logic for Programming Artificial Intelligence and Reasoning’.
- T. Hales and G. Gonthier and J. Harrison and F. Wiedijk (2008), ‘A Special Issue on Formal Proof’. Notices of the AMS 55(11).
- Tarski, A. (1938), ‘Über Unerreichbare Kardinalzahlen’, *Fundamenta Mathematicae* 30, 176–183.
- Trybulec, A. (1989), ‘Tarski Grothendieck Set Theory’, *Journal of Formalized Mathematics Axiomatics*.
- Trybulec, A. & Blair, H. (1985), Computer Assisted Reasoning with MIZAR, *in* A. Joshi, ed., ‘Proceedings of the 9th International Joint Conference on Artificial Intelligence’, pp. 26–28.
- Urban, J. (2003), Translating Mizar for First Order Theorem Provers, *in* A. Asperti, B. Buchberger & J. Davenport, eds, ‘Mathematical Knowledge Management’, Springer, pp. 203–215.
- van Bentham Jutting, L. (1977), Checking Landau’s “Grundlagen” in the AUTOMATH system, PhD thesis, Eindhoven University of Technology.
- Wenzel, M. (2009), ‘The Isabelle/Isar Reference Manual’. <http://isabelle.in.tum.de/documentation.html>, Dec 3, 2009.
- Whitehead, A. & Russell, B. (1913), *Principia Mathematica*, Cambridge University Press.
- Wiedijk, F. (2006), ‘Is ZF a hack?: Comparing the complexity of some (formalist interpretations of) foundational systems for mathematics’, *Journal of Applied Logic* 4(4), 622–645.
- Wiedijk, F. (2007), Mizar’s Soft Type System, *in* K. Schneider & J. Brandt, eds, ‘Theorem Proving in Higher Order Logics’, Vol. 4732 of *Lecture Notes in Computer Science*, Springer, pp. 383–399.
- Wiener, N. (1967), A Simplification of the Logic of Relations, *in* J. van Heijenoort, ed., ‘From Frege to Gödel’, Harvard Univ. Press, pp. 224–227.
- Zermelo, E. (1908), ‘Untersuchungen über die Grundlagen der Mengenlehre I’, *Mathematische Annalen* 65, 261–281. English title: Investigations in the foundations of set theory I.

# An Institutional View on Categorical Logic

Florian Rabe, Jacobs University Bremen

Till Mossakowski, DFKI-Lab Bremen and University of Bremen

Valeria de Paiva, Palo Alto Research Center, Palo Alto, California

Lutz Schröder, DFKI-Lab Bremen and University of Bremen

Joseph Goguen, University of California, San Diego †

## Abstract

We introduce a generic notion of categorical propositional logic and provide a construction of a preorder-enriched institution out of such a logic, following the Curry-Howard-Tait paradigm. The logics are specified as theories of a meta-logic within the logical framework LF such that institution comorphisms are obtained from theory morphisms of the meta-logic. We prove several logic-independent results including soundness and completeness theorems and instantiate our framework with a number of examples: classical, intuitionistic, linear and modal propositional logic.

We dedicate this work to the memory of our dear friend and colleague Joseph Goguen who passed away during its preparation.

## 1 Introduction

The well-known Curry-Howard isomorphism [25, 18] establishes a correspondence between propositions and types, and proofs and terms, and proof and term reductions. A number of correspondences between various kinds of logical theories,  $\lambda$ -calculi and categories have been established, see Fig. 1.

Here, we present work aimed at casting the propositional part of these correspondences in a common framework based on the theory of *institutions*. The notion of institution arose within computer science as a response to the population explosion among logics in use [20, 22]. Its key idea is to focus on abstractly axiomatizing the *satisfaction* relation between sentences and models. A wide range of logics, e.g., equational [20], Horn [20], first-order [20], modal [8, 40], higher-order [7], polymorphic [36], temporal [17], process [17], behavioral [5], and object-oriented [21] logics, have been formalized as institutions. A surprisingly large amount of meta-logical reasoning can be carried out in this abstract framework; e.g., institutions have been used to give general foundations for modularization of theories and programs [35, 14], and substantial portions of classical model theory can be lifted to the level of institutions [42, 9, 10, 12]. The objective is to be able to apply each meta-mathematical result to the widest possible range of abstract institutions; see e.g. [13, 9, 10, 11, 12].

In the sequel, we give a general notion of categorical propositional logic, and a construction of a proof-theoretic institution out of such a logic that formalizes the abstract Curry-Howard-Tait correspondence. Then we use the institutional meta-theory to establish logic-independent results including general soundness and completeness theorems, and we discuss how the Curry-Howard-Tait isomorphism can be cast as a morphism of institutions.

|                                       |  |  |      |
|---------------------------------------|--|--|------|
| conjunctive logic                     | pairing  | cartesian categories                                       | [26] |
| positive logic                        | $\lambda$ with paring                                    | cartesian closed categories                                | [26] |
| intuitionistic propositional logic    | $\lambda$ with pairing and sums                          | bicartesian closed categories                              | [26] |
| classical propositional logic         | $\lambda$ with pairing, sums and $\neg\neg$ -elimination | bicartesian closed categories with $\neg\neg$ -elimination | [26] |
| modal logics, e.g., intuitionistic S4 |  | monoidal comonad with strong monad                         | [6]  |
| linear logic                          |  | *-autonomous categories                                    | [39] |
| first-order logic                     | $\lambda$ with pairing, sums and certain dependent types | hyperdoctrines   | [37] |
| Martin-Löf type theory                | $\lambda$ with pairing, sums and dependent types         | locally cartesian closed categories                        | [38] |

Figure 1: Curry-Howard correspondences

## 2 Institutions and Logics

We assume that the reader is familiar with basic notions from category theory (cf. e.g. [1, 27]). We denote the class of objects of a category  $\mathbb{C}$  by  $|\mathbb{C}|$ , the set of morphisms from  $A$  to  $B$  by  $\mathbb{C}(A, B)$ , and composition by  $\circ$ . Let  $\mathbb{CAT}$  be the category of all categories (which of course lives in a higher set-theoretic universe). Institutions are defined as follows.

**Definition 2.1.** An *institution*  $\mathcal{I} = (\text{Sign}^{\mathcal{I}}, \text{Sen}^{\mathcal{I}}, \text{Mod}^{\mathcal{I}}, \models^{\mathcal{I}})$  consists of

- a category  $\text{Sign}^{\mathcal{I}}$  of *signatures*;
- a functor  $\text{Sen}^{\mathcal{I}}: \text{Sign}^{\mathcal{I}} \rightarrow \text{Set}$  giving, for each signature  $\Sigma$ , the set  $\text{Sen}^{\mathcal{I}}(\Sigma)$  of *sentences*, and for each signature morphism  $\sigma: \Sigma \rightarrow \Sigma'$ , the *sentence translation map*  $\text{Sen}^{\mathcal{I}}(\sigma): \text{Sen}^{\mathcal{I}}(\Sigma) \rightarrow \text{Sen}^{\mathcal{I}}(\Sigma')$ , with  $\sigma(\varphi)$  abbreviating  $\text{Sen}^{\mathcal{I}}(\sigma)(\varphi)$ ;
- a functor  $\text{Mod}^{\mathcal{I}}: (\text{Sign}^{\mathcal{I}})^{\text{op}} \rightarrow \mathbb{CAT}$  giving, for each signature  $\Sigma$ , the category  $\text{Mod}^{\mathcal{I}}(\Sigma)$  of *models*, and for each signature morphism  $\sigma: \Sigma \rightarrow \Sigma'$ , the *reduct functor*  $\text{Mod}^{\mathcal{I}}(\sigma): \text{Mod}^{\mathcal{I}}(\Sigma') \rightarrow \text{Mod}^{\mathcal{I}}(\Sigma)$ , with  $M'|_{\sigma}$  abbreviating  $\text{Mod}^{\mathcal{I}}(\sigma)(M')$ , the  $\sigma$ -reduct of  $M'$ ; and
- a satisfaction relation  $\models_{\Sigma}^{\mathcal{I}} \subseteq |\text{Mod}^{\mathcal{I}}(\Sigma)| \times \text{Sen}^{\mathcal{I}}(\Sigma)$  for each  $\Sigma \in |\text{Sign}^{\mathcal{I}}|$ ,

such that for each  $\sigma: \Sigma \rightarrow \Sigma'$  in  $\text{Sign}^{\mathcal{I}}$ , the *satisfaction condition*

$$M' \models_{\Sigma'}^{\mathcal{I}} \sigma(\varphi) \iff M'|_{\sigma} \models_{\Sigma}^{\mathcal{I}} \varphi$$

holds for all  $M' \in |\text{Mod}^{\mathcal{I}}(\Sigma')|$  and all  $\varphi \in \text{Sen}^{\mathcal{I}}(\Sigma)$ .

**Example 2.2** (Classical propositional logic). As a running example, we give the institution **CPL** of classical propositional logic. Its signatures are just sets  $\Sigma$  of propositional variables, i.e.,  $\text{Sign}^{\text{CPL}} = \text{Set}$ .  $\text{Sen}^{\text{CPL}}(\Sigma)$  is the set of propositional formulas with propositional variables from  $\Sigma$  and connectives for conjunction, disjunction, implication and negation. A

signature morphism  $\sigma$  is a mapping between the propositional variables, and sentence translation  $\text{Sen}^{\mathbf{CPL}}(\sigma)$  is the extension of  $\sigma$  to all formulas. For example,  $A \wedge B \in \text{Sen}^{\mathbf{CPL}}(\{A, B\})$ ; and for  $\sigma : \{A, B\} \rightarrow \{C, D\}$ ,  $\sigma : A \mapsto C, B \mapsto D$ , we have  $\text{Sen}^{\mathbf{CPL}}(\sigma)(A \wedge B) = C \wedge D$ . Models of  $\Sigma$  are truth valuations, i.e., mappings from  $\Sigma$  into the standard two-valued Boolean algebra  $Bool = \{0, 1\}$  with 0 denoting false and 1 denoting truth. A unique model morphism between  $\Sigma$ -models  $M$  and  $M'$  exists iff for all  $p \in \Sigma$ ,  $M(p) \leq M'(p)$ . Given  $\sigma : \Sigma_1 \rightarrow \Sigma_2$  and a  $\Sigma_2$ -model  $M_2 : \Sigma_2 \rightarrow Bool$ , the reduct  $M_2|_\sigma$  is just the composition  $M_2 \circ \sigma$ . Reducts of morphisms are then clear. Finally,  $M \vdash_{\Sigma}^{\mathbf{CPL}} \varphi$  holds iff  $\varphi$  evaluates to 1 under the usual extension of  $M$  to all formulas. For example,  $M : A \mapsto 1, B \mapsto 0$  is an object of  $\text{Mod}^{\mathbf{CPL}}(\{A, B\})$ , and  $M \not\models_{\{A, B\}}^{\mathbf{CPL}} A \wedge B$  because  $A \wedge B$  evaluates to  $1 \wedge 0 = 0$ .

**Example 2.3** (Classical first-order logic). In the institution **FOL** of first-order logic, signatures are first-order signatures, consisting of function and predicate symbols with arity. Signature morphisms map symbols such that arity is preserved. Sentences are first-order formulas extending  $\text{Sen}^{\mathbf{CPL}}$  with variables, terms, and universal and existential quantification. Sentence translation means replacement of the translated symbols. Models are first-order structures  $(U, \nu)$  with a universe  $U$  and an interpretation function  $\nu$  assigning functions and relations of appropriate arities to the function and predicate symbols. Nullary predicate symbols are interpreted as an element of  $Bool$ . Model morphisms are the usual homomorphisms between first-order structures (the need to preserve but not necessarily reflect the holding of predicates). Model reduction means reassembling the model's components according to the signature morphism. Satisfaction is the usual satisfaction of a first-order sentence in a first-order structure. We do not go into the details here because we only need **FOL** to give an example of an institution comorphism below.

Proof-theoretic extensions of institutions use richer structures than sets of sentences to express the syntax of an institution. They were first explored in [19] where for each signature, there is a category of proofs, with sentences as objects and proofs as morphisms. Here, we introduce preorder-enriched institutions that extend this with reductions between proof terms, modeled by a preorder on morphisms. We write  $f \rightsquigarrow g$  to express that  $f$  reduces to  $g$  for two proofs  $f, g$  with the same domain and codomain. In other words, proof categories are small preorder-enriched categories, and we write  $\text{OrdCat}$  for the category of preorder-enriched small categories. If  $U : \text{OrdCat} \rightarrow \text{Set}$  is the functor forgetting morphisms (i.e.,  $U(A)$  is the set of objects of  $A$ ), this is formally defined as follows.

**Definition 2.4.** A *preorder-enriched institution* is a tuple  $(\text{Sign}, \text{Pr}, \text{Mod}, \models)$  where  $\text{Pr} : \text{Sign} \rightarrow \text{OrdCat}$  is a functor such that  $(\text{Sign}, U \circ \text{Pr}, \text{Mod}, \models)$  is an institution. In that case, we will use  $\text{Sen}$  to abbreviate  $U \circ \text{Pr}$ .

**Example 2.5. CPL** can be turned into a preorder-enriched institution in various ways. Let  $\mathbf{CPL}^{ND\lambda}$  denote propositional logic with natural deduction. Then  $\text{Pr}^{\mathbf{CPL}}(\Sigma)$  is the category where the objects are propositional formulae over  $\Sigma$ , and morphisms from  $\varphi$  to  $\psi$  are natural deduction proof terms (modulo  $\alpha$ -congruence) in context  $x : \varphi \triangleright M : \psi$  as given by the rules in Fig. 2, cf. [6]. The assumption rule gives us  $x : \varphi \triangleright x : \varphi$  as the identity morphism; and the composition of morphisms  $x : \varphi \triangleright M : \psi$  and  $y : \psi \triangleright N : \chi$  is given by substitution:  $x : \varphi \triangleright N[y := M] : \chi$ . The category laws easily follow from general properties of substitutions. The pre-order on these morphisms is given by  $\beta$ -reduction on proof terms as given in Fig. 3. For a signature morphism  $\sigma : \Sigma \rightarrow \Sigma'$ , the functor  $\text{Pr}^{\mathbf{CPL}}(\sigma)$  acts on objects of  $\text{Pr}^{\mathbf{CPL}}(\Sigma)$  like

|  |  |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|---|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| (assumption) $\frac{}{\Gamma, x : A \triangleright x : A}$   |  |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| <hr/>  |  |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| (⊤-I) $\frac{}{\Gamma \triangleright \Delta : \top}$   |  |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| (∧-I) $\frac{\Gamma \triangleright M : A \quad \Gamma \triangleright N : B}{\Gamma \triangleright \langle M, N \rangle : A \wedge B}$  | (∧-E1) $\frac{\Gamma \triangleright M : A \wedge B}{\Gamma \triangleright \text{fst}(M) : A}$                            | (∧-E2) $\frac{\Gamma \triangleright M : A \wedge B}{\Gamma \triangleright \text{snd}(M) : B}$ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| <hr/>  |  |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| (→-I) $\frac{\Gamma, x : A \triangleright M : B}{\Gamma \triangleright \lambda x : A. M : A \rightarrow B}$  | (→-E) $\frac{\Gamma \triangleright M : A \rightarrow B \quad \Gamma \triangleright N : A}{\Gamma \triangleright MN : B}$ |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| <hr/>  |  |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| (⊥-E) $\frac{\Gamma \triangleright M : \perp}{\Gamma \triangleright \nabla_A(M) : A}$  |  |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| (∨-I1) $\frac{\Gamma \triangleright M : A}{\Gamma \triangleright \text{inl}(M) : A \vee B}$  | (∨-I2) $\frac{\Gamma \triangleright M : B}{\Gamma \triangleright \text{inr}(M) : A \vee B}$                              |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| (∨-E) $\frac{\Gamma \triangleright M : A \vee B \quad \Gamma, x : A \triangleright N : C \quad \Gamma, y : B \triangleright P : C}{\Gamma \triangleright \text{case } M \text{ of } \text{inl}(x) \rightarrow N \mid \text{inr}(y) \rightarrow P : C}$ |  |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| <hr/>  |  |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| (tnd) $\frac{}{\Gamma \triangleright \text{tnd}_A : A \vee \neg A}$  |  |   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Figure 2: Natural deduction proof rules and proof terms for classical propositional logic, introduced in five stages: void logic, conjunctive logic, conjunctive-implicational logic, intuitionistic logic, classical logic.

$\text{Sen}^{\text{CPL}}(\sigma)$ ; similarly,  $\text{Pr}^{\text{CPL}}(\sigma)$  acts on morphisms of  $\text{Pr}^{\text{CPL}}(\Sigma)$ , i.e., on natural deduction proofs, by replacing all occurrences of a symbol of  $\Sigma$  with its image under  $\sigma$ .

We define a functor  $\overline{\cdot} : \text{OrdCat} \rightarrow \text{CAT}$  by quotienting out the preorder; i.e., given a preorder-enriched category  $A$ ,  $\overline{A}$  is the quotient of  $A$  by the equivalence generated by the preorder on hom-sets. Similarly,  $\text{thin}(\cdot) : \text{OrdCat} \rightarrow \text{CAT}$  is a functor that quotients all non-empty hom-sets (i.e. sets of 1-cells) to singletons.

In any institution, we have the usual notion of *semantic consequence*: For a set  $\Phi$  of  $\Sigma$ -sentences, let  $M \models_{\Sigma} \Phi$  denote  $M \models_{\Sigma} \varphi$  for every  $\varphi \in \Phi$ . Then we say that that a  $\Sigma$ -sentence  $\psi$  is a consequence of  $\Phi$ , and write  $\Phi \models_{\Sigma} \psi$ , iff  $M \models_{\Sigma} \Phi$  implies  $M \models_{\Sigma} \psi$  for every  $\Sigma$ -model  $M$ .

In a preorder-enriched institution, we can also define an *entailment* relation  $\vdash_{\Sigma}$  between  $\Sigma$ -

$$\begin{array}{ll}
 M : \top & \rightsquigarrow_{\beta} \Delta : \top \\
 \text{fst}(\langle M, N \rangle) & \rightsquigarrow_{\beta} M \\
 \text{snd}(\langle M, N \rangle) & \rightsquigarrow_{\beta} N \\
 (\lambda x : A. M) N & \rightsquigarrow_{\beta} M[x := N] \\
 \text{case inl}(M) \text{ of } \text{inl}(x) \rightarrow N \mid \text{inr}(y) \rightarrow P & \rightsquigarrow_{\beta} N[x := M] \\
 \text{case inr}(M) \text{ of } \text{inl}(x) \rightarrow N \mid \text{inr}(y) \rightarrow P & \rightsquigarrow_{\beta} P[x := M]
 \end{array}$$

Figure 3:  $\beta$ -reduction rules for proof terms.

sentences as follows:  $\varphi \vdash_{\Sigma} \psi$  iff there exists a morphism  $\varphi \rightarrow \psi$  in  $\text{Pr}(\Sigma)$ . A preorder-enriched institution is *sound* if  $\phi \vdash_{\Sigma} \psi$  implies  $\{\phi\} \models_{\Sigma} \psi$ , and *complete* if the converse implication holds.

In an institution  $\mathcal{I}$ , a *theory* is a pair  $T = \langle \Sigma, \Gamma \rangle$ , where  $\Sigma \in |\text{Sign}|$  and  $\Gamma \subseteq \text{Sen}(\Sigma)$ . A *theory morphism*  $\sigma: \langle \Sigma, \Gamma \rangle \rightarrow \langle \Sigma', \Gamma' \rangle$  is a signature morphism  $\sigma: \Sigma \rightarrow \Sigma'$  for which  $\Gamma' \models_{\Sigma'} \sigma(\Gamma)$ , that is,  $\sigma$  maps axioms to consequences.

This defines a category **Th** of theories, and it is easy to extend **Sen** (or **Pr**) and **Mod** to **Th** by putting  $\text{Sen}(\langle \Sigma, \Gamma \rangle) = \text{Sen}(\Sigma)$  and letting  $\text{Mod}(\langle \Sigma, \Gamma \rangle)$  be the full subcategory of  $\text{Mod}(\Sigma)$  induced by the class of those models  $M$  satisfying  $\Gamma$ .

Relationships between institutions are captured by several variants of institution morphisms and comorphisms. Here we use the following definition.

**Definition 2.6.** Given preorder-enriched institutions  $\mathcal{I}$  and  $\mathcal{J}$ , a *comorphism* between them is a tuple  $(\Phi, \alpha, \beta)$  consisting of

- a functor  $\Phi: \text{Sign}^{\mathcal{I}} \rightarrow \text{Sign}^{\mathcal{J}}$ ,
- a natural transformation  $\alpha: \text{Pr}^{\mathcal{I}} \rightarrow \text{Pr}^{\mathcal{J}} \circ \Phi$ ,
- a natural transformation  $\beta: \text{Mod}^{\mathcal{J}} \circ \Phi^{op} \rightarrow \text{Mod}^{\mathcal{I}}$

such that the following satisfaction condition holds for all  $\Sigma \in |\text{Sign}^{\mathcal{I}}|$ ,  $M' \in |\text{Mod}^{\mathcal{J}}(\Phi(\Sigma))|$  and  $\varphi \in |\text{Sen}^{\mathcal{I}}(\Sigma)|$ :

$$M' \models_{\Phi(\Sigma)}^{\mathcal{J}} \alpha_{\Sigma}(\varphi) \text{ iff } \beta_{\Sigma}(M') \models_{\Sigma}^{\mathcal{I}} \varphi.$$

**Example 2.7.** To understand the intuition behind this definition, assume that the institution **FOL** of first-order logic from Example 2.3 is extended to a preorder-enriched institution **FOL**<sup>ND $\lambda$</sup>  by extending the natural deduction calculus used in Example 2.5 with rules for the quantifiers. Then the natural inclusion of propositional into first-order logic is formalized as a comorphism  $(\Phi, \alpha, \beta)$  from **CPL**<sup>ND $\lambda$</sup>  to **FOL**<sup>ND $\lambda$</sup> .

For every **CPL**<sup>ND $\lambda$</sup> -signature  $\Sigma$ ,  $\Phi(\Sigma)$  is the first-order signature with  $\Sigma$  as the set of nullary predicate symbols and with no other function or predicate symbols. The sentence and proof translation  $\alpha_{\Sigma}$  is an inclusion functor from  $\text{Pr}^{\text{CPL}}(\Sigma)$  to  $\text{Pr}^{\text{FOL}}(\Phi(\Sigma))$  because every propositional formula and every propositional proof over  $\Sigma$  is also a first-order formula or proof over  $\Phi(\Sigma)$ . For a model  $(U, \nu) \in |\text{Mod}^{\text{FOL}}(\Phi(\Sigma))|$ ,  $\nu$  is simply a valuation  $\Sigma \rightarrow \text{Bool}$ ; and thus we can put  $\beta_{\Sigma}(U, \nu) = \nu$ . A model morphism in  $\text{Mod}^{\text{FOL}}(\Phi(\Sigma))$  needs to preserve holding of nullary predicates and hence induces a model morphism in  $\text{Mod}^{\text{CPL}}(\Sigma)$ . The satisfaction condition, intuitively, requires that truth and consequence are preserved under a comorphism. It is easy to verify because  $\alpha_{\Sigma}(\varphi) = \varphi$  and  $\beta_{\Sigma}(U, \nu) = \nu$  for a model  $M' = (U, \nu)$  and because  $\models^{\text{FOL}}$  agrees with  $\models^{\text{CPL}}$  on propositional formulas.

**Example 2.8.** The well-known Curry-Howard isomorphism can be formulated as an institution isomorphism between **CPL**<sup>ND $\lambda$</sup>  and **CPL**<sup>ND</sup>, where **CPL**<sup>ND</sup> is the standard formulation of natural deduction, with proof trees as morphisms in proof categories, and proof tree reduction as the preorder.

Together with the obvious composition and identities, this defines a category<sup>1</sup> **CoIns** of preorder-enriched institutions and comorphisms.

---

<sup>1</sup>Of course, this category lives in a higher set-theoretic universe. We omit the foundational issues here.

### 3 Categorical Logic

Lambek and Scott [26] study categorical logic by introducing *deductive systems* as directed graphs with a composition structure, and later impose the usual axioms for (cartesian, cartesian closed, bicartesian closed) categories on these. Objects in categories serve as ‘types’ for morphisms, hence the ‘propositions as types’ paradigm becomes ‘propositions as objects’.

We will use a formal meta-language to formalize categorical logic, namely an institution  $\mathcal{DFOL}$  that extends many-sorted first-order logic with dependent sorts.  $\mathcal{DFOL}$  is formally introduced in [32], and we will only sketch its definition here.

The idea of  $\mathcal{DFOL}$  is to add a restricted version of dependent sorts in a way that preserves much of the meta-theory of first-order logic. A  $\mathcal{DFOL}$ -signature consists of a sequence of partial signatures, called levels. Each level is a sequence of typed symbol declarations. A declaration is of the form  $s : \Pi x : S_1. \dots. \Pi x_n : S_n. T$ ; here every  $S_i$  is an instance of a dependent sort, i.e., a sort symbol followed by argument terms according to its type, and every  $S_i$  may contain the variables  $x_1, \dots, x_{i-1}$ ; and  $T$  is one of the following three depending on whether  $s$  is a sort, function, or predicate symbol, respectively: *sort*, an instance of a dependent sort possibly containing the variables  $x_1, \dots, x_n$ , or *Form*.<sup>2</sup>

These declarations are subject to certain conditions. On level 0, only base sorts, i.e., sort symbols that do not take arguments, may be declared. Thus, level 0 is exactly a signature of many-sorted first-order logic. And on level  $n + 1$ , only sort symbols may be declared whose argument sorts contain only symbols from at most level  $n$ . Furthermore, a function symbol declared on level  $n + 1$  must have a return sort that is also declared on level  $n + 1$ .

