The Future of Logic: Foundation-Independence

Florian Rabe

Jacobs University Bremen

Abstract

Throughout the 20th century, the automation of formal logics in computers has created unprecedented potential for practical applications of logic – most importantly the mechanical verification of mathematics and software. But the high cost of these applications makes them infeasible but for a few flagship projects, and even those are negligible compared to the ever-rising needs for verification. We hold that the biggest challenge in the future of logic will be to enable applications at much larger scales and simultaneously at much lower costs.

This will require a far more efficient allocation of resources. Wherever possible, theoretical and practical results must be formulated generically so that they can be instantiated to arbitrary logics; this will allow reusing results in the face of today's multitude of diverging logical systems. Moreover, the software engineering problems concerning automation support must be decoupled from the theoretical problems of designing logics and calculi; this will allow researchers outside or at the fringe of logic to contribute scalable logic-independent tools.

Anticipating these needs, the author has developed the MMT framework. It offers a modern approach towards defining, implementing, and applying logics that focuses on modular design and logic-independent results. This paper summarizes the ideas behind and the results about MMT. It focuses on showing how MMT provides a theoretical and practical framework for the future of logic.

1 Introduction and Related Work

Motivation While logic