For example, if level 0 declares a sort  $S : \text{sort}$ , level 1 may declare a sort  $S' : \Pi x : S. \text{sort}$  that depends on arguments of the level 0 sort  $S$ , and a function symbol  $f : \Pi x : S. S'(x)$ ; a function symbol declaration  $f' : \Pi x : S. \Pi y : S'(x). S$  is forbidden because  $f'$  would return a term of the level 0 sort  $S$  although there is an argument sort  $S'(x)$  of level 1. As usual when dealing with dependent types, we write  $\Pi x : S. S'$  as  $S \rightarrow S'$  if  $x$  does not occur freely in  $S'$ . We also write curried  $\rightarrow$  as  $\times$ , i.e.,  $S \rightarrow S' \rightarrow S''$  as  $S \times S' \rightarrow S''$ .

The atomic formulas over a  $\mathcal{DFOL}$ -signature  $\Sigma$  are given by an application of a predicate symbol to terms according to its type or by an application of equality to terms of the same sort. Then the sentences are built up in the same way as for many-sorted first-order logic. In particular, the sentences use the symbols  $\wedge$ ,  $\implies$ ,  $\equiv$  and  $\forall$ . The quantification has the additional subtlety that in formulas such as  $\forall x : S. \forall y : S(x). \varphi$ , the sorts of the variables may be dependent and may contain other variables. We define the level of a formula  $\phi$  to be the highest level of a symbol that occurs in  $\phi$ .

The models of a  $\mathcal{DFOL}$ -signature  $\Sigma$  are constructed level-wise. A model for the declarations at level 0 is a usual model of many-sorted first-order logic. To extend a model for level  $n$  to a model for level  $n + 1$ , every sort symbol of level  $n + 1$  is interpreted as an assignment of universes to the possible argument tuples according to its type. Then every function and predicate symbol of level 1 can be interpreted as a function or relation according to its type. Then satisfaction is defined in essentially the same way as for many-sorted first-order logic.

To formalize categories, we will use the following  $\mathcal{DFOL}$ -signature:

$Ob : \text{sort}$ .

$Des : Ob$ .

$Mor : Ob \times Ob \rightarrow \text{sort}$ .

---

<sup>2</sup>For simplicity, we omit the  $\mathcal{DFOL}$ -symbol *Univ* here.

$$\begin{aligned}
id : \Pi A : Ob. Mor(A, A). \\
; : \Pi A, B, C : Ob. Mor(A, B) \times Mor(B, C) \rightarrow Mor(A, C). \\
\rightsquigarrow : \Pi A, B : Ob. Mor(A, B) \times Mor(A, B) \rightarrow Form.
\end{aligned}$$

Here,  $Ob$  is a base sort, and  $Des$  is a constant of sort  $Ob$ . They are the only declaration on level 0. All remaining declarations are on level 1. Firstly,  $Mor$  is a dependent sort, its type means that for all terms  $A, B$  of sort  $Ob$ ,  $Mor(A, B)$  is a sort.  $id$  is a function symbol that takes an argument of type  $Ob$  and returns a term of a sort that depends on the argument. Similarly  $;$  takes three arguments of sort  $Ob$ , say  $A, B$  and  $C$ , and then two arguments of the sorts  $Mor(A, B)$  and  $Mor(B, C)$  and returns a term of sort  $Mor(A, C)$ .  $\rightsquigarrow$  is a predicate symbol: It takes two arguments of sort  $Ob$ , say  $A$  and  $B$ , and two arguments of sort  $Mor(A, B)$  and returns an atomic formula.

The intended semantics is that  $Ob$  is interpreted as the set of objects of a small category (via the Curry-Howard interpretation: formulas of a categorical logic),  $Mor(A, B)$  as the set of morphisms (proofs) from  $A$  to  $B$ ,  $id(A)$  as the identity (self-proof) of  $A$ ,  $;(A, B, C, f, g)$  as the composition of  $f : Mor(A, B)$  and  $g : Mor(B, C)$  (the proof of  $C$  from  $A$  obtained from applying cut with  $B$ ), and  $\rightsquigarrow (A, B, f, g)$  as a relation on morphisms that expresses that  $f \rightsquigarrow g$  (the reducibility of  $f$  to  $g$ ).  $Des$  singles out a certain object (the minimal designated truth value). For simplicity, we will write  $f;g$  instead of  $;(A, B, C, f, g)$  and  $f \rightsquigarrow g$  instead of  $\rightsquigarrow (A, B, f, g)$ . Similarly, we drop the arguments of sort  $Ob$  from other function symbols, like  $\pi_1$  below, if they are clear from the context.

To achieve the intended semantics, we define the  $\mathcal{DFOL}$ -theory  $CatLog$  by adding to the above signature the axioms

$$\begin{aligned}
\forall A, B, C, D : Ob \quad \forall f, f', f'' : Mor(A, B) \quad \forall g, g' : Mor(B, C) \quad \forall h : Mor(C, D) \\
id(A); f == f \\
h; id(D) == h \\
(f; g); h == f; (g; h) \\
f \rightsquigarrow f \\
f \rightsquigarrow f' \wedge f' \rightsquigarrow f'' \implies f \rightsquigarrow f'' \\
f \rightsquigarrow f' \wedge g \rightsquigarrow g' \implies f; g \rightsquigarrow f'; g'
\end{aligned}$$

Then we have indeed that the model category  $Mod^{\mathcal{DFOL}}(CatLog)$  is essentially  $\mathbb{O}rdCat$ . Precisely, a model of  $CatLog$  is a preorder-enriched category with one distinguished object interpreting  $Des$ .

We can axiomatize other enriched categories by adding function symbols to  $CatLog$ . We permit the addition of two kinds of declarations. Firstly, function symbols on level 0; these simply take  $n$  terms of sort  $Ob$  and return a term of sort  $Ob$ . Such function symbols represent propositional connectives. And secondly, function symbols on level 1; these must take terms of sort  $Ob$  or an instance of  $Mor$  and return a term of a sort that is an instance of  $Mor$ . Such function symbols represent proof rules. Furthermore, we permit certain axioms that represent equalities and rewrites between proofs. The axioms must be in Horn form in order to apply the free model theorem for  $\mathcal{DFOL}$  ([32]). Formally, we define this as follows.

**Definition 3.1.** A *categorical logic*  $\mathcal{L}$  is an extension of  $CatLog$  with function symbols and axioms such that

- axioms are in Horn form
- if an axiom is of level 1, its head must be an equality or rewrite between morphisms

- there are congruence axioms for  $\rightsquigarrow$ , i.e., for every new function symbol of the form

$$c : \Pi \vec{A} : Ob. Mor(S_1, T_1) \rightarrow \dots \rightarrow Mor(S_n, T_n) \rightarrow Mor(S, T)$$

where  $\vec{A}$  abbreviates  $A_1, \dots, A_m$  and  $S_i, T_i, S, T$  are terms of the sort  $Ob$  with free variables  $\vec{A} : Ob$ , there is an axiom

$$\begin{aligned} \forall \vec{A} : Ob \quad & \forall f_1, f'_1 : Mor(S_1, T_1) \dots \forall f_n, f'_n : Mor(S_n, T_n) \\ (f_1 \rightsquigarrow f'_1 \wedge \dots \wedge f_n \rightsquigarrow f'_n) & \implies c(\vec{A}, f_1, \dots, f_n) \rightsquigarrow c(\vec{A}, f'_1, \dots, f'_n) \end{aligned}$$

The category of categorical logics, denoted by  $CatLog$ , has such theories  $\mathcal{L}$  as objects and  $\mathcal{DFOL}$ -theory morphisms that are the identity for the symbols of  $CatLog$  as morphisms. (More generally, we could use the slice category of  $\mathcal{DFOL}$ -theories with domain  $CatLog$ .)

Then for any categorical logic  $\mathcal{L}$ , an element of  $Mod^{\mathcal{DFOL}}(\mathcal{L})$  consists of an element of  $\mathbb{O}rdCat$  along with interpretations for the symbol  $Des$  and the added function symbols.  $Mod^{\mathcal{DFOL}}(\mathcal{L})$ -morphisms are functors in  $\mathbb{O}rdCat$  that preserve the interpretation of  $Des$  and that commute with the added function symbols. We call this category  $\mathbb{C}(\mathcal{L})$ , its elements  $\mathcal{L}$ -categories, and its morphisms  $\mathcal{L}$ -functors. In an  $\mathcal{L}$ -category  $A$ , the preorders on all hom-sets are denoted by  $\rightsquigarrow^A$ ; and the interpretation of a function symbol  $c$  is denoted by  $c^A$ . Then the usual notion of (small) enriched categories becomes a special case of  $\mathcal{L}$  categories.

We have the following result:

**Proposition 3.2.** *For every categorical logic  $\mathcal{L}$  and every set  $X$  of variables of object or morphism sort, there is a free model  $T_{\mathcal{L}}(X) \in |\mathbb{C}(\mathcal{L})|$ , i.e., for every  $A \in |\mathbb{C}(\mathcal{L})|$  and every mapping  $m$  that maps variables in  $X$  to objects or morphisms of  $A$  according to their sort, there is a unique extension  $\bar{m}$  of  $m$  to an  $\mathcal{L}$ -functor  $T_{\mathcal{L}}(X) \rightarrow A$ .*

*Proof.* This is a special case of the result proven in [32]. Therefore, we only give the idea of the proof. In the first step, only the sort  $Ob$ , the variables of sort  $Ob$ , the function symbols on level 0, and all axioms of level 0 are considered. These induce a free term model in the usual way of first-order logic. Then in a second step, all elements of the universe of this free model are added to the signature as constants; and then another term model construction can be used to obtain the desired model.

For the correctness of the construction, it is crucial that there are no function symbols of level 1 that return a term of sort  $Ob$ , and no axioms of level 1 that might imply an equality between terms of the  $Ob$ ; this ensures that the model found in the first step is not influenced by the second step.  $\square$

The objects and morphisms of  $T_{\mathcal{L}}(X)$  are equivalence classes of terms. For simplicity, we use every term as an abbreviation for its equivalence class in  $T_{\mathcal{L}}(X)$ .

**Example 3.3** (Cartesian Categories). The categorical logic *Cartesian* arises by extending  $CatLog$  with declarations

$$\begin{aligned} \top & : Ob \\ * & : Ob \times Ob \rightarrow Ob \\ ! & : \Pi A : Ob. Mor(A, \top) \\ \langle \_, \_ \rangle & : \Pi A, B, C : Ob. Mor(A, B) \times Mor(A, C) \rightarrow Mor(A, B * C) \\ \pi_1 & : \Pi A, B : Ob. Mor(A * B, A) \\ \pi_2 & : \Pi A, B : Ob. Mor(A * B, B) \end{aligned}$$

and axioms

$$\begin{aligned} f : A \rightarrow \top &\rightsquigarrow ! \\ \langle f; \pi_1, f; \pi_2 \rangle &\rightsquigarrow f \\ \langle f, g \rangle; \pi_1 &\rightsquigarrow f \\ \langle f, g \rangle; \pi_2 &\rightsquigarrow g \\ \langle f; g, f; h \rangle &\rightsquigarrow f; \langle g, h \rangle \end{aligned}$$

making  $\top$  a terminal object and  $*$  a binary product with projections  $\pi_1$  and  $\pi_2$ . Logically, this corresponds to adding “true” ( $\top$ ) and conjunction ( $*$ ). The first three rewrite axioms axiomatize the elimination of redundant proof steps, and the last rewrite axiom is a step in the cut (i.e., composition ; ) elimination.

Note that we have some freedom in specifying the axioms. For example, we can either add the axiom  $\langle f, g \rangle; \pi_1 == f$  (omitting the universal quantifiers), which identifies the two proof terms and hence, regards their difference as merely bureaucratic and caused by the needs of syntactic representation. Or we can specify the rewrite rule  $\langle f, g \rangle; \pi_1 \rightsquigarrow f$ , which keeps  $\langle f, g \rangle; \pi_1$  and  $f$  distinct.

Extensions  $\mathcal{L}$  of *Cartesian* are called *cartesian*. If  $\mathcal{L}$  also has the axiom  $Des == \top$ ,  $\mathcal{L}$  is called  $\top$ -*cartesian*. We will give further examples in Sect. 5.

## 4 The Institutional Curry-Howard-Tait Construction

Now we construct a preorder-enriched institution out of a categorical logic, thereby following the Curry-Howard-Tait isomorphism paradigm. First, given a categorical logic  $\mathcal{L}$ , we define two theory extensions: Let  $\bar{\mathcal{L}}$  be the extension of  $\mathcal{L}$  with the axiom

$$\forall A, B : Ob \ \forall f, g : Mor(A, B) (f \rightsquigarrow g \implies f == g),$$

which, semantically, quotients out the preorder in the  $\mathcal{L}$ -categories, and let  $\mathcal{L}_{thin}$  be the extension of  $\bar{\mathcal{L}}$  with the axiom

$$\forall A, B : Ob \ \forall f, g : Mor(A, B) f == g,$$

which, semantically, quotients the  $\mathcal{L}$ -categories to preorders. Let  $\overline{\mathbb{C}(\mathcal{L})} = \mathbb{C}(\bar{\mathcal{L}})$  and  $\mathbb{C}_{thin}(\mathcal{L}) = \mathbb{C}(\mathcal{L}_{thin})$ .

**Definition 4.1.** Given a categorical logic  $\mathcal{L}$ , the preorder-enriched institution  $\mathcal{I}(\mathcal{L}) = (\mathbb{S}ign, \mathbf{Pr}, \mathbf{Mod}, \models)$  is defined by:

- $\mathbb{S}ign$  is the category of sets (seen as sets of propositional variables).
- $\mathbf{Pr}$  is the universal functor of Prop. 3.2. Explicitly,  $\mathbf{Pr}(\Sigma) = T_{\mathcal{L}}(\Sigma)$  for a signature  $\Sigma$ , and for a signature morphism (that is, a function)  $\sigma : \Sigma \rightarrow \Sigma'$ ,  $\mathbf{Pr}(\sigma)$  is the unique extension of  $\sigma$  to an  $\mathcal{L}$ -functor from  $T_{\mathcal{L}}(\Sigma)$  to  $T_{\mathcal{L}}(\Sigma')$ .
- $\mathbf{Mod}$  is the lax comma category functor  $(\mathbf{Pr}(\cdot) \downarrow \overline{\mathbb{C}(\mathcal{L})})$ . Explicitly,  $\mathbf{Mod}$  assigns to a signature  $\Sigma$  a category  $\mathbf{Mod}(\Sigma)$  such that
  - objects are pairs  $(A, m)$  where  $A \in |\overline{\mathbb{C}(\mathcal{L})}|$  and  $m$  is a valuation  $m : \Sigma \rightarrow |A|$  of the propositional variables to  $A$ -objects,<sup>3</sup>

- model morphisms from  $(A, m)$  to  $(A', m')$  are pairs  $(F, \mu)$  where  $F: A \rightarrow A'$  is an  $\mathcal{L}$ -functor and  $\mu$  is a family of  $A'$ -morphisms over  $p \in \Sigma$  such that  $\mu_p: F(m(p)) \rightarrow m'(p)$ ,

and for a signature morphism  $\sigma: \Sigma \rightarrow \Sigma'$ , the model reduct functor  $\text{Mod}(\sigma)$  is given by composition:  $\text{Mod}(\sigma)(A, m) = (A, m \circ \sigma)$  for models and  $\text{Mod}(\sigma)(F, \mu) = (F, \mu \circ \sigma)$  for model morphisms.

- Satisfaction is defined by:  $(A, m) \models_{\Sigma} \varphi$  iff  $A(\text{Des}^A, \bar{m}(\varphi))$  is inhabited.

For example, intuitionistic and classical propositional logic are both captured by this construction. A detailed discussion of examples can be found in Sect. 5.

**Proposition 4.2** (Institutionality). *For any categorical logic  $\mathcal{L}$ ,  $\mathcal{I}(\mathcal{L})$  is a preorder-enriched institution.*

*Proof.* The satisfaction condition is shown as follows. Let  $\sigma: \Sigma \rightarrow \Sigma'$  be a morphism, let  $\varphi$  be a  $\Sigma$ -sentence, and let  $(A', m')$  be a  $\Sigma'$ -model. Then  $(A', m')|_{\sigma} \models_{\Sigma} \varphi$  iff  $A'(\text{Des}^{A'}, \overline{m' \circ \sigma}(\varphi)) \neq \emptyset$  iff  $A'(\text{Des}^{A'}, \overline{m'}(\sigma(\varphi))) \neq \emptyset$  iff  $(A', m') \models_{\Sigma'} \sigma(\varphi)$ .  $\square$

For a signature  $\Sigma$ , the Curry-Howard-Tait correspondence then takes the following shape:

**Propositions as types/objects** Sentences are the objects of  $\text{Pr}(\Sigma)$ , i.e., sentences are  $\mathcal{L}$ -terms of sort  $Ob$  with propositional variables from  $\Sigma$ .

**Proofs as terms** Proofs are the morphisms of  $\text{Pr}(\Sigma)$ . That is, a  $\Sigma$ -proof between sentences  $\varphi$  and  $\psi$  is simply an equivalence class of  $\mathcal{L}$ -terms of sort  $Mor(\varphi, \psi)$ . If the only equations in  $\mathcal{L}$  are those of *CatLog*, this means that  $\Sigma$ -proofs are strings of composable composition-free proof terms.

**Proof reduction as morphism ordering** The reducibility of proofs is given by the  $\rightsquigarrow$  predicate in  $T_{\mathcal{L}}(\Sigma)$ , which is a preorder on morphisms (which is preserved under composition).

**Categorical models** A  $\Sigma$ -model is determined by a category  $A$  in  $[\overline{\mathbb{C}(\mathcal{L})}]$  and a valuation of the propositional variables into  $A$ . Proof reduction is modeled by the interpretation of  $\rightsquigarrow$ .

**Satisfaction via designated truth values** For a  $\Sigma$ -model  $(A, m)$ , the preorder  $\text{thin}(A)$  gives the truth values of  $A$ . Then the set of designated truth values is the upper set of  $\text{Des}^A$  in  $\text{thin}(A)$ . (Defining more complex sets of designated truth values is possible by making  $\text{Des}$  a predicate on objects instead of a constant, as in [2].)

In the remainder of this section, we derive some crucial properties of this construction.

**Proposition 4.3** (Soundness). *For every categorical logic  $\mathcal{L}$ ,  $\mathcal{I}(\mathcal{L})$  is sound.*

*Proof.* Let  $\phi \vdash_{\Sigma} \psi$ , i.e. we have a morphism  $p: \phi \rightarrow \psi$  in  $\text{Pr}(\Sigma)$ . Let  $(A, m)$  be a  $\Sigma$ -model such that  $(A, m) \models \phi$ , i.e.  $A(\text{Des}^A, \bar{m}(\phi)) \neq \emptyset$ . Postcomposition with the morphism  $\bar{m}(p): \bar{m}(\phi) \rightarrow \bar{m}(\psi)$  yields  $A(\text{Des}^A, \bar{m}(\psi)) \neq \emptyset$ , i.e.  $(A, m) \models \psi$ .  $\square$

---

<sup>3</sup>Remember that such a valuation uniquely determines an  $\mathcal{L}$ -functor  $\bar{m}$  from  $\text{Pr}(\Sigma)$  to  $A$ .

For a categorical logic  $\mathcal{L}$ , we put  $\vdash_{\Sigma} \psi$  iff  $\text{Pr}(\Sigma)(\text{Des}, \psi)$  is non-empty. It is trivial to note that  $\mathcal{I}(\mathcal{L})$  is *weakly sound*, i.e.  $\vdash_{\Sigma} \psi$  implies  $\emptyset \models_{\Sigma} \psi$ . We say that  $\mathcal{I}(\mathcal{L})$  is *weakly complete* if the converse implication holds.

When  $\mathcal{L}$  is cartesian, then we have a binary connective  $*$  (conjunction) on sentences of  $\mathcal{I}(\mathcal{L})$  arising from binary product  $\times$ ; moreover, the terminal object  $\top$  induces a sentence  $\top$  (truth). We can then extend the entailment relation  $\vdash$  to sets of hypotheses by putting  $\Phi \vdash_{\Sigma} \psi$  if there exist  $\varphi_1, \dots, \varphi_n \in \Phi$  such that

$$\varphi_1 * \dots * \varphi_n \vdash_{\Sigma} \psi$$

where we put  $\varphi_1 * \dots * \varphi_n = \top$  if  $n = 0$ . In this notation,  $\phi \vdash_{\Sigma} \psi$  is equivalent to  $\{\phi\} \vdash_{\Sigma} \psi$ , and  $\vdash_{\Sigma} \psi$  is equivalent to  $\emptyset \vdash_{\Sigma} \psi$ . In this setting, we say that  $\mathcal{I}(\mathcal{L})$  is *strongly sound* if  $\Phi \vdash_{\Sigma} \psi$  implies  $\Phi \models_{\Sigma} \psi$ , that  $\mathcal{I}(\mathcal{L})$  is *strongly complete* if the converse implication holds.

**Proposition 4.4** (Strong soundness). *For any cartesian categorical logic  $\mathcal{L}$ ,  $\mathcal{I}(\mathcal{L})$  is strongly sound.*

*Proof.* By Proposition 4.3, it suffices to show that  $\Phi \models_{\Sigma} \varphi_1 * \dots * \varphi_n$  for all  $\varphi_1, \dots, \varphi_n \in \Phi$ . If  $(A, m) \models \Phi$  for a model  $(A, m)$ , then we can pick  $q_i \in A(\text{Des}^A, \bar{m}(\phi_i))$  for  $i = 1, \dots, n$ . Then  $\langle q_1, \dots, q_n \rangle \in A(\text{Des}^A, \bar{m}(\phi_1) * \dots * \bar{m}(\phi_n)) \cong A(\text{Des}^A, \bar{m}(\phi_1 * \dots * \phi_n))$  and hence  $(A, m) \models \phi_1 * \dots * \phi_n$ .  $\square$

It is important to recognize that, unlike the categories  $A$  in  $\Sigma$ -models  $(A, m)$ , the proof categories  $\text{Pr}(\Sigma)$  are generally *not* elements of  $\overline{\mathbb{C}(\mathcal{L})}$ . This avoids the problem stated in [26] that for classical propositional logic, the categorical semantics of proofs collapses: All classical bicartesian closed categories are partial orders (and thus boolean algebras). If one does not wish to distinguish between rewritable proofs, one can collapse the proof categories by using  $\mathcal{I}(\mathcal{L}) := \mathcal{I}(\overline{\mathcal{L}})$ . Proof irrelevance is obtained by using  $\mathcal{I}_{\text{thin}}(\mathcal{L}) := \mathcal{I}(\mathcal{L}_{\text{thin}})$ .

**Definition 4.5** (Deduction Theorem). We say that a categorical logic  $\mathcal{L}$  satisfies the *weak deduction theorem* if for every  $\Sigma$ -proof term  $p : \text{Mor}(\text{Des}, \psi)$  in one free variable  $x : \text{Mor}(\text{Des}, \chi)$ , there exists a closed  $\Sigma$ -proof term  $\kappa x.p$  of type  $\text{Mor}(\chi, \psi)$ . If  $\mathcal{L}$  is cartesian, then  $\mathcal{L}$  satisfies the *deduction theorem* if for every  $\Sigma$ -proof term  $p : \text{Mor}(\varphi, \psi)$  in free variables  $X$ ,  $x : \text{Mor}(\text{Des}, \chi)$ , there exists a  $\Sigma$ -proof term  $\kappa x.p$  of type  $\text{Mor}(\varphi * \chi, \psi)$  in free variables  $X$ .

**Theorem 4.6** (Completeness). *For every categorical logic  $\mathcal{L}$ ,  $\mathcal{I}(\mathcal{L})$  is weakly complete. If  $\mathcal{L}$  satisfies the weak deduction theorem, then  $\mathcal{L}$  is complete. If  $\mathcal{L}$  is  $\top$ -cartesian and satisfies the deduction theorem, then  $\mathcal{I}(\mathcal{L})$  is strongly complete.*

*Proof.* We prove only the third claim; the first two follow by analogous, but simpler arguments. Assume  $\Phi \models_{\Sigma} \psi$ . Let  $F \in |\text{Mod}^{\mathcal{DFOL}}(\mathcal{L})|$  be the free  $\mathcal{L}$ -category existing by Prop. 3.2 over the following sorted variables: variables  $v : \text{Ob}$  for every  $v \in \Sigma$  and variables  $x_{\varphi} : \text{Mor}(\top, \varphi)$  for all  $\varphi \in \Phi$ . Then  $F \models_{\Sigma} \varphi$  for all  $\varphi \in \Phi$ . Hence, by the assumption of the theorem, we have  $F \models_{\Sigma} \psi$ . Then, since  $F$  is a free term model differing from  $\text{Pr}(\Sigma)$  only by having more variables, there must be a  $\text{Pr}(\Sigma)$ -term  $p(x_{\varphi}, \dots) : \text{Mor}(\top, \psi)$  in free variables  $x_{\varphi}$ . Clearly,  $p$  can only refer to finitely many  $x_{\varphi}$ . Thus repeated application of the deduction theorem yields a closed  $\text{Pr}(\Sigma)$ -term of type  $\text{Mor}(\top * \varphi_1 * \dots * \varphi_n, \psi)$ , and precomposition with the isomorphism  $\varphi_1 * \dots * \varphi_n \cong \top * \varphi_1 * \dots * \varphi_n$  yields  $\phi_1 * \dots * \phi_n \vdash \psi$  and hence  $\Phi \vdash \psi$ .  $\square$

It should be noted that the fact that this general completeness theorem has a rather easy proof is caused by its restricted applicability, mainly to propositional logics (there is no handling of variables and quantification). Even for most modal propositional logics, the essential assumption for the second result, namely the deduction theorem, fails. However, there is an easy criterion for obtaining the deduction theorem:

**Proposition 4.7.** *The deduction theorem holds in a  $\top$ -cartesian categorical logic provided that newly introduced function symbols that return morphisms do not take morphisms as arguments (that is, there are no new logical rules).*

*Proof.* See Proposition I.2.1 of [26]. The addition of operations adhering to the above restriction does not destroy the induction proof given there.  $\square$

**Example 4.8.** Let  $\text{Cartesian}_{\text{top}}$  be the  $\top$ -cartesian categorical logic arising by only adding  $\text{Des} == \top$  to  $\text{Cartesian}$ . The institution  $\mathcal{I}(\text{Cartesian}_{\text{top}})$  is described as follows. Signatures are sets of propositional variables. Sentences are the corresponding fragment of propositional logic. A model consists of a category together with an interpretation of the propositional variables as objects in this category. The conjunction is interpreted by a product, and truth by a terminal element. Evaluation of sentences is just term evaluation in the category. The designated truth values are  $\top$  and everything provable from it.

If  $p: C \rightarrow A$  and  $q: C \rightarrow B$  are proofs in a model  $(A, m)$ , they can be combined to  $\langle p, q \rangle^A: C \rightarrow A *^A B$ , and  $\langle p, q \rangle^A; \pi_1^A$  is another proof from  $C$  to  $A$ . The rewriting structure gives us  $\langle p, q \rangle^A; \pi_1^A \rightsquigarrow^A p$ . In the institution  $\overline{\mathcal{I}(\text{Cartesian}_{\text{top}})}$ , these two proofs are identified.

We arrive at cartesian categories ([26]) if we quotient out rewrites, i.e., use  $\mathcal{I}_{\text{thin}}(\text{Cartesian}_{\text{top}})$ .

We can define cut elimination in the context of categorical logic ([15]): The cut rule corresponds to the composition of morphisms. Then *cut elimination* means that the composition operation can be eliminated from proof terms.

**Definition 4.9.** A categorical logic  $\mathcal{L}$  *admits cut* if for all signatures  $\Sigma$  and all objects  $A, B \in |T_{\mathcal{L}}(\Sigma)|$ , for every  $\Sigma$ -proof term  $p: \text{Mor}(A, B)$  there is a  $\Sigma$ -proof term  $p': \text{Mor}(A, B)$  such that  $p'$  does not contain the function symbol  $\text{;}$ .  $\mathcal{L}$  has cut elimination if, in addition,  $p \rightsquigarrow p'$  holds in  $T_{\mathcal{L}}(\Sigma)$ .

This analogy is not perfect, however, because not in all logics, the more complex cut rules with multiple formulas on each side of the turnstile can be derived from this simple version of cut. A solution might be to use polycategories ([41, 3]), but this is beyond the scope of this work.

Both cut admissibility and cut elimination need to be established independently for every categorical logic: Minor changes in the specification can destroy these properties, or require redoing large portions of their proofs. While some of the above examples have cut admissibility, additional rewrites are necessary to establish cut elimination. These additional rewrites correspond to the various cases and subcases of the induction step of a constructive cut elimination proof.

**Proposition 4.10.** *Cartesian has cut elimination.*

*Proof.* The proof proceeds by nested term inductions on  $f$  and  $g$  in  $f; g$ . This result can also be found in [15].  $\square$

Exactness of institutions is a property important for modular specifications and proofs [35]:

**Definition 4.11.** An institution is said to be *semi-exact*, if any pushout

$$\begin{array}{ccc} \Sigma & \longrightarrow & \Sigma_1 \\ \downarrow & & \downarrow \\ \Sigma_2 & \longrightarrow & \Sigma_R \end{array}$$

in  $\mathbb{S}ign$  is mapped by  $\mathbf{Mod}$  to a pullback

$$\begin{array}{ccccc} \mathbf{Mod}(\Sigma) & \longleftarrow & \mathbf{Mod}(\Sigma_1) & & \\ \uparrow & & \uparrow & & \\ \mathbf{Mod}(\Sigma_2) & \longleftarrow & \mathbf{Mod}(\Sigma_R) & & \end{array}$$

of categories. Explicitly, this means that any pair  $(M_1, M_2) \in \mathbf{Mod}(\Sigma_1) \times \mathbf{Mod}(\Sigma_2)$  that is *compatible* in the sense that  $M_1$  and  $M_2$  reduce to the same  $\Sigma$ -model can be *amalgamated* to a unique  $\Sigma_R$ -model  $M$  (i.e., there exists a unique  $M \in \mathbf{Mod}(\Sigma_R)$  that reduces to  $M_1$  and  $M_2$ , respectively), and similarly for model morphisms.

**Proposition 4.12.** *For any categorical logic  $\mathcal{L}$ ,  $\mathcal{I}(\mathcal{L})$  is semi-exact.*

*Proof.* This follows immediately because the model reduct functor  $\mathbf{Mod}(\sigma)$  is defined by composition of valuations with signature morphisms.  $\square$

**Definition 4.13.** An institution is said to be *(weakly) liberal*, if the reduct functor of each *theory morphism*  $\sigma: (\Sigma_1, \Psi_1) \rightarrow (\Sigma_2, \Psi_2)$  has a (weak) left adjoint. A functor  $F$  is a weak left adjoint to  $U$  via unit  $\eta: Id \rightarrow UF$ , if any morphism  $f: X \rightarrow UA$  factors (not necessarily uniquely) as  $Ug \circ \eta_X$  for some  $g: FX \rightarrow A$ .

$\mathcal{I}(IProp)$  and  $\mathcal{I}(Prop)$  are not liberal: the theory  $(\{A, B\}, \{A + B\})$  (viewed as extension of the empty theory) has no free model. However, we have

**Proposition 4.14.** *For any categorical logic  $\mathcal{L}$  with truth,  $\mathcal{I}(\mathcal{L})$  is weakly liberal.*

*Proof.* Given a  $\Sigma_1$ -model  $(A, m)$ , the  $(\Sigma_2, \Psi_2)$ -diagram of  $(A, m)$  is obtained as follows. Add all objects of  $A$  as propositional variables to  $\Sigma_2$ , arriving at the signature  $\Sigma_2(A)$ ;  $(A, m)$  is easily extended to a model of that signature. Add all  $\Sigma_2(A)$ -sentences holding in  $(A, m)$  to  $\Psi_2$ , obtaining a theory  $\Psi_2(A)$ . By Prop. 3.2, there is a free  $\mathcal{L}(\Sigma_2(A) \cup \{x_\varphi : Mor(\top, \varphi) \mid \varphi \in \Psi_2(A)\})$ -model. It canonically is a  $\Sigma_2$ -model, which is a weakly free extension of  $(A, m)$ .  $\square$

**Proposition 4.15.** *The construction  $\mathcal{I}(-)$  is functorial, i.e., it can be extended to a functor  $\mathcal{I}(-): \mathbb{CatLog} \rightarrow \mathbb{CoIns}$ .*

*Proof.* Given two categorical logics  $\mathcal{L}_1$  and  $\mathcal{L}_2$  and a  $\mathcal{DFOL}$ -theory morphism  $\sigma: \mathcal{L}_1 \rightarrow \mathcal{L}_2$ , we can construct a preorder-enriched institution comorphism  $(\Phi, \alpha, \beta): \mathcal{I}(\mathcal{L}_1) \rightarrow \mathcal{I}(\mathcal{L}_2)$  as follows:

- $\Phi$  is the identity functor in the category  $\mathbb{Set}$ .

- To define  $\alpha_\Sigma: \text{Pr}^{\mathcal{L}_1}(\Sigma) \rightarrow \text{Pr}^{\mathcal{L}_2}(\Phi(\Sigma))$ , first note that  $\Phi(\Sigma) = \Sigma$ , and remember that the objects and morphisms of  $\text{Pr}^{\mathcal{L}_i}$  are equivalence classes  $[t]$  of terms  $t$  over  $\mathcal{L}_i$ . But since the  $\text{Sen}^{\mathcal{DFOL}}(\sigma)$ -images of  $\mathcal{L}_1$ -axioms are consequences of the  $\mathcal{L}_2$ -axioms, all elements of such an equivalence class of  $\mathcal{L}_1$  are mapped to the same equivalence class in  $\mathcal{L}_2$ . Therefore, for an object or morphism  $[t]$  of  $\text{Pr}^{\mathcal{L}_1}(\Sigma)$ ,  $\alpha_\Sigma$  is well-defined by putting  $\alpha_\Sigma([t]) = [\text{Sen}^{\mathcal{DFOL}}(\sigma)(t)]$ .  $\alpha_\Sigma$  preserves the preordering on  $\text{Pr}^{\mathcal{L}_1}(\Sigma)$ -morphisms because the preordering is defined by rewrite axioms of  $\mathcal{L}_1$ , which  $\text{Sen}^{\mathcal{DFOL}}(\sigma)$  maps to consequences of  $\mathcal{L}_2$ -axioms.
- $\beta_\Sigma: \text{Mod}^{\mathcal{L}_2}(\Phi(\Sigma)) \rightarrow \text{Mod}^{\mathcal{L}_1}(\Sigma)$  is obtained as follows. Define  $\mathcal{L}_i^*$  by adding the elements of  $\Sigma$  as constants of sort  $Ob$  to  $\mathcal{L}_i$ ; then define  $\sigma^*$  by extending  $\sigma$  such that the new constants are mapped to themselves. Clearly,  $\sigma^*$  is a theory-morphism from  $\mathcal{L}_1^*$  to  $\mathcal{L}_2^*$ . Then a model  $(A', m') \in |\text{Mod}^{\mathcal{L}_2}(\Sigma)|$  induces a model  $M' \in |\text{Mod}^{\mathcal{DFOL}}(\mathcal{L}_2^*)(\Phi(\Sigma))|$ ;  $M'$  is obtained by taking  $A'$  and interpreting each of the new constants  $c \in \Sigma$  as  $m'(c)$ . Then the model reduction of  $\mathcal{DFOL}$  yields a model  $M \in |\text{Mod}^{\mathcal{DFOL}}(\mathcal{L}_1^*)|$ , i.e.,  $M = \text{Mod}^{\mathcal{DFOL}}(\sigma^*)(M')$ . Finally,  $M$  induces a model  $(A, m) \in |\text{Mod}^{\mathcal{L}_1}(\Sigma)|$ ;  $A$  arises from  $M$  by forgetting the interpretations of the new constants, and  $m$  is the interpretation of the new constants in  $M$ . Thus we put  $\beta_\Sigma(A', m') = (A, m)$ . The action of  $\beta_\Sigma$  on morphisms is defined similarly.
- The proof of the satisfaction condition is straightforward.

□

**Proposition 4.16.** *Let  $\mathcal{L}$  be a categorical logic. There are preorder-enriched institution comorphisms from  $\mathcal{I}(\mathcal{L})$  to  $\overline{\mathcal{I}(\mathcal{L})}$  and from  $\overline{\mathcal{I}(\mathcal{L})}$  to  $\mathcal{I}_{thin}(\mathcal{L})$ . In particular, semantic consequence is the same in the three institutions.*

*Proof.* The comorphisms  $\mathcal{I}(\mathcal{L}) \rightarrow \overline{\mathcal{I}(\mathcal{L})}$  and  $\overline{\mathcal{I}(\mathcal{L})} \rightarrow \mathcal{I}_{thin}(\mathcal{L})$  are obtained if Prop. 4.15 is applied to the obvious theory morphisms  $\mathcal{L} \rightarrow \overline{\mathcal{L}}$  and  $\overline{\mathcal{L}} \rightarrow \mathcal{L}_{thin}$ , respectively.

Because the translation of signatures and sentences of the comorphisms is the identity, the claim that semantic consequence coincides is well-defined. And it is easy to prove: Satisfaction is defined via the existence of certain morphisms and the model translations of the comorphisms only identify existing morphisms. □

## 5 Examples

Our framework permits the use of  $\mathcal{DFOL}$ -theory morphisms to obtain a precise semantics for parametric and modular specifications. Fig. 4 gives a hierarchy of several examples of modular categorical logic specifications. Nodes are categorical logics, the solid arrows are  $\mathcal{DFOL}$ -theory inclusions, and all rectangles are pushouts.

The pushout of two  $\mathcal{DFOL}$ -theory inclusions is easy to construct. A theory inclusion  $\sigma_i: T \rightarrow T_i$  arises by adding declarations  $D_i$  and axioms  $A_i$  to  $T$  for  $i = 1, 2$ . Then a pushout of  $\sigma_1$  and  $\sigma_2$  arises by adding declarations  $D_1$  and  $D_2$  and axioms  $A_1$  and  $A_2$  to  $T$ . Since  $\mathcal{DFOL}$  is realized as a signature in the logical framework LF ([23]), this is a special case of the construction given in [24] for LF.

By Prop. 4.2 and Prop. 4.15, all nodes and edges of Fig. 4 induce preorder-enriched institutions and comorphisms between them.

For example, we obtain classical  $S4$  as a pushout of classical logic  $Prop$  and intuitionistic modal logic  $IS4$  over intuitionistic logic  $IProp$ .

All specifications are available in the input syntax of Twelf, an implementation of LF ([31]), such that they can be type-checked automatically. They can be obtained at [33].

Furthermore, we could reuse a specification for monoidal comonads in the specification of both modal and linear logic: The dotted arrows are theory morphisms that are not inclusions but instantiations. However, a module system for Twelf that could handle these instantiations is not implemented yet.

**Example 5.1** (Cartesian closed logic). *CartClosed* specifies implication  $\rightarrow$  as an exponential object by adding to *Cartesian<sub>top</sub>* the declarations

$\rightarrow : Ob \times Ob \rightarrow Ob$ .

$eval : \prod B, C : Ob. Mor((B \rightarrow C) * B, C)$ .

$curry : \prod A, B, C : Ob. Mor(A * B, C) \rightarrow Mor(A, B \rightarrow C)$ .

and the axioms

$\forall f : Mor(A * B, C). \langle \pi_1; curry(f), \pi_2; id(B) \rangle; eval(B, C) \rightsquigarrow f$ .

$\forall f : Mor(A, B \rightarrow C). curry(\langle \pi_1; f, \pi_2; id(B) \rangle); eval(B, C) \rightsquigarrow f$ .

**Example 5.2** (Intuitionistic logic). The theory *Cocartesian* specifies disjunction  $+$  as a coproduct and falsity  $\perp$  as an initial object by adding to *CatLog* the declarations

$\perp : Ob$ .

$!! : Mor(\perp, A)$ .

$+ : Ob \times Ob \rightarrow Ob$ .

$inl : \prod A, B : Ob. Mor(A, A + B)$ .

$inr : \prod A, B : Ob. Mor(B, A + B)$ .

$[_,_] : Mor(A, C) \times Mor(B, C) \rightarrow Mor(A + B, C)$ .

and the axioms

$\forall f : Mor(\perp, A). f \rightsquigarrow !!$ .

$\forall f : Mor(A + B, C). [inl(A, B); f], [inr(A, B); f] \rightsquigarrow f$ .

$\forall f : Mor(A, C) \forall g : Mor(B, C). inl(A, B); [f, g] \rightsquigarrow f$ .

$\forall f : Mor(A, C) \forall g : Mor(B, C). inr(A, B); [f, g] \rightsquigarrow g$ .

Combining *Cartesian* and *Cocartesian* by a pushout yields *Bicartesian*, the theory of bicartesian closed categories ([26]). To obtain intuitionistic logic *IProp*, we take the pushout of *Bicartesian* and *CartClosed* (as morphisms out of *Cartesian*). Then we only need to add the abbreviation  $-A := A \rightarrow \perp$  to define negation. In *IProp*, there are already terms of the sorts  $Mor(A * -A, \perp)$  and  $Mor(A, --A)$ .

**Example 5.3** (Classical logic). Several possibilities exist to extend *IProp* to classical logic (see e.g., [16]). We define *Prop* by adding an operation  $tnd : \prod A : Ob. Mor(\top, A + -A)$ . We also add rewrite axioms that make proofs from  $A$  to  $A$  via  $--A$  reducible to  $id(A)$ .

**Example 5.4** (Intuitionistic S4). Following [6], we extend *IProp* to *IS4* by adding operations  $\square, \diamond : Ob \rightarrow Ob$  and other operations on morphisms modeling the necessity and possibility operators. The necessity modality  $\square$  is interpreted as a monoidal comonad, while the possibility modality  $\diamond$  is interpreted as a monad, which is strong relative to the necessity comonad.

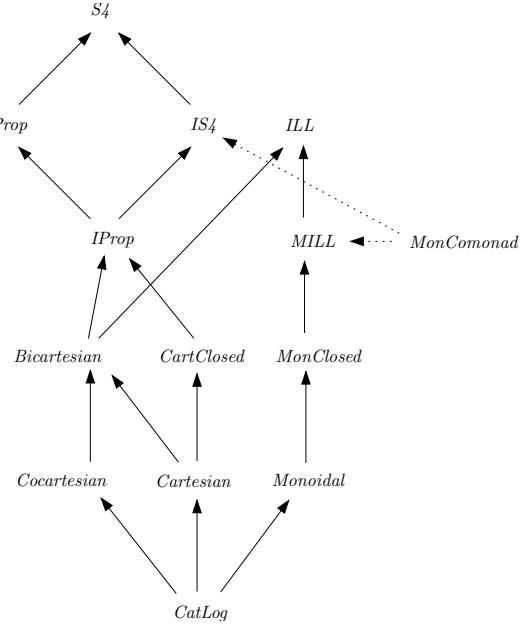


Figure 4: Graph of Logics

In all the above examples, we have the axiom  $Des == \top$ , which expresses that  $\top$  is the minimal designated truth value. Since  $\top$  is also a terminal element, the models have (up to isomorphism) the greatest truth value as the only designated one.

**Example 5.5** (Linear logic). For a different kind of categorical logic, which does not build up from *CartClosed*, but simply from *CatLog*, consider multiplicative intuitionistic linear logic *MILL* as in [4]. Multiplicative conjunction and linear implication are interpreted as a symmetric monoidal closed category. The of-course operator  $!$  is interpreted as a comonad, and each object  $!A$  is equipped with a comonoid structure such that the comonoid maps are also coalgebra maps. We add the axiom  $Des == I$ , i.e., all truth values greater than the multiplicative truth  $I$  are designated. If we take the pushout of *MILL* and *Bicartesian* over *CatLog*, we obtain intuitionistic linear logic *ILL* where the bicartesian structure corresponds to the additive connectives. Note that in *ILL*, we have a morphism from  $I$  to  $\top$ , i.e.,  $\top$  is a greater designated truth value than  $I$ , but not (except for trivial models) the other way round.

All preorder-enriched institutions induced in the above examples are strongly sound by Prop. 4.4. Prop. 4.7 permits the introduction of falsity and disjunction, which yields the deduction theorem for *IProp* and *Prop*. Thus *IProp* and *Prop* are strongly complete. The deduction theorem does not hold for modal logic: For every proof term  $x : Mor(\top, A)$  there is a proof term  $\square x : Mor(\square\top, \square A)$ , which, together with the canonical morphism  $m_{\top} : Mor(\top, \square\top)$ , yields a morphism  $m_{\top}; \square x : Mor(\top, \square A)$ , but there is in general no closed proof term of the sort  $Mor(A, \square A)$ . A similar argument disproves the deduction theorem for linear logic. Indeed, these preorder-enriched institutions are only weakly complete.

For modal and linear logic, it is not trivial to define the rewriting structure in a way that permits normalization. Therefore, we give most properties for these as equalities instead of as rewrites.

## 6 Equivalence Between $\lambda$ and $Cat$

$\mathbf{CPL}^{ND\lambda}$  models are valuations into the boolean algebra  $\{0, 1\}$  whereas  $\mathcal{I}(Prop)$ -models are valuations into arbitrary boolean algebras. This difference is not trivial. In a boolean algebra, regarded as a  $Prop$ -category, an object has a global element iff it is terminal (i.e., equal to the 1 of the boolean algebra). There is no mapping from valuations into arbitrary boolean algebras to valuations into  $Bool$  that preserves and reflects truth (i.e. terminalhood) of formulas: Let  $\nu: \{p, q\} \rightarrow Bool^2$  map  $p$  to  $(\top, \perp)$  and  $q$  to  $(\perp, \top)$ . Assume that  $\nu': \{p, q\} \rightarrow Bool$  makes true the same formulas as  $\nu$ . Then  $\nu'(p)$  and  $\nu'(q)$  must both be  $\perp$ , because neither  $\nu(p)$  nor  $\nu(q)$  is terminal. However,  $\nu(p \vee q)$  is terminal, while  $\nu'(p \vee q)$  is not, a contradiction. Hence, the model translation of the comorphism from  $\mathcal{I}(\mathcal{L})$  to  $\overline{\mathcal{I}(\mathcal{L})}$  introduced in Prop. 4.16 cannot be reversed.

A related observation is that  $\mathbf{CPL}^{ND\lambda}$  and  $\mathcal{I}(Prop)$  differ in their behavior of disjunction. While in  $\mathbf{CPL}^{ND\lambda}$ , a model  $M$  satisfies a disjunction iff it satisfies either of the disjuncts (which is called ‘‘model-theoretic disjunction’’ in [30]), this is not the case for  $\mathcal{I}(Prop)$ : Consider the weakly free model over the theory  $(\{A, B\}, \{A + B\})$ , existing by Prop. 4.14.

However, from the point of view of semantic consequence, we can restrict ourselves to valuations into  $Bool$ : Semantic consequence in  $\mathbf{CPL}^{ND\lambda}$  and  $\mathcal{I}(Prop)$  coincide. We turn this observation into a general notion:

**Definition 6.1.** Let  $\mathcal{L}$  be a categorical logic, and let  $A \in |\overline{\mathbb{C}(\mathcal{L})}|$ . Let  $\mathcal{I}^A(\mathcal{L})$  denote the institution obtained from  $\mathcal{I}(\mathcal{L})$  by allowing only those  $\Sigma$ -models  $(M, m)$  for which  $M = A$ . We denote satisfaction and semantic consequence in this institution with the superscript  $A$ . The category  $A$  is called a *weak truth value object* for  $\mathcal{L}$  if  $\varphi \models^A \psi$  implies  $\varphi \models \psi$  for all sentences  $\varphi, \psi$ , and a *strong truth value object* if  $\Phi \models^A \psi$  implies  $\Phi \models \psi$  for all sets  $\Phi$  of sentences and sentences  $\psi$ .

(N.B.: Even in the presence of conjunction, a weak truth value object need not be a strong truth value object, as the set  $\Phi$  in question may be infinite.)

**Proposition 6.2.** Let  $\mathcal{L}$  be a categorical logic extending the theory of cartesian closed categories (i.e., minimal intuitionistic logic with  $\top, *, \rightarrow$ ), and let  $A \in |\overline{\mathbb{C}(\mathcal{L})}|$  be thin and skeletal, hence essentially being a partial ordering  $\leq$ . Then  $A$  is a strong truth value object for  $\mathcal{L}$  iff the following condition holds.

- (\*) Let  $B \in |\overline{\mathbb{C}(\mathcal{L})}|$ , and let  $a, b \in |B|$ . If  $f(a) \leq f(b)$  for each  $\mathcal{L}$ -morphism  $f: B \rightarrow A$ , then  $\text{hom}_B(a, b) \neq \emptyset$ .

(If  $B$  is a thin category, i.e. a preorder, then condition (\*) states that the source of all morphisms from  $B$  into  $A$  is jointly order-reflecting.)

*Proof.* Assume that (\*) holds, let  $\Phi \models^A \psi$ , and let  $(B, m) \models \Phi$ . Then for every  $f: B \rightarrow A$ ,  $(A, f \circ m) \models \Phi$  and hence  $(A, f \circ m) \models \psi$ , i.e.,  $\top \leq f(m(\psi))$ . By (\*), it follows that  $B(\top, m(\psi)) \neq \emptyset$ , i.e.,  $(B, m) \models \psi$ .

Conversely, let  $A$  be a strong truth value object, let  $B \in |\overline{\mathbb{C}(\mathcal{L})}|$ , and let  $a, b \in |B|$  such that  $A(f(a), f(b)) \neq \emptyset$  for all  $f: B \rightarrow A$ . Let  $\Sigma = |B|$ , and let  $\Phi$  be the theory (i.e., the set of valid formulas) of the  $\Sigma$ -model  $(B, \eta)$  where  $\eta: \Sigma \rightarrow |B|$  is the inclusion. Since morphisms  $f: B \rightarrow A$  are then just the  $\Sigma$ -valuations in  $A$  that validate  $\Phi$  (because by the assumption on  $\mathcal{L}$ , formulas  $\chi_1$  and  $\chi_2$  denote isomorphic objects of  $B$  iff  $(B, \eta) \models \chi_1 \leftrightarrow \chi_2$ ), the premise

|                                 |                   |
|---------------------------------|-------------------|
| void logic                      | <i>CatLog</i>     |
| conjunctive logic               | <i>Cartesian</i>  |
| conjunctive-implicational logic | <i>CartClosed</i> |
| intuitionistic logic            | <i>IProp</i>      |
| classical logic                 | <i>Prop</i>       |

Figure 5: Five stages of the correspondence.

says that  $\Phi \models^A a \rightarrow b$  (note that the assumption on  $\mathcal{L}$  implies that an implication holds iff there exists a morphism between the corresponding objects). Thus,  $\Phi \models a \rightarrow b$ , so that  $a \rightarrow b$  holds in  $(B, \eta)$ ; i.e.,  $B(a, b) \neq \emptyset$  as claimed.  $\square$

**Example 6.3.** *Bool*, seen as a *Prop*-category, is a strong truth value object for *Prop*. For any dense-in-itself metric space  $X$ , the Heyting algebra  $\mathfrak{O}(X)$  of open sets in  $X$  (qua bicartesian closed category) is a weak truth value object for *IProp* [34].

One can use the above result to show that no strong truth value object for *IProp* exists. To see this, assume that  $A$  is a Heyting algebra satisfying condition  $(*)$  of Proposition 6.2. Let  $\alpha > |A|$  be a cardinal, and let  $B$  be the ordinal  $\alpha + 2$ , considered as a Heyting algebra. The two largest elements of  $\alpha + 2$  are  $\alpha$  and  $\alpha + 1$ . We show that for every morphism  $f : B \rightarrow A$ ,  $f(\alpha) = f(\alpha + 1)$ ; then by  $(*)$ ,  $\alpha + 1 \leq \alpha$ , contradiction. For cardinality reasons, we have  $a < b$  in  $B$  such that  $f(a) = f(b)$ . Then

$$f(a) = f(b \rightarrow a) = (f(b) \rightarrow f(a)) = \top = f(\alpha + 1),$$

and since  $a \leq \alpha$ , we obtain  $f(\alpha) = f(\alpha + 1)$  as claimed.

A consequence of this observation is that, for every dense-in-itself metric space  $X$ , the consequence relation on  $\mathfrak{O}(X)$  is non-compact, since otherwise, the weak truth value object  $\mathfrak{O}(X)$  would also be a strong truth value object.

We are now ready to formalize the Curry-Howard-Tait correspondence in terms of preorder-enriched institutions. We cannot expect this to be constructed in an institution-independent manner. Rather, for some given categorical propositional logic  $\mathcal{L}$ , we can relate  $\mathcal{I}(\mathcal{L})$  to a well-known institution. We follow on the five stages in Fig. 2 and map the natural deduction logics to categorical logics as depicted in Fig. 5.<sup>4</sup>

Fig. 6 describes the mapping from categorical logic to natural deduction. A morphism  $M : Mor(A, B)$  in categorical logic is mapped to a natural deduction proof term  $\alpha_x(M)$  of form  $x : A \triangleright \alpha_x(M) : B$ . Note that  $\alpha_x(M)$  may contain the free variable  $x : A$ . For the top-level translation, we may chose  $x$  in an arbitrary way.

Fig. 7 describes the converse mapping from natural deduction to categorical logic. Since the rule system in Fig. 2 deals with multi-variable contexts, we translate a proof  $x_1 : A_1, \dots, x_n : A_n \triangleright M : B$  to a morphism  $\alpha'(M) : Mor(A_1 \times \dots \times A_n, B)$ . This corresponds to a switch from multi-variable to one-variable contexts using conjunction. Ultimately, we need this translation only for one-variable contexts on both sides; however, the recursive definition at some places involves enlargement of contexts.

<sup>4</sup>See [33] for the full specifications of the categorical logics.

| $M$   | $x \triangleright \alpha_x(M)$   |
|---|--|
| $id : Mor(A, A)$                                      | $x : A \triangleright x : A$   |
| $(M : Mor(A, B)); (N : Mor(B, C))$                    | $x : A \triangleright \alpha_y(N)[y := \alpha_x(M)]$   |
| $! : Mor(A, \top)$                                    | $x : A \triangleright \Delta : \top$   |
| $\langle M, N \rangle : Mor(A, B \times C)$           | $x : A \triangleright \langle \alpha_x(M), \alpha_x(N) \rangle : B \wedge C$   |
| $\pi_1 : Mor(A \times B, A)$                          | $x : A \wedge B \triangleright \mathsf{fst}(x) : A$  |
| $\pi_2 : Mor(A \times B, B)$                          | $x : A \wedge B \triangleright \mathsf{snd}(x) : B$  |
| $\mathsf{eval} : Mor(B^A \times A, B)$                | $x : (A \rightarrow B) \wedge A \triangleright \mathsf{fst}(x) \mathsf{snd}(x) : B$  |
| $\mathsf{curry}(M) : Mor(A, C^B)$                     | $x : A \triangleright \lambda y : B . \alpha_z(M)[z := \langle x, y \rangle]$  |
| $!! : Mor(\perp, A)$                                  | $x : \perp \triangleright \nabla_A(x) : A$   |
| $\mathsf{inl} : Mor(A, A + B)$                        | $x : A \triangleright \mathsf{inl}(x) : A \vee B$  |
| $\mathsf{inr} : Mor(B, A + B)$                        | $x : B \triangleright \mathsf{inr}(x) : A \vee B$  |
| $[M, N] : Mor(A + B, C)$                              | $x : A \vee B \triangleright \mathbf{case} \ x \ \mathbf{of} \ \mathsf{inl}(y) \rightarrow \alpha_y(M) \mid \mathsf{inr}(z) \rightarrow \alpha_z(N)$ |
| $\mathsf{tnd} : Mor(\top, A + (A \rightarrow \perp))$ | $x : \top \triangleright \mathsf{tnd}_A : A \vee \neg A$   |

Figure 6: Mapping categorical logic to natural deduction, following the five stages in Fig. 5.

| $M$  | $\alpha'(M)$   |
|--|--|
| $\Gamma, x : A \triangleright x : A$   | $\pi_i : Mor(\prod \Gamma \times A, A)$ resp. $id : Mor(A, A)$   |
| $\Gamma \triangleright \Delta : \top$  | $! : Mor(\prod \Gamma, \top)$  |
| $\Gamma \triangleright \langle M, N \rangle : A \wedge B$  | $\langle \alpha'(M), \alpha'(N) \rangle : Mor(\prod \Gamma, A \times B)$   |
| $\Gamma \triangleright \mathsf{fst}(M) : A$  | $\alpha'(M); \pi_1 : Mor(\prod \Gamma, A)$   |
| $\Gamma \triangleright \mathsf{snd}(M) : B$  | $\alpha'(M); \pi_2 : Mor(\prod \Gamma, B)$   |
| $\Gamma \triangleright \lambda x : A. M : A \rightarrow B$   | $\mathsf{curry}(\alpha'(M)) : Mor(\prod \Gamma \times A, B)$   |
| $\Gamma \triangleright MN : B$   | $\langle \alpha'(M), \alpha'(N) \rangle; \mathsf{eval} : Mor(\prod \Gamma, B)$   |
| $\Gamma \triangleright \nabla_A(M) : A$  | $\alpha'(M); !! : Mor(\prod \Gamma, A)$  |
| $\Gamma \triangleright \mathsf{inl}(M) : A \vee B$   | $\alpha'(M); \mathsf{inl} : Mor(\prod \Gamma, A + B)$  |
| $\Gamma \triangleright \mathsf{inr}(M) : A \vee B$   | $\alpha'(M); \mathsf{inr} : Mor(\prod \Gamma, A + B)$  |
| $\Gamma \triangleright \mathbf{case} \ M \ \mathbf{of} \ \mathsf{inl}(x) \rightarrow N \mid \mathsf{inr}(y) \rightarrow P : C$ | $\langle \alpha'(M), id \rangle; [\mathsf{curry}(\alpha'(N)), \mathsf{curry}(\alpha'(P))]; \mathsf{eval} : Mor(\prod \Gamma, C)$ |
| $\Gamma \triangleright \mathsf{tnd}_A : A \vee \neg A$   | $!; \mathsf{tnd} : Mor(\prod \Gamma, A + (A \rightarrow \perp))$   |

Figure 7: Mapping natural deduction to categorical logic, following the five stages in Fig. 5.  $\pi_i$  is the  $i$ -th projection, obtained as a composite of  $\pi_1$  and  $\pi_2$ .

$$\begin{array}{ll}
\langle \mathsf{fst}(M), \mathsf{snd}(M) \rangle & \rightsquigarrow_{\eta} M \\
\lambda x. M x & \rightsquigarrow_{\eta} M \\
x : \perp \triangleright M : A & \rightsquigarrow_{\eta} \nabla_A(x) \\
\mathbf{case} \ x \ \mathbf{of} \ \mathsf{inl}(y) \rightarrow N[x := \mathsf{inl}(y)] \mid \mathsf{inr}(y) \rightarrow N[x := \mathsf{inr}(y)] & \rightsquigarrow_{\eta} N
\end{array}$$

Figure 8:  $\eta$ -reduction rules for proof terms.

**Proposition 6.4.** Assume that the  $\eta$ -rules of Fig. 8 are added to the reduction of  $\mathbf{CPL}^{ND\lambda}$ . Moreover, let  $\overline{\mathbf{CPL}}^{ND\lambda}$  be  $\mathbf{CPL}^{ND\lambda}$  with proof reductions quotiented out. Then, for any signature  $\Sigma \in \mathbb{S}\text{et}$ ,

1. the mapping given in Fig. 6 is a preorder-preserving functor  $\alpha_\Sigma: \text{Sen}^{\mathcal{I}(Prop)}(\Sigma) \rightarrow \text{Sen}^{\overline{\mathbf{CPL}}^{ND\lambda}}(\Sigma)$ , and similarly for the other stages in Fig. 5.
2. the mapping given in Fig. 7 is a functor  $\alpha'_\Sigma: \text{Sen}^{\mathbf{CPL}^{ND\lambda}}(\Sigma) \rightarrow \text{Sen}^{\overline{\mathcal{I}(Prop)}}(\Sigma)$ , and similarly for the other stages in Fig. 5.
3.  $\alpha_\Sigma$  naturally extends to quotienting of proofs with respect to the reduction relation, and then becomes the inverse of  $\alpha'_\Sigma$  (hence, both are isomorphisms).

*Proof.* 1. Preservation of identity and composition follows from the first two lines in the table of Fig. 6. Preservation of proof reduction is proved by considering all the reduction axioms. For example, consider the reduction axiom  $\langle f; \pi_1, f; \pi_2 \rangle \rightsquigarrow f$ . We have  $\alpha(\langle f; \pi_1, f; \pi_2 \rangle) = \langle \text{fst}(\alpha(f)), \text{snd}(\alpha(f)) \rangle \rightsquigarrow_\eta \alpha(f)$ .

2. Preservation of identities is clear. Preservation of composition follows from the fact that

$$(\alpha'(x : A \triangleright M : B) \times id); \alpha'(y : B, \Gamma \triangleright N : C) \sim \alpha'(x : A, \Gamma \triangleright N[y := M]),$$

where  $\sim$  is the equivalence relation generated by reducibility. This fact is proved by induction over  $N$ . For example, assume that by induction hypothesis  $\alpha'(y : B, z : D, \Gamma \triangleright N[y := M]) \sim (\alpha'(M) \times id); \alpha'(N)$ . Assuming suitable  $\alpha$ -renaming, then also

$$\begin{aligned} \alpha'(y : B, \Gamma \triangleright (\lambda z : D.N)[y := M]) &= \\ \alpha'(y : B, \Gamma \triangleright \lambda z : D.(N[y := M])) &= \\ \text{curry}(\alpha'(N[y := M])) &\sim \\ \text{curry}((\alpha'(M) \times id); \alpha'(N)) &\rightsquigarrow \\ \alpha'(M); \text{curry}(\alpha'(N)) & \end{aligned}$$

We should remark that most of the cases even go through with  $\rightsquigarrow$  instead of  $\sim$ , which would show  $\alpha'_\Sigma$  to be a lax functor. The only problem is the treatment of variables. Here we have

$$\begin{aligned} (\alpha'(x : A \triangleright M : B) \times id); \pi_1 &= \\ \langle \pi_1; \alpha'(x : A \triangleright M : B), \pi_2 \rangle; \pi_1 &\rightsquigarrow \\ \pi_1; \alpha'(x : A \triangleright M : B) &= \\ \alpha'(x : A, \Gamma \triangleright M : B) & \end{aligned}$$

but for obtaining a lax functor, the opposite rewrite would be needed. This point needs further investigation.

3. Again by induction, one can show that  $\alpha_\Sigma(\alpha'_\Sigma(M))$  is equivalent to  $M$ , and  $\alpha'_\Sigma(\alpha_\Sigma(N))$  is equivalent to  $N$ . □

**Theorem 6.5.** There is an isomorphism  $\mathcal{I}^{\text{Bool}}(\overline{\text{Prop}})$  to  $\overline{\mathbf{CPL}}^{ND\lambda}$ .

*Proof.* The signature translations are the identity functors; the model translations are the obvious bijections; and the sentence translation is given by Prop. 6.4. □

Similarly, there is an isomorphism between  $\overline{\mathcal{I}(IProp)}$  and intuitionistic logic with Heyting algebra semantics. For modal and to a lesser extent for linear logic, the Curry-Howard-Tait correspondence is not so much a proved result, but rather a design paradigm that is satisfied a priori. In these cases, it makes more sense to see the institution constructed within our framework as the incarnation of the correspondence.

## 7 Conclusion and Future Work

We have presented a canonical way of obtaining proof-theoretic institutions for categorical propositional logics, following the spirit of the Curry-Howard-Tait isomorphism. We have proved generic deduction, soundness and completeness theorems, and given examples of categorical logics, for which categorical treatment had already been established in a non-institutional framework. Our definitions have the crucial advantage that they offer the possibility of parametric and modular specifications, and that they have been formalized (and machine-checked) as Twelf specifications. We avoid the use of more complicated frameworks like polycategories by relying on conjunction for the treatment of multi-assumption contexts.

For classical logic, the institutional structure sheds light on the usual collapsing of proofs problem in classical logic (classical bicartesian closed categories are boolean algebras), which we avoided by using preorder-enriched categories, as in [16]. For Linear Logic, the specification of the modality  $!$  is considerably involved; to simplify this specification, we might use a different categorical model (see [29]). But this would be too much of a departure from the point of view we have taken in this paper of using established notions of categorical models, not involving fibrations or indexed categories, for the time being. Another interesting direction left to future work is to consider linear logic with both a classical and a linear function space as in [28].

The Curry-Howard-Tait isomorphism can be recovered as an explicit isomorphism between institutions, one institution using proof trees, the other one  $\lambda$ -terms. Concerning the relation between various  $\lambda$ -calculi and categorical logic, only a weaker correspondence could be set up. One obstacle is the difference between, e.g., boolean algebra-valued and *Bool*-valued models. We have provided some general results about the relations between such models. Another obstacle are the different notions of proofs and proof reductions that make it hard to obtain isomorphic reduction-preserving translations. In order to obtain even the weaker correspondences, the usual  $\beta$ -rules for proof terms have to be extended by  $\eta$ -rules. But even then, functoriality of proof translation only holds for one direction; for the other direction, proof reductions have to be quotiented out. Without the quotienting, we were almost able to obtain a lax functor, but there are problems caused by the necessary switching between variable contexts. This point needs further investigation.

The study of further properties, such as Craig interpolation and Beth definability, is the subject of future work, as is the extension of our framework to first-order logic. The latter will require a more powerful meta-language, namely one that permits to declare function symbols that take functions as arguments. Another interesting question is whether the institutions  $\mathcal{I}(\mathcal{L})$  have elementary diagrams in the sense of [10]. Our current notion of model is obviously too weak to ensure this; for ensuring elementary diagrams one would need "intensional models" over signatures containing proof variables, to be valued with proofs — such models would not only determine which propositions are true, but also why. With such models, it should also be easy to show liberality of  $\mathcal{I}(\mathcal{L})$ .

## References

- [1] J. Adámek, H. Herrlich, and G. E. Strecker. *Abstract and Concrete Categories*. Wiley Interscience, 1990.
- [2] A. Avron. The Semantics and Proof Theory of Linear Logic. Technical Report ECS-LFCS-87-27, University Edinburgh, 1987.
- [3] G. Bellin, M. Hyland, E. Robinson, and C. Urban. Categorical proof theory of classical propositional calculus. *Theor. Comput. Sci.*, 364(2):146–165, 2006.
- [4] N. Benton, G. Bierman, V. de Paiva, and M. Hyland. Term Assignment for Intuitionistic Linear Logic. Technical Report 262, University of Cambridge Computer Laboratory, 1992.
- [5] M. Bidoit and R. Hennicker. On the integration of observability and reachability concepts. In *Fossacs 2002*, vol. 2303 of *LNCS*, pp. 21–36. Springer, 2002.
- [6] G. Bierman and V. de Paiva. On an intuitionistic modal logic. *Studia Logica*, 65:383–416, 2000.
- [7] T. Borzyszkowski. Moving specification structures between logical systems. In *WADT 98*, vol. 1589 of *LNCS*, pp. 16–30. Springer, 1999.
- [8] C. Cîrstea. Institutionalising many-sorted coalgebraic modal logic. In *Coalgebraic Methods in Computer Science*, vol. 65 of *ENTCS*. Elsevier, 2002.
- [9] R. Diaconescu. Institution-independent ultraproducts. *Fundamenta Informaticae*, 55:321–348, 2003.
- [10] R. Diaconescu. Elementary diagrams in institutions. *J. Logic Computation*, 14(5):651–674, 2004.
- [11] R. Diaconescu. Herbrand theorems in arbitrary institutions. *Information Processing Letters*, 90:29–37, 2004.
- [12] R. Diaconescu. An institution-independent proof of Craig Interpolation Theorem. *Studia Logica*, 77(1):59–79, 2004.
- [13] R. Diaconescu. *Institution-independent Model Theory*. Birkhäuser Basel, 2008. To appear.
- [14] R. Diaconescu, J. Goguen, and P. Stefaneas. Logical support for modularisation. In *Logical Environments*, pp. 83–130. Cambridge, 1993.
- [15] K. Došen. *Cut Elimination in Categories*. Kluwer Academic Publishers, Dordrecht, 1999.
- [16] C. Führmann and D. Pym. On the geometry of interaction for classical logic. In *LICS 2004*, pp. 211–220. IEEE Computer Society Press, 2004. Extended version at <http://www.cs.bath.ac.uk/~cf/dj.pdf>.

- [17] J. L. Fiadeiro and J. F. Costa. Mirror, mirror in my hand: A duality between specifications and models of process behaviour. *Math. Struct. Comput. Sci.*, 6(4):353–373, 1996.
- [18] J. H. Gallier. Constructive logics part I: A tutorial on proof systems and typed gamma-calculi. *Theor. Comput. Sci.*, 110(2):249–339, 1993.
- [19] J. A. Goguen and R. M. Burstall. A study in the foundations of programming methodology: Specifications, institutions, charters and parchments. In D. P. et al., editor, *Category Theory and Computer Programming*, vol. 240 of *Lecture Notes in Computer Science*, pp. 313–333. Springer Verlag, 1985.
- [20] J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM*, 39:95–146, 1992.
- [21] J. A. Goguen and R. Diaconescu. Towards an algebraic semantics for the object paradigm. In *WADT 94*, vol. 785 of *LNCS*. Springer, 1994.
- [22] J. A. Goguen and G. Roṣu. Institution morphisms. *Formal Aspects of Computing*, 13:274–307, 2002.
- [23] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
- [24] R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.
- [25] W. A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pp. 479–490. Academic Press, 1980.
- [26] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge, 1986.
- [27] S. Mac Lane. *Categories for the Working Mathematician*. Springer, 1997.
- [28] M. Maietti, V. de Paiva, and E. Ritter. Categorical Models for Intuitionistic and Linear Type Theory. In J. Tiuryn, editor, *Foundations of Software Science and Computation Structures, Third International Conference*, pp. 223–237. Springer, 2000.
- [29] M. E. Maietti, P. Maneggia, V. de Paiva, and E. Ritter. Relating categorical semantics for intuitionistic linear logic. *Appl. Categ. Structures*, 13(1):1–36, 2005.
- [30] T. Mossakowski, J. Goguen, R. Diaconescu, and A. Tarlecki. What is a logic? In J.-Y. Beziau, editor, *Logica Universalis*, pp. 113–133. Birkhäuser, 2005.
- [31] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.
- [32] F. Rabe. First-Order Logic with Dependent Types. In N. Shankar and U. Furbach, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, vol. 4130 of *Lecture Notes in Computer Science*, pp. 377–391. Springer, 2006.

- [33] F. Rabe, V. de Paiva, and T. Mossakowski. Categorical Logic Specifications, 2007. See <http://kwarc.eecs.iu-bremen.de/frabe/Research/cht/index.html>.
- [34] H. Rasiowa and R. Sikorski. *The Mathematics of Metamathematics*. PWN, Warsaw, 1963.
- [35] D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Inform. and Comput.*, 76:165–210, 1988.
- [36] L. Schröder, T. Mossakowski, and C. Lüth. Type class polymorphism in an institutional framework. In J. Fiadeiro, editor, *WADT 04*, vol. 3423 of *LNCS*, pp. 234–248. Springer, 2004.
- [37] R. A. G. Seely. Hyperdoctrines, natural deduction and the Beck condition. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 29(6):505–542, 1983.
- [38] R. A. G. Seely. Locally cartesian closed categories and type theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 95(1):33–48, 1984.
- [39] R. A. G. Seely. Linear logic,  $*$ -autonomous categories, and cofree coalgebras. In *Categories in Computer Science and Logic*, Contemp. Math., pp. 371–382. AMS, 1989.
- [40] A. Sernadas, C. Sernadas, C. Caleiro, and T. Mossakowski. Categorical fibring of logics with terms and binding operators. In D. Gabbay and M. d. Rijke, editors, *Frontiers of Combining Systems 2*, Studies in Logic and Computation, pp. 295–316. Research Studies Press, 2000.
- [41] M. E. Szabo. Polycategories. *Communications in Algebra*, 3(8):663–689, 1975.
- [42] A. Tarlecki. On the existence of free models in abstract algebraic institutions. *Theoret. Comput. Sci.*, 37(3):269–304, 1985.

# THF0 – The core of the TPTP Language for Higher-Order Logic

Christoph Benzmüller<sup>1\*</sup>, Florian Rabe<sup>2</sup>, and Geoff Sutcliffe<sup>3</sup>

<sup>1</sup> Saarland University, Germany

<sup>2</sup> Jacobs University Bremen, Germany

<sup>3</sup> University of Miami, USA

**Abstract.** One of the keys to the success of the Thousands of Problems for Theorem Provers (TPTP) problem library and related infrastructure is the consistent use of the TPTP language. This paper introduces the core of the TPTP language for higher-order logic – THF0, based on Church’s simple type theory. THF0 is a syntactically conservative extension of the untyped first-order TPTP language.

## 1 Introduction

There is a well established infrastructure that supports research, development, and deployment of first-order Automated Theorem Proving (ATP) systems, stemming from the Thousands of Problems for Theorem Provers (TPTP) problem library [29]. This infrastructure includes the problem library itself, the TPTP language [27], the Szs ontologies [30], the Thousands of Solutions from Theorem Provers (TSTP) solution library, various tools associated with the libraries [26], and the CADE ATP System Competition (CASC) [28]. This infrastructure has been central to the progress that has been made in the development of high performance first-order ATP systems.

One of the keys to the success of the TPTP and related infrastructure is the consistent use of the TPTP language. Until now the TPTP language has been defined for only untyped first-order logic (first-order form (FOF) and clause normal form (CNF)). This paper introduces the core of the TPTP language for classical higher-order logic – THF0, based on Church’s simple type theory. THF0 is the *core* language in that it provides only the commonly used and accepted aspects of a higher-order logic language. The full THF language includes the THFF extension that allows the use of first-order style prefix syntax, and the THF1 and THF2 extensions that provide successively richer constructs in higher-order logic.<sup>4</sup>

---

\* This work was supported by EPSRC grant EP/D070511/1 (LEO-II)

<sup>4</sup> The THFF, THF1, and THF2 extensions have already been completed. The initial release of only THF0 allows users to adopt the language without being swamped by the richness of the full THF language. The full THF language definition is available from the TPTP web site, [www.tptp.org](http://www.tptp.org).

As with the first-order TPTP language, a common adoption of the THF language will enable convenient communication of higher-order data between different systems and researchers. THF0 is the starting point for building higher-order analogs of the existing first-order infrastructure components, including a higher-order extension to the TPTP problem library, evaluation of higher-order ATP systems in CASC, TPTP tools to process THF0 problems, and a higher-order TSTP proof representation format. An interesting capability that has become possible with the THF language is easy comparison of higher-order and first-order versions of problems, and evaluation of the relative benefits of the different encodings with respect to ATP systems for the logics.

## 2 Preliminaries

### 2.1 The TPTP Language

The TPTP language is a human-readable, easily machine-parsable, flexible and extensible language suitable for writing both ATP problems and solutions. The BNF of the THF0 language, which defines common TPTP language constructs and the THF0 specific constructs, is given in Appendix A.

The top level building blocks of the TPTP language are *annotated formulae*, *include directives*, and *comments*. An annotated formula has the form:

*language*(*name*, *role*, *formula*, [*source*, [*useful\_info*]]) .

An example annotated first-order formula, supplied from a file, is:

```
fof(formula_27, axiom,
! [X,Y] :
  ( subclass(X,Y)
  <=> ! [U] :
    ( member(U,X)
    => member(U,Y) )),
file('SET005+0.ax',subclass_defn),
[description('Definition of subclass'), relevance(0.9)]).
```

The *languages* supported are first-order form (*fof*), clause normal form (*cnf*), and now typed higher-order form (*thf*). The *role*, e.g., *axiom*, *lemma*, *conjecture*, defines the use of the formula in an ATP system - see the BNF for the list of recognized roles. The details of the *formula* notation for connectives and quantifiers can be seen in the BNF. The forms of identifiers for uninterpreted functions, predicates, and variables follow Prolog conventions, i.e., functions and predicates start with a lowercase letter, variables start with an uppercase letter, and all contain only alphanumeric characters and underscore. The TPTP language also supports interpreted symbols, which either start with a \$, or are composed of non-alphanumeric characters. The basic logical connectives are !, ?, ~, |, &, =>, <=>, and <~>, for  $\forall$ ,  $\exists$ ,  $\neg$ ,  $\vee$ ,  $\wedge$ ,  $\Rightarrow$ ,  $\Leftarrow$ ,  $\Leftrightarrow$ , and  $\oplus$  respectively. Quantified variables follow the quantifier in square brackets, with a colon to separate the quantification from the logical formula. The *source* is an optional term

describing where the formula came from, e.g., an input file or an inference. The *useful\_info* is an optional list of terms from user applications.

An `include` directive may include an entire file, or may specify the names of the annotated formulae that are to be included from the file. Comments in the TPTP language extend from a % character to the end of the line, or may be block comments within /\* ... \*/ bracketing.

## 2.2 Higher-order Logic

There are many quite different frameworks that fall under the general label “higher-order”. The notion reaches back to Frege’s original predicate calculus [11]. Inconsistencies in Frege’s system, caused by the circularity of constructions such as “the set of all sets that do not contain themselves”, made it clear that the expressivity of the language had to be restricted in some way. One line of development, which became the traditional route for mathematical logic, and which is not addressed further here, is the development of axiomatic first-order set theories, e.g. Zermelo-Fraenkel set theory [32].

Russell suggested using type hierarchies, and worked out ramified type theory. Church (inspired by work of Carnap) later introduced simple type theory [9], a higher-order framework built on his simply typed  $\lambda$  calculus, employing types to reduce expressivity and to remedy paradoxes and inconsistencies. Church’s simple type theory was later extended and refined in various ways, e.g., by Martin Löf who added type universes, and by Girard and Reynolds who introduced type systems with polymorphism. This stimulated much further research in intuitionistic and constructive type theory.

Variants and extensions of Church’s simple type theory have been the logic of choice for interactive proof assistants such as HOL4 [13], HOL Light [15], PVS [22], Isabelle/HOL [21], and OMEGA [25]. Church’s simple type theory is also a common basis for higher-order ATP systems. The TPS system [1], which is based on a higher-order mating calculus, is a pioneering ATP system for Church’s simple type theory. More recently developed higher-order ATP systems, based on extensional higher-order resolution, are LEO [5] and LEO-II [7]. Otter- $\lambda$  [2] is an extension of the first-order system Otter to an untyped variant of Church’s type theory. This common use of Church’s simple type theory motivates using it as the starting point for THF0. For the remainder of this paper “higher-order” is therefore synonymous with Church’s simple type theory [4].

A major application area of higher-order logic is hardware and software verification. Several interactive higher-order proof assistants put an emphasis on this. For example, the Isabelle/HOL system has been applied in the Verisoft project [12]. The formal proofs to be delivered in such a project require a large number of user interactions – a resource intensive task to be carried out by highly trained specialists. Often only a few of the interaction steps really require human ingenuity, and many of them could be avoided through better automation support. There are several barriers that hamper the application of automated higher-order ATP systems in this sense: (i) the available higher-order ATP systems are not yet optimized for this task; (ii) typically there are large syntax gaps between the

higher-order representation languages of the proof assistants, and the input languages of the higher-order ATP systems; (iii) the results of the higher-order ATP systems have to be correctly interpreted and (iv) their proof objects might need to be translated back. The development of a commonly accepted infrastructure, and increased automation in higher-order ATP, will benefit these applications and reduce user interaction costs. Another promising application area is knowledge based reasoning. Knowledge based projects such as Cyc [19] and SUMO [20] contain a significant fraction of higher-order constructs. Reasoning in and about these knowledge sources will benefit from improved higher-order ATP systems. A strong argument for adding higher-order ATP to this application area is its demand for natural and human consumable problem and solution representations, which are harder to achieve after translating higher-order content into less expressible frameworks such as first-order logic. Further application areas of higher-order logic include computer-supported mathematics [24], and reasoning within and about multimodal logics [6].

### 2.3 Church's Simple Type Theory

Church's notion of higher-order logic is based on the simply typed  $\lambda$ -calculus. The set of simple types is freely generated from basic types  $i$  (written  $\$i$  in THF0) and  $o$  ( $\$o$  in THF0), and possibly further base types using the function type constructor  $\rightarrow$  ( $>$  in THF0).

Simply typed higher-order terms are built up from simply typed variables ( $X_\alpha$ ), simply typed constants ( $c_\alpha$ ),  $\lambda$ -abstraction, and application. It is assumed that sufficiently many special logical constants are available, so that all other logical connectives can be defined. For example, it is assumed that  $\neg_{o \rightarrow o}$ ,  $\vee_{o \rightarrow o \rightarrow o}$ , and  $\Pi_{(\alpha \rightarrow o) \rightarrow o}$  (for all simple types  $\alpha$ ) are given. The semantics of these logical symbols is fixed according to their intuitive meaning.

Well-formed (simply typed) terms are defined simultaneously for all simple types  $\alpha$ . Variables and constants of type  $\alpha$  are well-formed terms of type  $\alpha$ . Given a variable  $X$  of type  $\alpha$  and a term  $T$  of type  $\beta$ , the abstraction term  $\lambda X.T$  is well-formed and of type  $\alpha \rightarrow \beta$ . Given terms  $S$  and  $T$  of types  $\alpha \rightarrow \beta$  and  $\alpha$  respectively, the application term  $(S\ T)$  is a well-formed and of type  $\beta$ .

The initial target semantics for THF0 is Henkin semantics [4, 16]. However, there is no intention to fix the semantics within THF0. THF0 is designed to express problems syntactically, with the semantics being specified separately [3].

## 3 The THF0 Language

THF0 is a syntactically conservative extension of the untyped first-order TPTP language, adding the syntax for higher-order logic. Maintaining a consistent style between the first-order and higher-order languages facilitates easy adoption of the new language, through reuse or adaptation of existing infrastructure for processing TPTP format data, e.g., parsing tools, pretty-printing tools, system testing, and result analysis, (see Section 5). A particular feature of the TPTP

language, which has been maintained in THF0, is Prolog compatibility. This allows an annotated formula to be read with a single Prolog `read/1` call, in the context of appropriate operator definitions. There are good reasons for maintaining Prolog compatibility [27].

Figure 1 presents an example problem encoded in THF0. The example is from the domain of basic set theory, stating the distributivity of union and intersection. It is a higher-order version of the first-order TPTP problem `SET171+3`, which employs first-order set theory to achieve a first-order encoding suitable for first-order theorem provers. The encoding in THF0 exploits the fact that higher-order logic provides a naturally built-in set theory, based on the idea of identifying sets with their characteristic functions. The higher-order encoding can be solved more efficiently than the first-order encoding [8].

The first three annotated formulae of the example are *type declarations* that declare the type signatures of  $\in$ ,  $\cap$ , and  $\cup$ . The `type` role is new, added to the TPTP language for THF0.<sup>5</sup> A simple type declaration has the form `constant_symbol:type signature`. For example, the type declaration

```
thf(const_in,type,(  
    in: ( $i > ( $i > $o ) > $o ) )).
```

declares the symbol `in` (for  $\in$ ), to be of type  $\iota \rightarrow (\iota \rightarrow o) \rightarrow o$ . Thus `in` expects an element of type  $\iota$  and a set of type  $\iota \rightarrow o$  as its arguments. The mapping arrow is right associative. Type declarations use rules starting at `<thf_typed_const>` in the BNF. Note the use of the TPTP interpreted symbols `$i` and `$o` for the standard types  $\iota$  and  $o$ , which are built in to THF0. In addition to `$i` and `$o`, THF0 has the defined type `$tType` that denotes the collection of all types, and `$iType/$oType` as synonyms for `$i/$o`. Further base types can be introduced (as `<system_type>`s) on the fly. For example, the following introduces the base type `u` together with a corresponding constant symbol `in_u` of type  $u \rightarrow (u \rightarrow o) \rightarrow o$ .

```
thf(type_u,type,(  
    u: $tType)).  
  
thf(const_in_u,type,(  
    in_u: ( u > ( u > $o ) > $o ) )).
```

THF0 does not support polymorphism, product types or dependent types – such language constructs are addressed in THF1.

The next three annotated formulae of the example are *axioms* that specify the meanings of  $\in$ ,  $\cap$ , and  $\cup$ . A THF0 logical formula can use the basic TPTP connectives,  $\lambda$ -abstraction using the `^` quantifier followed by a list of typed  $\lambda$ -bound variables in square brackets, and function application using the `@` connective. Additionally, universally and existentially quantified variables must be typed. For example, the axiom

```
thf(ax_in,axiom,(  
    ( in  
    = ( ^ [X: $i,S: ( $i > $o )] :  
        ( S @ X ) ) ) ).
```

---

<sup>5</sup> It will also be used in the forthcoming typed first-order form (TFF) TPTP language.

```

%-----
%---Signatures for basic set theory predicates and functions.
thf(const_in,type,
  in: $i > ( $i > $o ) > $o )).

thf(const_intersection,type,
  intersection: ( $i > $o ) > ( $i > $o ) > ( $i > $o ) )).

thf(const_union,type,
  union: ( $i > $o ) > ( $i > $o ) > ( $i > $o ) )).

%---Some axioms for basic set theory. These axioms define the set
%---operators as lambda-terms. The general idea is that sets are
%---represented by their characteristic functions.
thf(ax_in,axiom,
  ( in
  = ( ^ [X: $i,S: ( $i > $o )] :
    ( S @ X ) ) )).

thf(ax_intersection,axiom,
  ( intersection
  = ( ^ [S1: ( $i > $o ),S2: ( $i > $o ),U: $i] :
    ( ( in @ U @ S1 )
      & ( in @ U @ S2 ) ) ) ) )).

thf(ax_union,axiom,
  ( union
  = ( ^ [S1: ( $i > $o ),S2: ( $i > $o ),U: $i] :
    ( ( in @ U @ S1 )
      | ( in @ U @ S2 ) ) ) ) )).

%---The distributivity of union over intersection.
thf(thm_distr,conjecture,
  ! [A: ( $i > $o ),B: ( $i > $o ),C: ( $i > $o )] :
  ( ( union @ A @ ( intersection @ B @ C ) )
    = ( intersection @ ( union @ A @ B ) @ ( union @ A @ C ) ) ) )).
%-----

```

**Fig. 1.** Distribution of Union over Intersection, encoded in THF0

specifies the meaning of the constant  $\in$  by equating it to  $\lambda X_i.\lambda S_{i \rightarrow o}.(S X)$ . Thus elementhood of an element  $X$  in a set  $S$  is reduced to applying the set  $S$  (seen as its characteristic function  $S$ ) to  $X$ . Using `in`, intersection is then equated to  $\lambda S_{i \rightarrow o}^1.\lambda S_{i \rightarrow o}^2.\lambda U_i.(\in U S^1) \wedge (\in U S^2)$ , and union is equated to  $\lambda S_{i \rightarrow o}^1.\lambda S_{i \rightarrow o}^2.\lambda U_i.(\in U S^1) \vee (\in U S^2)$ . Logical formulae use rules starting at `<thf_logic_formula>` in the BNF. The explicit function application operator

$\circledast$  is necessary for parsing function application expressions in Prolog. Function application is left associative.

The last annotated formula of the example is the *conjecture* to be proved.

```
thf(thm_distr,conjecture,
! [A: ( $i > $o ),B: ( $i > $o ),C: ( $i > $o )] :
( ( union @ A @ ( intersection @ B @ C ) )
= ( intersection @ ( union @ A @ B )
@ ( union @ A @ C ) ) ) ).
```

This encodes  $\forall A_{\iota \rightarrow o} \forall B_{\iota \rightarrow o} \forall C_{\iota \rightarrow o} (A \cup (B \cap C)) = ((A \cup B) \cap (A \cup C))$ . It uses rules starting at `<thf_quantified_formula>` in the BNF.

An advantage of higher-order logic over first-order logic is that the  $\forall$  and  $\exists$  quantifiers can be encoded using higher-order abstract syntax: Quantification is expressed using the logical constant symbols  $\Pi$  and  $\Sigma$  in connection with  $\lambda$ -abstraction. Higher-order systems typically make use of this feature in order to avoid introducing and supporting binders in addition to  $\lambda$ . THF0 provides the logical constants  $!!$  and  $??$  for  $\Pi$  and  $\Sigma$  respectively, and leaves use of the universal and existential quantifiers open to the user. Here is an encoding of the conjecture from the example, using  $!!$  instead of  $!$ .

```
thf(thm_distr,conjecture,
!! ( ^ [A: $i > $o,B: $i > $o,C: $i > $o] :
( ( union @ A @ ( intersection @ B @ C ) )
= ( intersection @ ( union @ A @ B )
@ ( union @ A @ C ) ) ) ) ).
```

Figure 2 presents the axioms from another example problem encoded in THF0. The example is from the domain of puzzles, and it encodes the following ‘Knights and Knaves Puzzle’:<sup>6</sup> *A very special island is inhabited only by knights and knaves. Knights always tell the truth. Knaves always lie. You meet two inhabitants: Zoey and Mel. Zoey tells you that Mel is a knave. Mel says, ‘Neither Zoey nor I are knaves.’ Can you determine who is a knight and who is a knave?* Puzzles of this kind have been discussed extensively in the AI literature. Here, we illustrate that an intuitive and straightforward encoding can be achieved in THF0. This encoding embeds formulas (terms of Boolean type) in terms and quantifies over variables of Boolean type; see for instance the `kk_6_2` axiom.

## 4 Type Checking THF0

The THF0 syntax provides a lot of flexibility. As a result, many syntactically correct expressions are meaningless because they are not well typed. For example, it does not make sense to say  $\& = \sim$  because the equated expressions have different types. It is therefore necessary to type check expressions using an inference system. Representative typing rules are given in Fig. 3 (the missing rules are obviously analogous to those provided). Here  $\$tType$  denotes the collection of all types, and  $S :: A$  denotes the judgement that  $S$  has type  $A$ .

---

<sup>6</sup> This puzzle was auto-generated by a computer program, written by Zac Ernst.

```

%-----
%---A very special island is inhabited only by knights and knaves.
thf(kk_6_1,axiom,( ! [X: $i] : ( ( is_a @ X @ islander ) => ( ( is_a @ X @ knight ) | ( is_a @ X @ knave ) ) ) ). 

%---Knights always tell the truth.
thf(kk_6_2,axiom,( ! [X: $i] : ( ( is_a @ X @ knight ) => ( ! [A: $o] : ( says @ X @ A ) => A ) ) ) ). 

%---Knaves always lie.
thf(kk_6_3,axiom,( ! [X: $i] : ( ( is_a @ X @ knave ) => ( ! [A: $o] : ( says @ X @ A ) => ~ A ) ) ) ). 

%---You meet two inhabitants: Zoey and Mel.
thf(kk_6_4,axiom,( ( is_a @ zoey @ islander ) & ( is_a @ mel @ islander ) ) ). 

%---Zoey tells you that Mel is a knave.
thf(kk_6_5,axiom,( says @ zoey @ ( is_a @ mel @ knave ) ) ). 

%---Mel says, 'Neither Zoey nor I are knaves.'
thf(kk_6_6,axiom,( says @ mel @ ~ ( ( is_a @ zoey @ knave ) | ( is_a @ mel @ knave ) ) ) ). 

%---Can you determine who is a knight and who is a knave?
thf(query,theorem,( ? [Y: $i,Z: $i] : ( ( Y = knight <~> Y = knave ) & ( Z = knight <~> Z = knave ) & ( is_a @ mel @ Y ) & ( is_a @ zoey @ Z ) ) ) ). 
%-----
```

**Fig. 2.** A ‘Knights and Knaves Puzzle’ encoded in THF0

$$\begin{array}{c}
\frac{}{\$i :: \$tType} \quad \frac{}{\$o :: \$tType} \quad \frac{}{\Gamma \vdash \$true :: \$o} \quad \frac{\text{thf}(\_, \text{type}, c : A)}{\Gamma \vdash c :: A} \\
\\
\frac{A :: \$tType \quad B :: \$tType}{A > B :: \$tType} \quad \frac{X :: A \text{ in } \Gamma}{\Gamma \vdash X :: A} \\
\\
\frac{\Gamma, X :: A \vdash S :: B}{\Gamma \vdash ^ [X : A] : S :: A > B} \quad \frac{\Gamma \vdash F :: A > B \quad \Gamma \vdash S :: A}{\Gamma \vdash F @ S :: B} \\
\\
\frac{\Gamma \vdash F :: A > \$o}{\Gamma \vdash !! F :: \$o} \quad \frac{\Gamma, X :: A \vdash F :: \$o}{\Gamma \vdash ! [X : A] : F :: \$o} \quad \frac{\Gamma \vdash F :: \$o}{\Gamma \vdash \sim F :: \$o} \\
\\
\frac{\Gamma \vdash F :: \$o \quad \Gamma \vdash G :: \$o}{\Gamma \vdash F \& G :: \$o} \quad \frac{\Gamma \vdash S :: A \quad \Gamma \vdash S' :: A}{\Gamma \vdash S = S' :: \$o}
\end{array}$$

**Fig. 3.** Typing rules of THF0

The typing rules serve only to provide an intuition. The normative definition is given by representing THF0 in the logical framework LF [14]. LF is a dependent type theory related to Martin-Löf type theory [18], particularly suited to logic representations. In particular, the maintenance of the context, substitution, and  $\alpha$ -conversion are handled by LF and do not have to be specified separately. A THF0 expression is well typed if its translation to LF is, and THF0 expressions can be type checked using existing tools such as Twelf [23]. In the following, Twelf syntax is used to describe the representation.

To represent THF0, a base signature  $\Sigma_0$  is defined, as given in Figure 4. It is similar to the Harper et al. encoding of higher-order logic [14]. Every list  $L$  of TPTP formulas can be translated to a list of declarations  $\Sigma_L$  extending  $\Sigma_0$ . A list  $L$  is well-formed iff  $\Sigma_0, \Sigma_L$  is a well-formed Twelf signature. The signature of the translation is given in Figure 5.

In Figure 4,  $\$tType : \text{type}$  means that the name  $\$tType$  is introduced as an LF type. Its LF terms are the translations of THF0 types. The declaration  $\$tm : \$tType \rightarrow \text{type}$  introduces a symbol  $\$tm$ , which has no analog in THF0: for every THF0 type  $A$  it declares an LF type  $\$tm A$ , which holds the terms of type  $A$ .  $\$i$  and  $\$o$  are declared as base types, and  $>$  is the function type constructor. Here  $\rightarrow$  is the LF symbol for function spaces, i.e., the declaration of  $>$  means that it takes two types as arguments and returns a type. Thus, on the ASCII level, the translation from THF0 types to LF is the identity. The symbols  $\wedge$  through  $\exists$  declare term and formula constructors with their types. The binary connectives other than  $\&$  have been omitted - they are declared just like  $\&$ . In Twelf, equality and inequality are actually ternary symbols, because they take the type  $A$  of the equated terms as an additional argument. By using implicit arguments, Twelf is able to reconstruct  $A$  from the context, so that equality behaves essentially like a binary symbol. Except for equality, the same ASCII notation can be used in Twelf as in THF0 so that on the ASCII level most cases of the translation are trivial. The translation of the binders  $\wedge$ ,  $!$ , and  $\exists$  is interesting: they are all expressed by the  $\lambda$  binder of Twelf. For example, if  $F$

is translated to  $F'$ , then  $\forall x_{\$i}.F$  is translated to  $!(\lambda x_{\$tm} \$i.F')$ . Since the Twelf ASCII syntax for  $\lambda x_{\$tm} \$i.F'$  is simply `[x:$tm $i] F'`, on the ASCII level the translation of binders consists of simply inserting `$tm` and dropping a colon. For all binders, the type `A` of the bound variable is an implicit argument and thus reconstructed by Twelf. Of course, most of the term constructors can be defined in terms of only a few primitive ones. In particular, the Twelf symbol `!` can be defined as the composition of `!!` and `^`, and similarly `?`. However, since they are irrelevant for type checking, definitions are not used here.

```

$tType  : type.
$tm     : $tType -> type.
$i      : $tType.
$o      : $tType.
>      : $tType -> $tType -> $tType.

^      : ($tm A -> $tm B) -> $tm (A > B).
@      : $tm(A > B) -> $tm A -> $tm B.
$true   : $tm $o.
$false  : $tm $o.
~      : $tm $o -> $tm $o.
&      : $tm $o -> $tm $o -> $tm $o.
==     : $tm A -> $tm A -> $tm $o.
!=     : $tm A -> $tm A -> $tm $o.

!      : ($tm A -> $tm $o) -> $tm $o.
?      : ($tm A -> $tm $o) -> $tm $o.
!!     : ($tm(A > $o)) -> $tm $o.
??     : ($tm(A > $o)) -> $tm $o.

$istru e : $tm $o -> type.

```

**Fig. 4.** The base signature for Twelf

| THF0                         | LF                                  |
|------------------------------|-------------------------------------|
| types                        | terms of type <code>\$tType</code>  |
| terms of type <code>A</code> | terms of type <code>\$tm A</code>   |
| formulas                     | terms of type <code>\$tm \$o</code> |

**Fig. 5.** Correspondences between THF0 and the LF representation

Figure 6 shows the translation of the formulae from Figure 1. A THF0 type declaration for a constant `c` with type `A` is translated to the Twelf declaration `c : $tm A`. An axiom or conjecture `F` is translated to a Twelf declaration for the

type `$istru F'`, where `F'` is the translation of `F`. In the latter case the Twelf name of the declaration is irrelevant for type-checking. By using the role, i.e., axiom or conjecture, as the Twelf name, the role of the original TPTP formula can be recovered from its Twelf translation.

```

in          : $tm($i > ($i > $o) > $o).
intersection : $tm(($i > $o) > ($i > $o) > ($i > $o)).
union       : $tm(($i > $o) > ($i > $o) > ($i > $o)).
axiom       : $istru in == ^[X : $tm $i] ^[S : $tm($i > $o)](S @ X).
axiom       : $istru intersection ==
              ^[S1: $tm($i > o)] ^[S2: $tm($i > $o)] ^[U: $tm $i]
              ((in @ U @ S1) & (in @ U @ S2)).
axiom       : $istru union ==
              ^[S1: $tm($i > $o)] ^[S2: $tm($i > $o)] ^[U: $tm $i]
              (in @ U @ S1) | (in @ U @ S2).
conjecture  : $istru
              !! [A: $tm($i > $o)] !! [B: $tm($i > $o)] !! [C: $tm($i > $o)]
              ((union @ A @ (intersection @ B @ C))
              == (intersection @ (union @ A @ B) @ (union @ A @ C))).
```

**Fig. 6.** Twelf translation of Figure 1

Via the Curry-Howard correspondence [10, 17], the representation of THF0 in Twelf can be extended to represent proofs – it is necessary only to add declarations for the proof rules to  $\Sigma_0$ , i.e.,  $\beta$ - $\eta$ -conversion, and the rules for the connectives and quantifiers. If future versions of THF0 provide a standard format for explicit proof terms, Twelf can serve as a neutral trusted proof checker.

## 5 TPTP Resources

The first-order TPTP provides a range of resources to support use of the problem library and related infrastructure. Many of these resources are immediately applicable to the higher-order setting, while some require changes to reflect the new features in the THF0 language.

The main resource for users is the TPTP problem library itself. At the time of writing around 100 THF problems have been collected, and are being prepared for addition to a THF extension of the library. THF file names have an `@` separator between the abstract problem name and the version number (corresponding to the `-` in CNF problem names and the `+` in FOF problem names), e.g., the example problem in Figure 1 will be put in `SET171@4.p`. The existing header fields of TPTP problems have been slightly extended to deal with higher-order features. First, the `Status` field, which records the semantic status of the problem in terms of the Szs ontology, provide status values for each semantics of interest [3, 4]. Second, the `Syntax` field, which records statistics of syntactic

characteristics of the problem, has been extended to include counts of the new connectives that are part of THF0.

Tools that support the TPTP include the `tptp2X` and `tptp4X` utilities, which read, analyse, transform, and output TPTP problems. `tptp2X` is written in Prolog, and it has been extended to read, analyse, and output problems written in THF0. The translation to Twelf syntax has been implemented as a `tptp2X` format module, producing files that can be type-checked directly. The Twelf translation has been used to type-check (and in some cases correct) the THF problems collected thus far. The extended `tptp2X` is part of TPTP v3.4.0. `tptp4X` is based on the `JJParser` library of code written in C. This will be extended to cope with higher-order formulae.

The flipside of the TPTP problem library is the TSTP solution library. Once the higher-order part of the TPTP problem library is in place it is planned to extend the TSTP to include results from higher-order ATP systems on the THF problems. The harnesses used for building the first-order TSTP will be used as-is for the higher-order extension.

A first competition “happening” for higher-order ATP systems that can read THF0 will be held at IJCAR 2008. This event will be similar to the CASC competition for first-order ATP systems, but with a less formal evaluation phase. It will exploit and test the THF0 language.

## 6 Conclusion

This paper has described, with examples, the core of the TPTP language for higher-order logic (Church’s simple type theory) – THF0. The TPTP infrastructure is being extended to support problems, solutions, tools, and evaluation of ATP systems using the THF0 language. Development and adoption of the THF0 language and associated TPTP infrastructure will support research and development in automated higher-order reasoning, providing leverage for progress leading to effective and successful application. To date only the LEO II system [7] is known to use the THF0 language. It is hoped that more high-order reasoning systems and tool will adopt the THF language, making easy and direct communication between the systems and tools possible.

**Acknowledgements:** Chad Brown contributed to the design of the THF language. Allen Van Gelder contributed to the development of the THF syntax and BNF. Frank Theiss and Arnaud Fietzke implemented the LEO-II parser for THF0 language and provided useful feedback.

## References

1. P.B. Andrews, M. Bishop, S. Issar, Nesmith. D., F. Pfenning, and H. Xi. TPS: A Theorem-Proving System for Classical Type Theory. *Journal of Automated Reasoning*, 16(3):321–353, 1996.

2. M. Beeson. Otter-lambda, a Theorem-prover with Untyped Lambda-unification. In G. Sutcliffe, S. Schulz, and T. Tammet, editors, *Proceedings of the Workshop on Empirically Successful First Order Reasoning, 2nd International Joint Conference on Automated Reasoning*, 2004.
3. C. Benzmüller and C. Brown. A Structured Set of Higher-Order Problems. In J. Hurd and T. Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, number 3606 in Lecture Notes in Artificial Intelligence, pages 66–81. Springer-Verlag, 2005.
4. C. Benzmüller, C. Brown, and M. Kohlhase. Higher-order Semantics and Extensionality. *Journal of Symbolic Logic*, 69(4):1027–1088, 2004.
5. C. Benzmüller and M. Kohlhase. LEO - A Higher-Order Theorem Prover. In C. Kirchner and H. Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction*, number 1421 in Lecture Notes in Artificial Intelligence, pages 139–143. Springer-Verlag, 1998.
6. C. Benzmüller and L. Paulson. Exploring Properties of Normal Multimodal Logics in Simple Type Theory with LEO-II. In C. Benzmüller, C. Brown, J. Siekmann, and R. Statman, editors, *Festschrift in Honour of Peter B. Andrews on his 70th Birthday*, page To appear. IfCoLog, 2007.
7. C. Benzmüller, L. Paulson, F. Theiss, and A. Fietzke. Progress Report on LEO-II - An Automatic Theorem Prover for Higher-Order Logic. In K. Schneider and J. Brandt, editors, *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics*, 2007.
8. C. Benzmüller, V. Sorge, M. Jamnik, and M. Kerber. Combined Reasoning by Automated Cooperation. *Journal of Applied Logic*, page To appear, 2007.
9. A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:5668, 1940.
10. H.B. Curry and R. Feys. *Combinatory Logic I*. North Holland, Amsterdam, 1958.
11. F. Frege. *Grundgesetze der Arithmetik*. Jena, 1893,1903.
12. P. Godefroid. Software Model Checking: the VeriSoft Approach. Technical Report Technical Memorandum ITD-03-44189G, Bell Labs, Lisle, USA, 2003.
13. M. Gordon and T. Melham. *Introduction to HOL, a Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
14. R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 40(1):143–184, 1993.
15. J. Harrison. HOL Light: A Tutorial Introduction. In M. Srivas and A. Camilleri, editors, *Proceedings of the 1st International Conference on Formal Methods in Computer-Aided Design*, number 1166 in Lecture Notes in Computer Science, pages 265–269. Springer-Verlag, 1996.
16. L. Henkin. Completeness in the Theory of Types. *Journal of Symbolic Logic*, 15:81–91, 1950.
17. W. Howard. The Formulas-as-types Notion of Construction. In J. Seldin and J. Hindley, editors, *To H B Curry, Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.
18. P. Martin-Löf. An Intuitionistic Theory of Types. In G. Sambin and J. Smith, editors, *iTwenty-Five Years of Constructive Type Theory*, pages 127–172. Oxford University Press, 1973.
19. C. Matuszek, Cabral J., M. Witbrock, and J. DeOliveira. An Introduction to the Syntax and Content of Cyc. In Baral C., editor, *Proceedings of the 2006 AAAI Spring Symposium on Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*, pages 44–49, 2006.

20. I. Niles and A. Pease. Towards A Standard Upper Ontology. In C. Welty and B. Smith, editors, *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems*, pages 2–9, 2001.
21. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Number 2283 in Lecture Notes in Computer Science. Springer-Verlag, 2002.
22. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 411–414. Springer-Verlag, 1996.
23. F. Pfenning and C. Schürmann. System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, number 1632 in Lecture Notes in Artificial Intelligence, pages 202–206. Springer-Verlag, 1999.
24. P. Rudnicki. An Overview of the Mizar Project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 311–332, 1992.
25. J.H. Siekmann, C. Benzmüller, V. Brezhnev, L. Cheikhrouhou, A. Fiedler, A. Franke, H. Horacek, M. Kohlhase, A. Meier, E. Melis, M. Moschner, I. Normann, M. Pollet, V. Sorge, C. Ullrich, C.P. Wirth, and J. Zimmer. Proof Development with OMEGA. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, number 2392 in Lecture Notes in Artificial Intelligence, pages 143–148. Springer-Verlag, 2002.
26. G. Sutcliffe. TPTP, TSTP, CASC, etc. In V. Diekert, M. Volkov, and A. Voronkov, editors, *Proceedings of the 2nd International Computer Science Symposium in Russia*, number 4649 in Lecture Notes in Computer Science, pages 7–23. Springer-Verlag, 2007.
27. G. Sutcliffe, S. Schulz, K. Claessen, and A. Van Gelder. Using the TPTP Language for Writing Derivations and Finite Interpretations. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, number 4130 in Lecture Notes in Artificial Intelligence, pages 67–81, 2006.
28. G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.
29. G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
30. G. Sutcliffe, J. Zimmer, and S. Schulz. TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In W. Zhang and V. Sorge, editors, *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, number 112 in Frontiers in Artificial Intelligence and Applications, pages 201–215. IOS Press, 2004.
31. A. Van Gelder and G. Sutcliffe. Extending the TPTP Language to Higher-Order Logic with Automated Parser Generation. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, number 4130 in Lecture Notes in Artificial Intelligence, pages 156–161. Springer-Verlag, 2006.
32. E. Zermelo. Über Grenzzahlen und Mengenbereiche. *Fundamenta Mathematicae*, 16:29–47, 1930.

## A BNF for THF0

The BNF uses a modified BNF meta-language that separates syntactic, semantic, lexical, and character-macro rules [27]. Syntactic rules use the standard ::= separator, semantic constraints on the syntactic rules use a :== separator, rules that produce tokens from the lexical level use a ::- separator, and the bottom level character-macros are defined by regular expressions in rules using a ::: separator. The BNF is easy to translate into parser-generator (`lex/yacc`, `antlr`, etc.) input [31].

The syntactic and semantic grammar rules for THF0 are presented here. The rules defining tokens and character macros are available from the TPTP web site, [www.tptp.org](http://www.tptp.org).

```
%-----
%---Files. Empty file is OK.
<TPTP_file>      ::= <TPTP_input>*
<TPTP_input>     ::= <annotated_formula> | <include>

%---Formula records
<annotated_formula> ::= <thf_annotated>
<thf_annotated>   ::= thf(<name>,<formula_role>,<thf_formula><annotations>).
<annotations>    ::= <null> | ,<source><optional_info>
%---In derivations the annotated formulae names must be unique, so that
%---parent references (see <inference_record>) are unambiguous.

%---Types for problems.
<formula_role>   ::= <lower_word>
<formula_role>   ::= axiom | hypothesis | definition | assumption |
                     lemma | theorem | conjecture | negated_conjecture |
                     plain | fi_domain | fi_functors | fi_predicates |
                     type | unknown

%-----
%---THF0 formulae. All formulae must be closed.
<thf_formula>    ::= <thf_logic_formula> | <thf_typed_const>
<thf_logic_formula> ::= <thf_binary_formula> | <thf_unitary_formula>
<thf_binary_formula> ::= <thf_pair_binary> | <thf_tuple_binary>
%---Only some binary connectives can be written without ()s.
%---There's no precedence among binary connectives
<thf_pair_binary>  ::= <thf_unitary_formula> <thf_pair_connective>
                      <thf_unitary_formula>
%---Associative connectives & and | are in <assoc_formula>.
<thf_tuple_binary>  ::= <thf_or_formula> | <thf_and_formula> |
                      <thf_apply_formula>
<thf_or_formula>   ::= <thf_unitary_formula> <vline> <thf_unitary_formula> |
                      <thf_or_formula> <vline> <thf_unitary_formula>
<thf_and_formula>  ::= <thf_unitary_formula> & <thf_unitary_formula> |
                      <thf_and_formula> & <thf_unitary_formula>
<thf_apply_formula> ::= <thf_unitary_formula> @ <thf_unitary_formula> |
                      <thf_apply_formula> @ <thf_unitary_formula>
%---<thf_unitary_formula> are in ()s or do not have a <binary_connective>
%---at the top level. Essentially, a <thf_unitary_formula> is any lambda
%---expression that "has enough parentheses" to be used inside a larger
%---lambda expression. However, lambda notation might not be used.
<thf_unitary_formula> ::= <thf_quantified_formula> | <thf_abstraction> |
                         <thf_unary_formula> | <thf_atom> |
                         (<thf_logic_formula>)
<thf_quantified_formula> ::= <thf_quantified_var> | <thf_quantified_noVar>
<thf_quantified_var>  ::= <quantifier> [<thf_variable_list>] :
                         <thf_unitary_formula>
<thf_quantified_noVar> ::= <thf_quantifier> (<thf_unitary_formula>)
%---@ (denoting apply) is left-associative and lambda is right-associative.
<thf_abstraction>   ::= <thf_lambda> [<thf_variable_list>] :
                         <thf_unitary_formula>
```

```

<thf_variable_list> ::= <thf_variable> | <thf_variable>,<thf_variable_list>
<thf_variable>      ::= <variable> | <thf_typed_variable>
<thf_typed_variable> ::= <variable> : <thf_top_level_type>
%---Unary connectives bind more tightly than binary. The negated formula
%---must be ()ed because a ~ is also a term.
<thf_unary_formula> ::= <thf_unary_connective> (<thf_logic_formula>)

%---An <thf_typed_const> is a global assertion that the atom is in this type.
<thf_typed_const>    ::= <constant> : <thf_top_level_type> | (<thf_typed_const>)

%---THF atoms
<thf_atom>          ::= <constant> | <defined_constant> | <system_constant> |
                           <variable> | <thf_conn_term>
%---<defined_constant> is really a <defined_prop>, but that's the syntax rule
%---used. <thf_atom> can also be <defined_type>, but they are syntactically
%---captured by <defined_constant>. Ditto for <system_*>.

%---<thf_top_level_type> appears after ":", where a type is being specified
%---for a term or variable.
<thf_unitary_type>   ::= <constant> | <variable> | <defined_type> |
                           <system_type> | (<thf_binary_type>)
<thf_top_level_type> ::= <constant> | <variable> | <defined_type> |
                           <system_type> | <thf_binary_type>
<thf_binary_type>    ::= <thf_mapping_type> | (<thf_binary_type>)
<thf_mapping_type>   ::= <thf_unitary_type> <arrow> <thf_unitary_type> |
                           <thf_unitary_type> <arrow> <thf_mapping_type>
%-----
%---Special higher order terms
<thf_conn_term>     ::= <thf_quantifier> | <thf_pair_connective> |
                           <assoc_connective> | <thf_unary_connective>

%---Connectives - THF
<thf_lambda>         ::= ^
<thf_quantifier>     ::= !! | ??
<thf_pair_connective> ::= <defined_infix_pred> | <binary_connective>
<thf_unary_connective> ::= <unary_connective>
%---Connectives - FOF
<quantifier>         ::= ! | ?
<binary_connective>   ::= <=> | => | <= | <~> | ^<vline> | ~&
<assoc_connective>    ::= <vline> | &
<unary_connective>    ::= ~

%---Types for THF and TFF
<defined_type>        ::= <atomic_defined_word>
<defined_type>        ::= $oType | $o | $iType | $i | $tType
%---$oType/$o is the Boolean type, i.e., the type of $true and $false.
%---$iType/$i is type of individuals. $tType is the type of all types.
<system_type>         ::= <atomic_system_word>

%---First order atoms
<defined_prop>        ::= <atomic_defined_word>
<defined_prop>        ::= $true | $false
<defined_infix_pred>   ::= = | !=

%---First order terms
<constant>            ::= <functor>
<functor>              ::= <atomic_word>
<defined_constant>     ::= <atomic_defined_word>
<system_constant>      ::= <atomic_system_word>
<variable>             ::= <upper_word>
%-----
%---General purpose
<name>                ::= <atomic_word> | <unsigned_integer>
<atomic_word>           ::= <lower_word> | <single_quoted>
<atomic_defined_word>  ::= <dollar_word>
<atomic_system_word>   ::= <dollar_dollar_word>
<null>                 ::=
%-----

```

# The MMT API

Florian Rabe

Computer Science, Jacobs University Bremen, Germany  
<http://trac.kwarc.info/MMT>

**Abstract.** The MMT language was developed as a scalable representation and interchange language for formal mathematical knowledge. It permits natural representations of the syntax and semantics of virtually all declarative languages while keeping MMT-based MKM services easy to implement. It is foundationally unconstrained and can be instantiated with specific formal languages.

The MMT API implements the MMT language along with multiple backends for persistent storage and frontends for machine and user access. Moreover, it implements a wide variety of MMT-based knowledge management services. The API and all services are generic and can be applied to any language represented in MMT. A plugin interface permits injecting syntactic and semantic idiosyncrasies of individual formal languages.

*The MMT Language* Content-oriented representation languages for mathematical knowledge are usually designed to focus on either of two goals: (i) the automation potential offered by mechanically verifiable representations, as pursued in semi-automated proof assistants like Isabelle (ii) the universal applicability offered by a generic meta-language, as pursued in XML-based content markup languages like OMDoc. MMT [12] (Module system for Mathematical Theories) was designed to realize both goals in one coherent system. Being a foundationally unconstrained language with a precise semantics, it permits very natural and adequate representations of individual languages.

Logical frameworks, logics, signatures, set theories, etc. are represented uniformly as MMT *theories*. These contain *symbol declarations*, which subsume constants, functions, and predicates as well as – via the Curry-Howard correspondence – judgments, inference rules, axioms, and theorems. MMT theories are related via *theory morphisms*, which subsume translations, functors, and models. Finally, MMT provides a module system for building large theories and morphisms via reuse and inheritance. Mathematical objects such as terms, formulas, and proofs are represented as OPENMATH objects [1], and theory morphisms move objects between theories in a truth-preserving way.

*The MMT API* Exploiting the small number of primitives in MMT, the MMT API provides a comprehensive, scalable implementation of MMT itself and of MMT-based knowledge management services. All algorithms are implemented generically, and all logic-specific aspects are relegated to thin plugin interfaces. It is written in the functional and object-oriented language Scala [10]. Scala is fully compatible with Java so that MMT plugins and programs using MMT can

be written in either language. The API depends on only one external library – the lean web server tiscaf [3]. Excluding plugins and libraries, it comprises about 10000 lines of Scala code compiling into about 2000 Java `class` files totaling about 2 MB of platform-independent bytecode. Sources, binaries, API documentation, and user manual are available at the project homepage.

*Knowledge Management Services* The MMT API provides a suite of coherently integrated MKM services. A *notation language* based on [8] is used to serialize MMT in arbitrary output formats. Notations are grouped into *styles*, and a rendering engine presents any MMT concept according to the chosen style.

MMT content can be organized in *archives* [5], a light-weight project abstraction to integrate source files, content markup, narrative structure, notation indices, and RDF-style relational indices. Archives can be built, indexed, and browsed, and a zip-based file format permits convenient distribution and reuse.

An *query language* [11] integrates hierachic, relational, and unification-based query paradigms. Recently developed services include an MMT-based *change management* infrastructure [6] and a universal *computation server*.

*User and System Interfaces* The API can be run as an application, in which case it responds with a shell that interacts via standard input/output. The shell is scriptable in the sense that sequences of commands can be read from files, which also provides an easy way for users to initialize and configure the system.

A second frontend is given by an HTTP server. For machine interaction, it exposes all functionality including services such as querying and computation. For human interaction, it offers an interactive browser based on HTML+presentation MathML. The latter is computed on demand by the rendering engine according to the style interactively selected by the user. Based on the JOBAD JavaScript library [4], user interaction is handled via JavaScript and Ajax. Example interface functions are definition lookup and type inference.

To facilitate distributing MMT content, all MMT knowledge items are referenced by canonical logical identifiers (URIs), and their physical locations (URLs) remain transparent. This is implemented as a catalog that translates MMT URIs into URLs. Catalog entries are created automatically when the user registers individual knowledge repositories. Besides the physical locations, the API also hides whether a knowledge item has been loaded into memory at all: Knowledge items are retrieved transparently when needed so that memory constraints are hidden from users.

Supported knowledge repositories are file systems, SVN working copies and repositories, and TNTBASE databases [13]. The latter also supports MMT-specific indexing and querying functions [9] permitting, e.g., the efficient retrieval of the dependency closure of an MMT knowledge item.

*Instantiating MMT for Specific Languages* In MMT representations, the underlying foundational language is represented explicitly as an MMT-theory  $M$  itself, and individual  $M$ -theories are formalized as MMT-theories  $T$  with *meta-theory*

$M$ . The meta-theory of  $T$  determines the syntax and semantics of declarations in  $T$ . Correspondingly, all API services are *foundation-independent* in the sense that they do not depend on the syntax and semantics of  $M$ . However,  $M$ -specific knowledge can be taken into account if it is provided via one of two plugin interfaces for the syntax and semantics of  $M$ .

Syntax plugins provide parsing methods that translate  $M$ -specific concrete syntax into MMT content markup. Such plugins are available for a varied list of languages: the ATP interface language TPTP, the ontology language OWL, the Mizar language for formalized mathematics, and the dependent type theory LF.

Semantics plugins provide methods for typing and equality of objects. Here the MMT module system remains transparent for the plugin so that only the core  $M$ -algorithms have to be implemented. For example, the semantics plugin for LF comprises only 200 lines of code. Moreover, being a logical framework, LF can serve as the meta-theory of many other formal systems, whose semantics becomes available without writing additional plugins.

The screenshot shows the MMT web interface. On the left, there is a tree view of a document derived from omdoc. The root node is a theory named IMPExt meta lf. Under this theory, there are several declarations, including imp2I and imp2E, which are highlighted in blue. Below these, there are remote module declarations for CONJExt, DISJExt, and Equiv. On the right, a context menu is open over the declaration imp2I. The menu has a title 'type' and contains the following items: 'infer type', 'reconstructed types', 'implicit arguments', 'implicit binders', 'redundant brackets', and 'Fold'. The 'infer type' item is currently selected.

```

document derived omdoc
remote module FalsityExt
remote module NEGExt
theory IMPExt meta lf
  include IMP
  imp2I : ((ded A → ded B → ded C) → ded A imp (B imp C))
    = [f:ded A → ded B → ded C]impl ([p:ded A]implI ([q:ded B]f p q))
  imp2E : (ded A imp (B imp C) → ded A → ded B → ded C)
    = [p:ded A imp (B imp C)][q:ded A][r:ded B]impE (impE p q) r
remote module CONJExt
remote module DISJExt
remote module Equiv

```

*An Example Use Case* The LATIN project [2, 7] builds an atlas of logics using the LF framework. Using the LF syntax plugin, the LATIN atlas can be maintained as an MMT archive. And using a style for rendering LF, this archive can be browsed via the MMT web server, from where the above screen shot is taken.

The selected declaration derives the rule  $\frac{A, B \vdash C}{\vdash A \text{ imp } (B \text{ imp } C)}$  in the theory IMP of the implication connective imp. Via the context menu, the user called type inference on the selected subobject, which resulted in a dialog displaying the *dynamically inferred type*.

The implementation of this feature uses server-side query evaluation. The rendering engine adds parallel markup annotations that identify (but not locate) the rendered content object. These are used by the JavaScript interface to build a query that is evaluated on the server and results in the rendered inferred type, which is sent back to the client and displayed. The query consists

of multiple steps, which retrieve the containing OPENMATH object, take the respective subobject, infer its type, and render it using the current style. For the type inference step, the query engine uses the meta-theory (here: LF) to select the appropriate semantics plugin, which performs the type inference.

Due to the genericity of MMT and the MMT API, all functionality can be made available to other languages at minimal cost, a process we call *rapid prototyping* for formal systems.

*Acknowledgements* Over the last 5 years, contributions to the API or to individual plugins have been made by Alin Iacob, Catalin David, Dimitar Misev, Fulya Horozal, Füsün Horozal, Maria Alecu, Mihnea Iancu, and Vladimir Zamdzhev. Some of them were partially supported by DFG grant KO-2428/9-1.

## References

1. S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See <http://www.openmath.org/standard/om20>.
2. M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. Project Abstract: Logic Atlas and Integrator (LATIN). In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 287–289. Springer, 2011.
3. A. Gaydenko. tiscaf http server, 2008. <http://gaydenko.com/scala/tiscaf/httpd/>.
4. J. Gićeva, C. Lange, and F. Rabe. Integrating Web Services into Active Mathematical Documents. In J. Carette, L. Dixon, C. Sacerdoti Coen, and S. Watt, editors, *Intelligent Computer Mathematics*, pages 279–293. Springer, 2009.
5. F. Horozal, A. Iacob, C. Jucovschi, M. Kohlhase, and F. Rabe. Combining Source, Content, Presentation, Narration, and Relational Representation. In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 211–226. Springer, 2011.
6. M. Iancu and F. Rabe. Management of Change in Declarative Languages. 2012.
7. M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN Project, 2009. see <https://trac.omdoc.org/LATIN/>.
8. M. Kohlhase, C. Müller, and F. Rabe. Notations for Living Mathematical Documents. In S. Autexier, J. Campbell, J. Rubio, V. Sorge, M. Suzuki, and F. Wiedijk, editors, *Mathematical Knowledge Management*, pages 504–519. Springer, 2008.
9. M. Kohlhase, F. Rabe, and V. Zholudev. Towards MKM in the Large: Modular Representation and Scalable Software Architecture. In S. Autexier, J. Calmet, D. Delahaye, P. Ion, L. Rideau, R. Rioboo, and A. Sexton, editors, *Intelligent Computer Mathematics*, pages 370–384. Springer, 2010.
10. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. artima, 2007.
11. F. Rabe. A Query Language for Formal Mathematical Libraries. 2012.
12. F. Rabe and M. Kohlhase. A Scalable Module System. see <http://arxiv.org/abs/1105.0548>, 2011.
13. V. Zholudev and M. Kohlhase. TNTBase: a Versioned Storage for XML. In *Proceedings of Balisage: The Markup Conference 2009*, volume 3 of *Balisage Series on Markup Technologies*. Mulberry Technologies, Inc., 2009.

# A Practical Module System for LF \*

Florian Rabe

Jacobs University Bremen  
f.rabe@jacobs-university.de

Carsten Schürmann

IT University of Copenhagen  
carsten@itu.dk

## Abstract

Module systems for proof assistants provide administrative support for large developments when mechanizing the meta-theory of programming languages and logics. We describe a module system for the logical framework LF that is based on two main primitives: signatures and signature morphisms. Signatures are defined as collections of constant declarations, and signature morphisms as homomorphism in between them. Our design is semantically transparent in the sense that it is always possible to elaborate modules into the module free version of LF. We have implemented our design as part of the Twelf system and rewritten parts of the Twelf example library to take advantage of the module system.

**Categories and Subject Descriptors** F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification; D.3.3 [*Language Constructs and Features*]: Modules, packages

**General Terms** Proof Assistants, Logical Frameworks

**Keywords** Twelf, modules, structures, views, Kolmogorov translation

## 1. Introduction

The Twelf system [Pfenning and Schürmann 1999] is a popular tool for reasoning about the design and properties of modern programming languages and logics. It has been used, for example, to verify the soundness of typed assembly language [Crary 2003] and Standard ML [Lee et al. 2007], for checking cut-elimination proofs for intuitionistic and classical logic [Pfenning 1995], and for specifying and validating logic morphisms, for example, between HOL and Nuprl [Schürmann and Stehr 2006]. Twelf, however, supports only monolithic proof developments and does not offer any support for modular proof engineering, composing logic morphisms, code reuse, or name space management. In this paper we develop a simple yet powerful module system for pure type systems in general, and therefore for the logical framework LF [Harper et al. 1993] in particular.

If one subscribes to the judgment-as-types methodology (as we do in the Twelf community), the equational theory underlying a

logical framework determines the application areas the framework performs well in. Twelf, for example, excels in the areas of programming languages and logics, where variable binding and substitution application are prevalent. It derives its strength from dependent types, higher-order abstract syntax, and the inductive definition of canonical forms justifies its use as a proof assistant.

Retrofitting a logical framework with a module system is a delicate undertaking. On the one hand, the module system should be as powerful as possible, convenient to use, and support brief, precise, and reusable program code. On the other hand, it must not break any of the features of the logical framework or of its reasoning and programming environments. Therefore, our design is conservative over LF. We guarantee that any development in LF with modules can be elaborated into LF without modules (to which we also refer as *core LF*).

The module system that we describe in this paper is deceptively simple. It introduces two new concepts, namely that of a signature and a signature morphism. A signature is simply a collection of constant declarations and constant definitions. Signature morphisms map terms valid in the source signature into terms valid in the target signature by replacing object-level and type-level constants with objects and types, respectively. This leads to the notion of signature graphs, which have proved to be simple, flexible, and scalable abstractions to express interrelations between signatures [Sannella and Tarlecki 1988, CoFI (The Common Framework Initiative) 2004, Autexier et al. 1999].

In our current design, we do not consider questions with regards to if and how signature morphisms preserve meta-theoretic properties such as, termination, totality, or coverage; these may have been established for type families in one signature but may or may not be preserved under signature morphisms. We realize the importance of this research problem and defer it to future work. In practice, however, not knowing the answer to this question is not a severe restriction. After elaboration, the set of tools and algorithms that are already part of the Twelf system, such as mode, termination, coverage analysis, etc. continue to work and can be applied to analyze the elaborated Twelf code.

We have implemented our design as part of the Twelf distribution, see <http://www.twelf.org/mod/> for details. We demonstrate in this paper that the module system allows for compact and elegant formalizations of logic morphisms when defining for example the Kolmogorov translation from classical into intuitionistic propositional logic in a modular manner. Other examples are available from the project homepage, including a modular and type-directed development of the meta theory of Mini-ML a modular definition of the algebraic hierarchy, and a soundness proof for first-order logic.

This paper is organized as follows. We briefly describe the relevant background of the logical framework LF and our running example in Section 2. In Section 3, we give a formal definition of the module system and its semantics, not only for LF but for pure type systems in general. In Section 4, we then provide experimental

\* The second author was in part supported by grant CCR-0325808 of the National Science Foundation and NABIT grant 2106-07-0019 of the Danish Strategic Research Council.

$$\begin{array}{c}
\frac{}{A \text{ true}} u \\
\vdots \\
B \text{ true} \quad \frac{A \supset B \text{ true} \quad A \text{ true}}{\supset I^u} \quad \frac{}{B \text{ true}} \supset E \\
\frac{}{A \supset B \text{ true}} \\
\\
\frac{}{A \text{ true}} u \\
\vdots \\
p \text{ true} \quad \frac{\neg A \text{ true} \quad A \text{ true}}{\neg I^{p,u}} \quad \frac{}{B \text{ true}} \neg E \\
\frac{}{\neg A \text{ true}}
\end{array}$$

**Figure 1.** Intuitionistic Logic

evidence that the Twelf module system does not degrade runtime performance by comparing the running times for Twelf vs. modular Twelf on large (non-modular) examples. Finally, we assess results and discuss future work in Section 5.

## 2. Preliminaries

### 2.1 The Logical Framework LF

The Twelf system is an implementation of the logical framework LF [Harper et al. 1993] designed as a meta-language for the representation of deductive systems, which we also call *core LF*. Judgments are represented as types, and derivations as objects:

$$\begin{array}{lll}
\text{Kinds:} & K ::= \text{type} \mid \{x:A\} K \mid A \rightarrow K \\
\text{Types:} & A, B ::= a \mid A M \mid \{x:A\} B \mid A \rightarrow B \\
\text{Objects:} & M ::= c \mid x \mid [x:A] M \mid M_1 M_2
\end{array}$$

where we write  $\{ \cdot \}$  for the  $\Pi$ -type constructor and  $[ \cdot ]$  for a  $\lambda$ -binder. We omit type labels whenever they are inferable. Core LF permits declarations of type- or object-level constants. Constant symbols may be declared (“ $a : K$ .” or “ $c : A$ .”) or defined (“ $a : K = A$ .” or “ $c : A = M$ .”). They may be used infix, for example by issuing “%infix  $n m c$ ” where  $n$  defines if  $c$  is left- or right-associative, and  $m$  the binding precedence of  $c$ . The Twelf system offers a variety of algorithms for checking the meta-theory of signatures, including termination, coverage, and totality, which we do not discuss further in this paper, but which remain available in modular Twelf.

### 2.2 The Kolmogorov Translation

As a running example, we have chosen to use the Kolmogorov translation that embeds classical logic into intuitionistic logic. We illustrate the characteristic features of the module system by defining the relevant logics through implication  $\supset$  and negation  $\neg$ . We say that  $A$  is true, if  $A \text{ true}$  can be derived using the rules depicted in Figure 1. By adding an axiom  $\neg\neg A \supset A \text{ true}$  (the law of double negation elimination), we obtain classical logic. The Kolmogorov translation uses double-negations to map formulas  $A$  to  $\bar{A}$  satisfying that  $A \text{ true}$  is derivable in classical logic iff  $\bar{A} \text{ true}$  is derivable in intuitionistic logic. For example, we have  $\overline{p \supset q} = \neg\neg(\neg\neg p \supset \neg\neg q)$  for propositional variables  $p, q$ .

## 3. The Module System

In the past, various module systems for proof assistants have been proposed, for example, for IMPS [Farmer et al. 1993], Agda [Norell 2007], Coq [Chrzaszcz 2003], Isabelle [Kammüller et al. 1999,

Haftmann and Wenzel 2007], and even LF [Harper and Pfenning 1998, Licata et al. 2006] itself.

In general, a module is an encapsulated context often with some free parameters. Depending on the module system, the modules may be called *signatures*, *theories*, or *functors*. Different from the module systems above, we use two kinds of signature morphisms to relate our modules. First, the instantiation of a parametrized module  $M$  induces a morphism from  $M$  into the context in which the instantiation occurs. This morphism is called a *structure* in SML (while the SML modules are called functors) and has a name; all module system designs for LF including ours follow this design choice. IMPS, Agda, and Isabelle, on the other hand, permit parametrized modules but do not use the concept of structures. Second, *views* are explicit morphisms used to translate between modules (sometimes called fitting morphisms). Views were pioneered in IMPS but are not used in any of the other systems. Our use of both structures and views is more related to module systems used in algebraic specification, e.g., in [Sannella and Tarlecki 1988] and [Autexier et al. 1999] where, however, structures are not named.

Furthermore, our system insists on elaborating the modular language into the core language, which is not possible for the Coq module system and not intended for LF/[Harper and Pfenning 1998]. Agda, Isabelle, and LF/[Licata et al. 2006] employ such an elaboration and use it to define the semantics of all modular constructs. In addition to the latter, our modular constructs have an elaboration-independent semantics, against which the elaboration is proved correct. This elaboration-independent semantics is defined in terms of morphisms and permits to reason about the adequacy of encodings without having to refer to the technicalities of the elaboration.

A comprehensive overview over the state of the art in module systems and their relations to ours is given in [Rabe 2008].

### 3.1 Syntax

Our modules are called **signatures**. A signature  $R, S, T$  is a named list of constant and structure declarations. Two signatures are equal iff they have the same name, which means that two signatures that contain the same declarations are not necessarily considered equal. This avoids complex equality reasoning about signatures, and is not a loss in expressivity because views, which we introduce in Section 3.1.2, can be used to establish isomorphisms between two such signatures.

EXAMPLE 1 (Judgments). The signature with name JUDGMENTS given below defines the judgments from Section 2.2.

```
%sig JUDGMENTS = {
  o : type.
  true : o -> type.
}.
```

It is easier to explain the module system for LF as an instance of the more general case: pure type systems [Barendregt 1991]. Therefore, we collapse the three syntactic categories of kinds, types, and objects into one single category of **terms**  $C$ :

$C ::= \vec{c} \mid x \mid \text{type} \mid \{x:C\} C \mid [x:C] C \mid C C$

The only difference to the core syntax is the case of references to constants, which are now qualified names  $\vec{c}$ . Qualified names access constants that are part of structures and are explained below.

Signature morphisms define mappings between signatures. A morphism  $\mu$  from a signature  $S$  to  $T$  maps every constant  $\vec{c}$  of  $S$  with type (kind)  $A$  to a term  $C$  over  $T$  such that  $C$  is typed (kinded) by  $\mu(A)$ . Here  $\mu(-)$  is the homomorphic extension of  $\mu$  to terms. The mapping  $\mu(-)$  preserves the typing relation and the

definitional equality of  $S$ , e.g., if  $B : A$  in  $S$ , then  $\mu(B) : \mu(A)$  in  $T$  (see, e.g., [Harper et al. 1994]).

In the following, we explain the two kinds of signature morphisms in detail: **structures** from  $S$  to  $T$  create an instance of  $S$  in  $T$ , and **views** from  $S$  to  $T$  define a translation from  $S$  to  $T$ .

### 3.1.1 Structures

In the simplest case, a structure declaration  $s : S$  occurring within a signature  $T$  has the following semantics: If  $S$  declares a constant  $c$  then the structure declaration induces a constant  $s.c$  in  $T$  by copying  $c$ . Because the structure declarations of  $S$  are also copied into  $T$ , the general way to refer to constants is via *qualified constant identifiers*  $\vec{c} ::= s. \dots .s.c$ . Similarly, we use *qualified structure identifiers*  $\vec{s} ::= s. \dots .s.s$ .

An important advantage of named structures and qualified identifiers is that  $T$  may contain multiple distinct instances of  $S$ . However, qualified identifiers can introduce a lot of clutter. In order to tame this clutter we introduce convenient **aliases** through the `%open` directive that permits the use of the constant name  $c$  instead of the qualified name  $s.c$ . Therefore, contrary to its analogue in SML, `%open` is syntactic sugar that provides a mechanism for introducing aliases that always resolve to the same internal identifier. Dealiasing is merely an implementation detail and therefore not discussed here.

**EXAMPLE 2** (Implication).  $\supset$  and its introduction and elimination rules from Figure 1 are encoded as follows:

```
%sig IMP = {
  %struct J : JUDGMENTS %open o true.
  ⊃ : o -> o -> o. %infix left 10 ⊃.
  ⊃I : (true A -> true B) -> true (A ⊃ B).
  ⊃E : true (A ⊃ B) -> true A -> true B.
}.
```

A structure  $s : S$  occurring in  $T$  induces a signature morphism from  $S$  to  $T$  as follows: Every constant  $\vec{c}$  of  $S$  is mapped to the constant  $s.\vec{c}$  of  $T$ . For example,  $J$  declared in **IMP** is a morphism from **JUDGMENTS** to **IMP**. If signatures are seen as records, then this is simply a record projection.

In the example above, we import `o` and `true` from **JUDGMENTS** using a structure  $J$  as abbreviations for the constants **IMP**" $J.o$  and **IMP**" $J.true$ . Within **IMP**, we refer to these constants as  $J.o$  and  $J.true$ .

**EXAMPLE 3** (Negation). Similarly, we encode negation and its rules:

```
%sig NEG = {
  %struct J : JUDGMENTS %open o true.
  ⊥ : o -> o.
  ⊥I : ({p} true A -> true p) -> true (⊥ A).
  ⊥E : true (⊥ A) -> true A -> true B.
  n = [p] (⊥ (⊥ p)).
  ⊥¬I : true A -> true (n A)
  = [D] (¬I [p:o] [u: true (⊥ A)] (¬E u D)).
}.
```

Negation satisfies the double negation introduction rule “If  $A$  true then  $\neg\neg A$  true.”. The proof is direct, and it defines the derived rule of inference  $\neg\neg I$ .

In addition to copying declarations from  $S$  to  $T$ , structures can instantiate constants and structures declared in  $S$  with corresponding expressions over  $T$ . We call such pairs of  $S$ -symbol and  $T$ -expression **assignments**.

**EXAMPLE 4** (Intuitionistic Logic). To obtain an encoding of intuitionistic logic as in Figure 1, we combine **IMP** and **NEG**. The common structure  $J$  must be shared:

```
%sig IL = {
  %struct I : IMP %open o true ⊃ ⊃I ⊃E.
  %struct N : NEG = { %struct J := I.J. }
  %open ⊥ n ¬I ¬E ¬¬I.
}.
```

Here `%struct J := I.J.` is an assignment:  $J$  refers to the structure declared in **NEG**, which is copied into **IL** resulting in the structure  $N.J$ ; assigning  $I.J$  to it, yields the desired sharing relation  $N.J = I.J$ . The assignment is well-typed because both  $N.J$  and  $I.J$  are instances of the same signature, namely **JUDGMENTS**.

In `%struct N : NEG = { %struct J := I.J.}`, readers familiar with SML may think of **NEG** as a functor, and of `{ %struct J := I.J.}` as its argument.

**EXAMPLE 5** (Classical Logic). Finally, by extending intuitionistic logic with the axiom of double negation elimination, we obtain the definition of classical logic.

```
%sig CL = {
  %struct IL : IL %open true ⊃ ⊥.
  dne : true (¬ (¬ A) ⊃ A).
}.
```

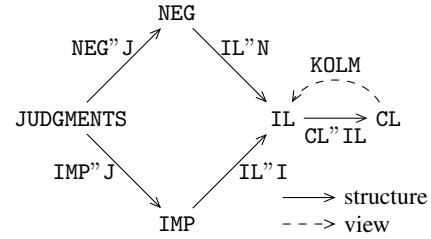
Formally, we define the body of a signature by

$$\Sigma ::= \cdot \mid \Sigma, D_c \mid \Sigma, D_s.$$

Here  $D_c$  stands for a constant declaration  $D_c ::= c : C \mid c : C = C$  and  $D_s ::= \vec{s} : T = \{\sigma\}$  stands for a structure declaration where

$$\sigma ::= \cdot \mid \sigma, \vec{c} := C \mid \sigma, \vec{s} := \mu$$

gives a list of assignments, where  $\vec{c}$  and  $\vec{s}$  are qualified constant and structure names, respectively. For the sake of convenience, we omit the keywords `%sig`, `%struct`, `%view` from the formal presentation.



A **signature graph** is a multi-graph with signatures as nodes and structures or views as edges. The signatures and structures introduced in the running example so far form the signature graph on the right. There, we use the **notation**  $S''$ 's to refer to the name  $s$  declared within the signature  $S$ . When talking about modular Twelf, we will occasionally use this notation to make references to constants and structures unambiguous.

### 3.1.2 Views

Next, we turn to views and define the view **KOLM** that interprets classical proofs over **CL** as intuitionistic proofs over **IL**. It is composed modularly from four different views into **IL**:

**EXAMPLE 6** (Kolmogorov view). We begin by translating the judgments in the view **KOLM** from **JUDGMENTS** to **IL**: The assignment `o := o` expresses that formulas are mapped to formulas, and the assignment `true := [x] true (n x)` expresses that the

judgment  $A$  true is mapped to the judgment  $\neg\neg B$  true where  $B$  is the translation of  $A$ . As for structures, the left hand side of an assignment is a symbol of the domain, and the right hand side is an expression over the codomain.

Similarly, we define the views KOLMJ and KOLMN, which translate implication and negation, respectively. The proof rules are translated to derived rules of inference, which are easily determined by pen and paper. These views are total: There is an assignment for every constant of the domain with the only exception of those that are defined.

Note that the assignments must abstract over the implicit arguments as well. For example, the assignment to  $\Box I$  must abstract over the implicit arguments  $A$  and  $B$  that occur in the (omitted) type of  $D$ .

```
%view KOLMJ : JUDGMENTS -> IL = {
  o   := o.
  true := [x] true (n x).
}.

%view KOLMI : IMP -> IL = {
  %struct J := KOLMJ.
  ⊃  := [x] [y] ((n x) ⊃ (n y)).
  □I := [A] [B] [D] ⊃I (□I D).
  □E := [A] [B] [D] [E] ⊃I [p] [u]
    →E D (¬I [q] [v]
    →E (□E v E) u).
}.

%view KOLMN : NEG -> IL = {
  %struct J := KOLMJ.
  ⊥  := [x] ⊥ x.
  ¬I := [A] [D] ¬I [q] [u]
    →E (D (¬ A) u) u.
  ¬E := [A] [C] [D] [E] ¬I [p] [u]
    →E D E .
}.

%view KOLM : CL -> IL = {
  %struct IL.I := KOLMI.
  %struct IL.N := KOLMN.
  dne := [A] ¬I [p] [u] →E u
    (□I [u] ¬I [p] [v] →E u
    (¬I [q] [w] →E w v)).
}.
```

In summary, the view KOLM is the Kolmogorov translation mapping the embedding from CL into IL. It illustrates nicely the expressive strength of what we call *deep* assignments: Instead of providing an assignment for the structure IL of CL, it assigns morphisms to the structures IL.I and IL.N. Intuitively, the assignment `%struct IL.I := KOLMI.` is justified as follows: IL.I is a copy of IMP into the domain CL of KOLM; thus, it is mapped to the view KOLMI, which is a translation of IMP into the codomain IL of KOLM. The last assignment in KOLM is the translation of the law of double negation elimination.

Thus, KOLM implements a meta-theoretic proof that classical proofs can be translated to intuitionistic ones: It covers all cases because it substitutes terms for all constants of CL. We do not provide any cases for variables, because inputs are always closed (do not contain free variables), and bound variables map to themselves. The application of KOLM is also clearly terminating.  $\square$

Formally, we write  $D_v := v : S \rightarrow T = \{\sigma\}$  for a view  $v$  from  $S$  to  $T$  where  $\sigma$  is as for structures except that it must provide assignments for *all* constants of  $S$  (except for those that have definitions).

Finally, given a set of signature and view declarations, we define signature morphisms by  $\mu ::= T''\vec{s} \mid v \mid \mu \mu'$ . Here  $T''\vec{s}$  refers to the structure  $\vec{s}$  of the signature  $T$ ,  $v$  refers to a view, and  $\mu \mu'$  represents the composition of two morphisms in diagrammatic order. In the running example,  $IL''N CL''IL KOLM$  is a morphism from NEG to IL.

Then we can define qualified structure identifiers precisely: The structure  $CL''IL.I.J$  is defined to be equal to the morphism  $IMP''I CL''IL$ .

Readers familiar with SML may interpret a view from  $S$  to  $T$  in two different ways. The view may be seen as a functor that is parametrized by a structure of type  $T$  (a morphism with domain  $T$ ) and produces a structure of type  $S$ . Alternatively, it may be seen as a structure implementing the signature  $S$  defined by explicitly providing values for all symbols of  $S$  – the role of  $T$  here is to represent the context in which  $S$  is implemented. Since LF is a logical framework and not a programming language, the implementation language must itself be represented as a signature, namely  $T$ . Thus, morphisms are paths in the signature graph.

This concludes the definition of the syntactic categories of our module system for LF, which we summarize in Figure 2.

### 3.2 Another Example

As a further example, we illustrate how signatures, structures, and views introduce module level parametricity. The List signature below is parametrized over a Monoid signature of elements and provide a relation that computes (parametrically) the fold of that list.

```
%sig Monoid = {
  a      : type.
  unit   : a.
  comp   : a -> a -> a -> type.
}.

%sig List = {
  %struct elem : Monoid.
  list   : type.
  nil    : list.
  cons   : elem.a -> list -> list.
  fold   : list -> elem.a -> type.
  foldnil : fold nil elem.unit.
  foldcons : fold L B -> elem.comp A B C
              -> fold (cons A L) C.
}.
```

The signature Monoid declares a monoid as a type together with the usual unit element and the binary composition operation (written as a relation), where we omit the axioms for simplicity. The signature List defines lists, which is parametric in type `elem.a`, and the fold relation, which is parametric in `elem.unit` and `elem.comp`.

Furthermore, we define a Monoid called NatMonoid that is based on natural numbers Nat.

```
%sig Nat = {
  nat    : type.
  zero   : nat.
  succ   : nat -> nat.
  add    : nat -> nat -> nat -> type.
  addzero : add N zero N.
  addsucc : add N P Q -> add N (succ P) (succ Q).
}.

%view NatMonoid : Monoid -> Nat = {
  a      : nat.
  unit   : zero.
}.
```

|                       |                 |   |
|-----------------------|-----------------|---|
| Signature graph       | $\mathcal{G}$   | $::= \cdot \mid \mathcal{G}, D_T \mid \mathcal{G}, D_v$           |
| Signature             | $D_T$           | $::= T = \{\Sigma\}$  |
| View                  | $D_v$           | $::= v : T \rightarrow T = \{\sigma\}$                            |
| Signature body        | $\Sigma$        | $::= \cdot \mid \Sigma, D_c \mid \Sigma, D_s$                     |
| Constant              | $D_c$           | $::= c : C \mid c : C = C$  |
| Structure             | $D_s$           | $::= s : T = \{\sigma\}$  |
| Assignment list       | $\sigma$        | $::= \cdot \mid \sigma, \vec{c} := C \mid \sigma, \vec{s} := \mu$ |
| Term                  | $C$             | $::= \vec{c} \mid type \mid \{x : C\}C \mid [x : C]C \mid C \ C$  |
| Morphism              | $\mu$           | $::= \vec{s} \mid v \mid \mu \mu$                                 |
| Qualified identifiers | $\vec{c}$       | $::= s. \dots .s.c$   |
|                       | $\vec{s}$       | $::= s. \dots .s.s$   |
| Identifiers           | $T, v, c, s, x$ |   |

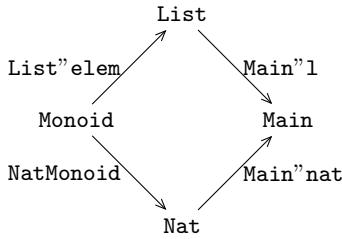
Figure 2. The Grammar for Expressions

```
comp := add.
}.
```

The view `NatMonoid` allows us to view `Nat` as a Monoid. Therefore, if we put all the pieces together, instantiate the parametric structure `elem`, we may now compute folds over natural numbers using Twelfs `%solve` directive.

```
%sig Main = {
  %struct nat : Nat.
  %struct l : List = {
    %struct elem := NatMonoid nat.
  }.
  %solve _ : fold
    (l.cons nat.zero (l.cons nat.zero l.nil)) N.
}.
```

Here the composite `NatMonoid nat` is the result of translating the structure `nat` along `NatMonoid` in order to fit it to the expected type of `elem`. The signature graph looks as below, and Theorem 9 below guarantees that it commutes.



The morphism denoted by `NatMonoid nat` is the result of applying `NatMonoid` to `nat`. It may be confusing that this application takes a structure of type `Nat` as its argument and returns a structure of type `Monoid`, although the view `NatMonoid` is typed as `Monoid -> Nat`. This is an effect of an adjunction between syntax and semantics: a view induces a translation of structures in the opposite direction. Here the translation of structures is seen as a semantical translation because every structure of type `S` is an implementation or a model of `S`.

### 3.3 Elaboration

In this section, we define the *elaboration semantics* of the module system into core LF. The target language of elaboration is that of Fig. 2 but without structure declarations and without assignments to structures. The elaboration of the remaining views to plain LF is straightforward.

The elaboration semantics provides implicitly an algorithm on how to *lookup* the type of a constant, for example, during type

checking. While this is trivial in the non-modular case, in a modular setting constants may be induced by structure declarations or appear in assignments or views.

In the remainder of this section, we define the judgments for elaborating declarations in a signature `T` and for morphisms `m` in a signature graph `G`, respectively:  $\mathcal{G} \ggg_T \vec{c} : A = B$  and  $\mathcal{G} \ggg_m \vec{c} := B$ . The former expresses that the declaration  $\vec{c} : A = B$  is present in the signature `T` after elaboration (or: that the lookup of the constant  $\vec{c}$  in the signature `T` returns the type `A` and the definition `B`). In the interest of brevity, we reuse the same judgment for defined and undefined constants, by writing `B = ⊥` in the latter case. Very similarly,  $\mathcal{G} \ggg_m \vec{c} := B$  expresses that the assignment  $\vec{c} := B$  is present in the view or structure `m` after elaboration (or: that the lookup of the assignment to  $\vec{c}$  in `m` returns `B`). Again we use `⊥` for brevity: If the domain of `m` has a constant  $\vec{c}$  but `m` provides no assignment for it, we write  $\mathcal{G} \ggg_m \vec{c} := \perp$ .

The intuitive definition of these judgments is as follows. Assume a structure declaration  $r : R = \{\sigma\}$  occurs in `S`, and assume `R` contains  $c : A$ , and  $\sigma$  contains no assignment for  $c$ . We have  $\mathcal{G} \ggg_S s.c : S''s(A) = \perp$ . Similarly, if  $\sigma$  contains the assignment  $c := B$ , then we have  $\mathcal{G} \ggg_S s.c : S''s(A) = B$ . In both cases `S''s(A)` is the result of translating `A` along the morphism `S''s`, prefixing all constants with  $s$ .

Now assume in addition a morphism `μ` from `R` to `T` and an assignment  $s := μ$  in a view or structure `m` from `S` to `T`. We have  $\mathcal{G} \ggg_m s.c := μ(c)$ , i.e., `m` maps the constant `s.c` of `S` to the result of applying `μ` to the constant `c` of `R`.

**Views and Structures** In order to define the above judgments, we need one auxiliary judgment that is defined by the three rules in Fig. 3.  $\mathcal{G} \ggg m : S \rightarrow T = \{\sigma\}$  expresses that `m` is a morphism from `S` to `T` defined by the list  $\sigma$  of assignments. `m` may be a view (first rule) or a structure (second rule). Finally, a structure `s` from `S` to `T` does not only induce constants `s.vec{c}` but also structures `s.vec{r}`. The meaning of the structure `T''s.r` is defined to be the composition `S''r T''s`. The judgment  $\mathcal{G} \ggg m : S \rightarrow T = \mu$  expresses that `m` is a morphism from `S` to `T` with definition `μ` (third rule).

$$\begin{array}{c}
 \frac{v : S \rightarrow T = \{\sigma\} \text{ in } \mathcal{G} \quad T = \{\dots, s : S = \{\sigma\}, \dots\} \text{ in } \mathcal{G}}{\mathcal{G} \ggg v : S \rightarrow T = \{\sigma\}} \quad \frac{}{\mathcal{G} \ggg T''s : S \rightarrow T = \{\sigma\}}
 \\[1em]
 \frac{\mathcal{G} \ggg T''s : S \rightarrow T = \_ \quad \mathcal{G} \ggg S''\vec{r} : R \rightarrow S = \_}{\mathcal{G} \ggg T''s.\vec{r} : R \rightarrow T = S''\vec{r} T''s}
 \end{array}$$

Figure 3. Views and Structures

**Semantics of Structures** We are now in the position to define the two main judgments. Fig. 4 gives the four rules defining the elaboration of structure occurring in a signature  $T$  (or: the lookup in a signature  $T$ ). The first two rules handle constants with and without definition declared in  $T$ . The other two rules handle constants  $s.\vec{c}$  induced by a structure  $s$  with domain  $S$ . If  $s$  provides an assignment  $B'$  for  $\vec{c}$  it is used as the definition of  $s.\vec{c}$  (third rule). Otherwise, the existing definition  $B$  of  $\vec{c}$  is translated along  $T''s$ . The translation along morphisms is defined formally below. In the interest of brevity, we put  $\mu(\perp) = \perp$ , i.e., neither  $\vec{c}$  has definition in  $S$  nor  $s$  provides an assignment for it, then  $s.\vec{c}$  has no definition either.

$$\begin{array}{c}
T = \{\dots, c : A = B, \dots\} \text{ in } \mathcal{G} \quad T = \{\dots, c : A, \dots\} \text{ in } \mathcal{G} \\
\mathcal{G} \ggg_T c : A = B \qquad \qquad \qquad \mathcal{G} \ggg_T c : A = \perp \\
\\
\mathcal{G} \ggg T''s : S \rightarrow T = \_ \quad \mathcal{G} \ggg_S \vec{c} : A = B \quad \mathcal{G} \ggg_{T''s} \vec{c} := B' \\
B' \neq \perp \\
\hline
\mathcal{G} \ggg_T s.\vec{c} : T''s(A) = B' \\
\\
\mathcal{G} \ggg T''s : S \rightarrow T = \_ \quad \mathcal{G} \ggg_S \vec{c} : A = B \quad \mathcal{G} \ggg_{T''s} \vec{c} := \perp \\
\mathcal{G} \ggg_T s.\vec{c} : T''s(A) = T''s(B)
\end{array}$$

Figure 4. Semantics of Structures

**Semantics of Assignments to Structures** Fig. 5 gives the three rules defining the elaboration of assignments to structures occurring in a morphism  $m$ . The first rule handles the case where  $m$  is defined by a morphism  $\mu$ , which is simply applied to all constants  $\vec{c}$  of  $S$ . If  $m$  is defined a list of assignments, two cases are possible. If  $m$  has an assignment for  $\vec{c}$ , the treatment is clear (second rule). But if  $m$  contains an assignment to a structure  $\vec{s}$ , it induces assignments to all constants  $\vec{s}.\vec{c}$  (third rule).

$$\begin{array}{c}
\mathcal{G} \ggg m : S \rightarrow T = \mu \quad \mathcal{G} \ggg m : S \rightarrow T = \{\dots, \vec{c} := C, \dots\} \\
\mathcal{G} \ggg_m \vec{c} := \mu(\vec{c}) \qquad \qquad \qquad \mathcal{G} \ggg_m \vec{c} := C \\
\\
\mathcal{G} \ggg m : S \rightarrow T = \{\dots, \vec{s} := \mu, \dots\} \quad \mathcal{G} \ggg S''\vec{s} : R \rightarrow S = \_ \\
\hline
\mathcal{G} \ggg_m \vec{s}.\vec{c} := \mu(\vec{c})
\end{array}$$

Figure 5. Semantics of Structure Assignments

**Morphism Application** Finally, we define the application of a morphism  $\mu$  to a term  $C$  by induction on  $\mu$  and  $C$ . We define the step cases by

$$\begin{array}{ll}
\mu \mu'(\vec{c}) & := \mu'(\mu(\vec{c})) \\
\mu(\text{type}) & := \text{type} \\
\mu(x) & := x \\
\mu([x : A] C) & := [x : \mu(A)] \mu(C) \\
\mu(\{x : A\} C) & := \{x : \mu(A)\} \mu(C) \\
\mu(C C') & := \mu(C) \mu(C')
\end{array}$$

The most interesting case is the base case when a view or a structure is applied to a constant: If  $\mathcal{G} \ggg m : S \rightarrow T = \_, \mathcal{G} \ggg_S \vec{c} := B$ , and  $\mathcal{G} \ggg_m \vec{c} := B'$ , then we define

$$m(\vec{c}) := \begin{cases} m(B) & \text{if } B \neq \perp \\ B' & \text{if } B = \perp, m = v \text{ view} \\ \vec{s}.\vec{c} & \text{if } B = \perp, m = T''\vec{s} \text{ structure} \end{cases}$$

|  |  |
|--|--|
| $\vdash \mathcal{G}$                       | $\mathcal{G}$ is a well-formed signature graph.  |
| $\mathcal{G} \triangleright D$             | The declaration $D$ can be added to $\mathcal{G}$ .  |
| $\mathcal{G} \triangleright_N D$           | The declaration or assignment $D$ can be added to the signature, view, or structure named $N$ .      |
| $\mathcal{G} \vdash \mu : S \rightarrow T$ | $\mu$ is a well-formed morphism from $S$ to $T$ (or: a well-formed structure over $T$ of type $S$ ). |
| $\mathcal{G} \vdash_T C \equiv B$          | $C$ and $B$ are equal over $\mathcal{G}$ and $T$ .   |
| $\mathcal{G} \vdash_T C : A$               | $C$ is a well-formed term of type $A$ over $\mathcal{G}$ and $T$ .                                   |

Figure 6. Main Judgments

If  $\vec{c}$  is defined then it is expanded (first case). If  $\vec{c}$  is declared then it is mapped according to the assignment provided by the view (second case) or mapped to the constant induced by the structure (third case). Clearly, this is only well-defined if the three judgments defined in this section are functional, i.e., there must be unique values  $S$ ,  $B$ , and  $B'$ . This follows from the requirement that there are no name clashes in  $\mathcal{G}$ , i.e. that constant symbols are not declared (in the case of a signature) or assigned (in the case of a view or structure) twice, as we show in the next section.

### 3.4 Type System

In this section, we present an inference system to define the **well-formedness** of our syntactic categories. The judgments are given in Fig. 6. The judgment  $\vdash \mathcal{G}$  states the well-formedness of signature graphs. The judgment  $\mathcal{G} \triangleright D$  expresses that  $\mathcal{G}$  can be extended with the signature or view  $D$ , and  $\mathcal{G} \triangleright_N D$  expresses that the signature, view, or structure  $N$  can be extended with the declaration or assignment, respectively,  $D$ . Finally, there are three judgments that define well-formed terms and morphisms relative to a signature graph and a signature declared in that graph.

While the conceptual core of the type system is quite simple, the technical details regarding namespace management can make the notation rather complex. Therefore, we choose a simplified presentation using a judgment  $\text{noClash}(\mathcal{G}, N, n)$  that expresses that a declaration or assignment for the identifier  $n$  can be added to the signature, view, or structure  $N$  without creating a name clash. Similarly,  $\text{noClash}(\mathcal{G}, N)$  expresses that a signature or view named  $N$  can be added to  $\mathcal{G}$ . We refrain from formalizing these judgments; instead, we only state that the consequent use of this judgment guarantees that the judgments  $\mathcal{G} \ggg m : S \rightarrow T = \_, \mathcal{G} \ggg_T \vec{c} : A = B$ , and  $\mathcal{G} \ggg_m \vec{c} := B$  are functional where the output arguments are indicated in red. Therefore, we can use these judgments to look up names, types, and definitions for the identifiers encountered during type-checking.

**Structure of Signature Graphs** We define  $\text{Sig}(\mathcal{G})$  to be the set of signature names declared in  $\mathcal{G}$ . The structure of signature graphs is defined by the rules in Fig. 7. These rules follow the grammar and iterate a well-formedness judgment over all components of a signature graph. They also check that views are total and that module names do not clash. The well-typedness of constant declarations and assignments (red assumptions) and objects (blue assumptions) is defined by the rules in Fig. 8.

The rule  $\mathcal{G}_\emptyset$  constructs an empty signature graph. The rules  $\text{Sig}$  and  $\text{View}$  extend a well-formed signature graph with a well-formed signature or view; while signatures can be added directly, views must be *total*, which means that they must provide an assignment for every declared constant.

Whether or not a signature or view is well-formed is defined in the remaining rules. The rules  $\text{Sig}_\emptyset$  and  $\text{Sym}$  construct signatures by successively adding well-formed symbols, and the rules  $\text{View}_\emptyset$  and  $\text{View}_{\text{Ass}}$  construct views by successively adding well-formed assignments. The rules for structures correspond to those for views:

$$\begin{array}{c}
\frac{}{\vdash \cdot \quad \mathcal{G}_\emptyset} \\
\frac{\vdash \mathcal{G} \quad \mathcal{G} \triangleright T = \{\Sigma\}}{\vdash \mathcal{G}, T = \{\Sigma\}} \text{ Sig} \qquad \frac{\text{noClash}(\mathcal{G}, T)}{\mathcal{G} \triangleright T = \{\cdot\}} \text{ Sig}_\emptyset \qquad \frac{\mathcal{G} \triangleright T = \{\Sigma\} \quad \mathcal{G}, T = \{\Sigma\} \triangleright_T D}{\mathcal{G} \triangleright T = \{\Sigma, D\}} \text{ Sym} \\
\\
\frac{\vdash \mathcal{G} \quad \mathcal{G} \triangleright v : S \rightarrow T = \{\sigma\} \quad \mathcal{G} \ggg_v \vec{c} := B \text{ for some } B \text{ whenever } \mathcal{G} \ggg_S \vec{c} : A = \perp}{\vdash \mathcal{G}, v : S \rightarrow T = \{\sigma\}} \text{ View} \\
\\
\frac{\text{noClash}(\mathcal{G}, v) \quad S \in \text{Sig}(\mathcal{G}) \quad T \in \text{Sig}(\mathcal{G})}{\mathcal{G} \triangleright v : S \rightarrow T = \{\cdot\}} \text{ View}_\emptyset \qquad \frac{\mathcal{G} \triangleright v : S \rightarrow T = \{\sigma\} \quad \mathcal{G}, v : S \rightarrow T = \{\sigma\} \triangleright_v D}{\mathcal{G} \triangleright v : S \rightarrow T = \{\sigma, D\}} \text{ View}_{Ass} \\
\\
\frac{\text{noClash}(\mathcal{G}, T, s) \quad S \in \text{Sig}(\mathcal{G}) \setminus \{T\}}{\mathcal{G} \triangleright_T s : S = \{\cdot\}} \text{ Str}_\emptyset \qquad \frac{\mathcal{G}, T = \{\Sigma\} \triangleright s : S = \{\sigma\} \quad \mathcal{G}, T = \{\Sigma, s : S = \{\sigma\}\} \triangleright_{T''s} D}{\mathcal{G}, T = \{\Sigma\} \triangleright_T s : S = \{\sigma, D\}} \text{ Str}_{Ass}
\end{array}$$

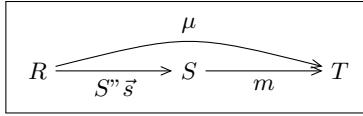
Figure 7. Structural Rules

$\text{Str}_\emptyset$  and  $\text{Str}_{Ass}$  construct structures by successively adding well-formed assignments. The rule  $\text{Str}_\emptyset$  also ensures that a signature may not instantiate itself.

**Well-formed Declarations and Assignments** The rules above the dotted line in Fig. 8 define when constants and assignments are well-typed. The rule  $\text{Con}$  says that constant declarations  $c : A = B$  are well-typed for a signature  $T$  if  $B$  has type  $A$ , and if  $c$  is not already declared in  $T$ . In order to save case distinctions, we use the following convention: We permit the case  $B = \perp$  for constants without definitions, and say that the typing judgment  $\mathcal{G} \vdash_T \perp : A$  holds if  $A$  is a well-formed type or kind. Note that extensions to other type systems only require to modify this convention appropriately.

The rule  $\text{ConAss}$  defines when an assignment  $\vec{c} := B$  is well-typed. The first three premises look up the domain and codomain of the last view or structure  $m$  in  $\mathcal{G}$ , make sure that an assignment for  $\vec{c}$  does not clash with existing assignments in  $m$ , and look up the type of  $\vec{c}$ . The definition of  $\vec{c}$  must be  $\perp$ , i.e., defined constants cannot be instantiated. The final premise type-checks  $B$  against the translation of  $A$ . If  $m(A)$  is not defined, which is possible if  $m$  is a view and  $A$  contains constants for which  $m$  does not provide an assignment yet, we consider the typing judgment not to hold. Thus, the order of assignments in a link must respect the dependency order between the symbols declared in the domain.

The rule  $\text{StrAss}$  for assignments to structures is similar to  $\text{ConAss}$ . The first three premises



correspond to those of  $\text{ConAss}$ . In particular,  $R$  corresponds to  $A$  as the type of  $\vec{s}$ , and the fourth premise checks the type of  $\mu$  against  $R$ . To understand the last premise, note that the intended semantics of assignments to structures is that the diagram given in Sect. 3.4 commutes. This is only possible if  $\mu$  agrees with  $S''\vec{s}m$  for all constants for which  $m$  is already determined.

The rules below the dotted line in Fig. 8 define the typing of objects.  $\mathcal{T}_\cong$  and  $\mathcal{T}_\equiv$  replace the core LF rule for the lookup of constants in the signature (called  $\text{con}$  in [Pfenning 2001]). All other typing and equality rules of core LF are retained. To obtain module systems for other type theories, the typing and equality rules have to be changed accordingly. Finally, the rules  $\mathcal{M}_m$  and  $\mathcal{M}_{comp}$  construct morphisms as sequences of views and structures. Composition is written in diagrammatic order, i.e., from the domain to the codomain.

### 3.5 Meta-Theory

We turn now to the meta-theoretical results that we have shown about the module system, most notably, conservativity.

First, we define two morphisms to be equal if they agree for all constants of the domain signature.

**DEFINITION 7.** Assume  $\mathcal{G} \vdash \mu : S \rightarrow T$  and  $\mathcal{G} \vdash \mu' : S \rightarrow T$ . We define the judgment  $\mathcal{G} \vdash \mu \equiv \mu'$  to hold iff for all  $\vec{c}$  for which  $\mathcal{G} \ggg_S \vec{c} : \_ = \_$  we have  $\mathcal{G} \vdash \mu(\vec{c}) \equiv \mu'(\vec{c})$ .

**THEOREM 8.** Assume  $\mathcal{G} \vdash \mu : S \rightarrow T$ . If  $\mathcal{G} \vdash_S C : A$ , then  $\mathcal{G} \vdash_T \mu(C) : \mu(A)$ . And if  $\mathcal{G} \vdash_S C \equiv C'$  and  $\mathcal{G} \vdash \mu \equiv \mu'$ , then  $\mathcal{G} \vdash_T \mu(C) \equiv \mu'(C')$ .

**Proof:** This is a special case of the results given in [Rabe 2008, Ch. 6]. The proof proceeds by induction on  $\mu$  and  $C$ . The key steps are that the rules  $\text{ConAss}$  and  $\text{StrAss}$  permit to add assignments to views or structures only if they do not violate the typing or equality relations of the domain signature.  $\square$

This result establishes the basic properties of signature morphisms as encoded in our module system: the preservation of typing and equality. The following result states the intended semantics of assignments to structures, namely that the diagram given in Sect. 3.4 commutes. Together with Thm. 8, it shows that the module system can be used to reason about signature morphisms. This is the cornerstone of adequacy proofs because adequacy proofs in terms of signature morphisms are very elegant and concise. For example, we obtain the adequacy of the structure sharing in the LF encoding of intuitionistic logic in Example 4 immediately without having to elaborate any of the involved structures.

**THEOREM 9.** Assume  $\vdash \mathcal{G}$ . If there is an assignment  $\vec{s} := \mu$  in a view or structure  $m$  from  $S$  to  $T$  in  $\mathcal{G}$ , then  $\mathcal{G} \vdash S''\vec{s}m \equiv \mu$ .

**Proof:** This is a special case of the results given in [Rabe 2008, Ch. 6]. All definitions are targeted at this result, most importantly rule  $\text{StrAss}$  rejects any assignment that would violate the theorem.  $\square$

Finally, we show that modular LF is conservative over core LF. The main argument is that elaboration is sound and that core LF signatures elaborate to themselves. The only caveat is easily explained: Qualified identifiers in modular LF need to be considered constants in core LF, which means that “” and “.” must be allowed in constant names.

**THEOREM 10.** Assume a signature graph  $\mathcal{G}$ . Let  $\Sigma$  be the core LF signature containing the declarations

$$\begin{array}{c}
\frac{\text{noClash}(\mathcal{G}, T, c) \quad \mathcal{G} \vdash_T B : A}{\mathcal{G} \triangleright_T c : A = B} \text{Con} \\
\frac{\text{noClash}(\mathcal{G}, m, \vec{c}) \quad \mathcal{G} \ggg m : S \rightarrow T =_- \quad \mathcal{G} \vdash_T B : m(A) \quad \mathcal{G} \ggg_S \vec{c} : A =_-}{\mathcal{G} \triangleright_m \vec{c} := B} \text{ConAss} \\
\frac{\text{noClash}(\mathcal{G}, m, \vec{s}) \quad \mathcal{G} \ggg m : S \rightarrow T =_- \quad \mathcal{G} \vdash \mu : R \rightarrow T \quad \text{whenever } \mathcal{G} \ggg_S \vec{s} \cdot \vec{c} : _- = B, B \neq \perp}{\mathcal{G} \triangleright_m \vec{s} := \mu} \text{StrAss} \\
\cdots \\
\frac{\mathcal{G} \ggg_T \vec{c} : A =_-}{\mathcal{G} \vdash_T \vec{c} : A} \mathcal{T}_! \quad \frac{\mathcal{G} \ggg_T \vec{c} : _- = B, B \neq \perp}{\mathcal{G} \vdash_T \vec{c} \equiv B} \mathcal{T}_\equiv \\
\frac{\mathcal{G} \ggg m : S \rightarrow T =_-}{\mathcal{G} \vdash m : S \rightarrow T} \mathcal{M}_m \quad \frac{\mathcal{G} \vdash \mu : R \rightarrow S \quad \mathcal{G} \vdash \mu' : S \rightarrow T}{\mathcal{G} \vdash \mu \mu' : R \rightarrow T} \mathcal{M}_{comp}
\end{array}$$

**Figure 8.** Typing Rules

- for all signatures  $T$  declared in  $\mathcal{G}$ : whenever  $\mathcal{G} \ggg_T \vec{c} : A = B$  for  $B \neq \perp$ , the declaration  $T''\vec{c} : A = B$ ; and whenever  $\mathcal{G} \ggg_T \vec{c} : A = \perp$ , the declaration  $T''\vec{c} : A$ ,
- for all views  $v$  with domain  $S$  declared in  $\mathcal{G}$ : whenever  $\mathcal{G} \ggg_v \vec{c} : A = \perp$  and  $\mathcal{G} \ggg_v \vec{c} := B$ , the declaration  $v''\vec{c} : v(A) = B$ ,
- for all structures  $s$  with domain  $S$  declared in a signature  $T$  of  $\mathcal{G}$ : whenever  $\mathcal{G} \ggg_S \vec{c} : _- = B$ ,  $B \neq \perp$ , and  $\mathcal{G} \ggg_{T''s} \vec{c} := B'$ ,  $B' \neq \perp$ , then a declaration that is well-typed iff  $B \equiv B'$ ,<sup>1</sup>

in some order that respects the dependencies between them. Then  $\vdash \mathcal{G}$  iff  $\Sigma$  is a valid core LF signature.

**Proof:** This is a special case of the results given in [Rabe 2008, Ch. 6]. The only modification of the argument is that here  $\Sigma$  is always ill-formed if any view in  $\mathcal{G}$  is not total because it will contain the illegal symbol  $\perp$ .  $\square$

## 4. Implementation

The module system for LF discussed in this paper has been implemented as part of the Twelf system. More information about the implementation and some examples can be found on the project webpage at <http://www.twelf.org/mod/>.

Our implementation of modular Twelf builds upon the code base of the original Twelf. Twelf's own highly modular design aided our effort to introduce namespace management and the treatment of qualified identifiers. Its built-in mechanism for notational definitions is heavily used in the implementation of elaboration in particular for dealiasing. Parts of the central data structures had to be changed to maintain qualified identifiers, and we frequently use hash tables instead of arrays. The additional convenience of the module system does not come at the expense of performance, as we can show next. The experimental results are reported in Figure 9. The experiments compare the running times of various mechanisms inside the two implementations of Twelf when loading a Twelf signature (that does not contain any modules). First, the cut-elimination proof for intuitionistic and classical logic [Pfenning

|                | Cut-Elim      | TALT          | SML           |
|----------------|---------------|---------------|---------------|
| Parsing        | 0.008 (0.008) | 3.590 (2.733) | 0.583 (0.851) |
| Reconstruction | 0.017 (0.017) | 8.620 (14.00) | 1.688 (2.324) |
| Abstraction    | 0.008 (0.007) | 7.154 (6.254) | 0.738 (1.004) |
| Modes          | 0.002 (0.002) | 2.130 (3.129) | 0.193 (0.477) |
| Subordination  | 0.004 (0.002) | 18.39 (10.87) | 5.851 (4.392) |
| Termination    | 0.009 (0.010) | 1.157 (0.698) | 0.273 (0.213) |
| Compilation    | 0.001 (0.001) | 0.077 (0.078) | 0.044 (0.045) |
| Solving        | 0.000 (0.000) | 0.838 (0.498) | 0.000 (0.000) |
| Coverage       | 0.225 (0.270) | 2173 (2176)   | 8.003 (7.190) |
| Worlds         | 0.002 (0.002) | 2.810 (1.241) | 2.124 (1.922) |
| Total          | 0.275 (0.319) | 2218 (2216)   | 19.49 (18.42) |

**Figure 9.** Experimental Data. Modular Twelf (Traditional Twelf) in seconds.

1995], the formalization of the meta-theory of typed assembly language [Crary 2003] (which consists of about 2500 meta-theorems), and the formalization of the meta-theory of the intermediate language of full Standard ML [Lee et al. 2007] (which consists of about 1300 meta-theorems). All timings are measured in seconds and rounded. The experiments were conducted on a Dell Poweredge 1950 equipped with two dual-core Xeon 5140 2.33GHz processors and 8GB RAM.

## 5. Conclusion

We have described a module system for the logical framework LF that is both expressive and elegant. Besides signatures, it introduces signature morphisms, in form of structures and views.

We believe that our examples have shown that named structures and views provide for elegant representations of inheritance relations and translations. In particular, views can themselves be composed modularly. We give another example in [Horozal and Rabe 2009], where we represent a soundness proof for first-order logic by representing proof and model theory as signatures. We define a view from the proof to the model theory. Both proof and model theory as well as the view are developed separately for each connective and quantifier and then plugged together as in the view KOLM.

<sup>1</sup> Such a declaration always exists. For example, if  $B$  and  $B'$  are types, the definition of the identity as a function from  $B$  to  $B'$  is only well-typed if  $B \equiv B'$ . This observation is due to Dan Licata.

Views are custom-tailored toward structural translations that translate constant symbols into compound terms. They are, however, too limited to capture the essence of non-structural translations that are defined by cases and appear frequently, especially in the study of the meta theory of deductive systems. We realize the importance of this observation, but leave further investigations to future work.

In summary, the module system described in this paper is conservative over LF because each signature, structure, or view can be fully elaborated into core LF. It is practical, because it is available to users of the Twelf system and we have shown that it does not degrade runtime performance.

**Acknowledgments** Our module system is a special case of a more generic system that the first author developed with Michael Kohlhase. Design and implementation of the version described here benefited greatly from discussions with Frank Pfenning and prior work by Kevin Watkins.

## References

- S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an Evolutionary Formal Software-Development Using CASL. In D. Bert, C. Choppy, and P. Mosses, editors, *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1999.
- H. Barendregt. An introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.
- J. Chrzaszcz. Implementing modules in the coq system. In D. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics (TPHOLs 03)*, pages 270–286. Springer Verlag, LNCS 2758, 2003.
- CoFI (The Common Framework Initiative). *CASL Reference Manual*, volume 2900 (IFIP Series) of *LNCS*. Springer, 2004.
- K. Crary. Toward a foundational typed assembly language. In G. Morrisett, editor, *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 38, No. 1, pages 198–212, New Orleans, Louisiana, Jan. 2003. ACM Press.
- W. Farmer, J. Guttman, and F. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11(2):213–248, 1993.
- F. Haftmann and M. Wenzel. Constructive type classes in isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs, TYPES 2006*, pages 149–165. Springer Verlag LNCS 4502, 2007.
- R. Harper and F. Pfenning. A module system for a programming language based on the LF logical framework. *Journal of Logic and Computation*, 8(1):5–31, 1998.
- R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, Jan. 1993.
- R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.
- F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. In "Logical and Semantic Frameworks, with Applications", 2009. To appear.
- F. Kammlüller, M. Wenzel, and L. C. Paulson. Locales: A sectioning concept for isabelle. In *Theorem Proving in Higher Order Logics (TPHOLs 99)*, LNCS 1690, pages 149–165. Springer, 1999.
- D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of standard ML. In *Proceedings of the 34th Annual Symposium on Principles of Programming Languages*, pages 173–184, New York, NY, USA, 2007. ACM Press. ISBN 1-59593-575-4. doi: <http://doi.acm.org/10.1145/1190216.1190245>.
- D. Licata, R. Simmons, and D. Lee. A simple module system for Twelf. <http://www.cs.cmu.edu/~dr1/pubs.html>, 2006.
- U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.
- F. Pfenning. Structural cut elimination. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 156–166, San Diego, California, June 1995. IEEE Computer Society Press.
- F. Pfenning. Logical frameworks. In *Handbook of automated reasoning*, pages 1063–1147. Elsevier, 2001.
- F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.
- F. Rabe. *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen, 2008.
- D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Control*, 76:165–210, 1988.
- C. Schürmann and M. O. Stehr. An executable formalization of the HOL/Nuprl connection in the meta-logical framework Twelf. In *Proceedings of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 150–166, Phnom Penh, Cambodia, 2006. Springer Verlag.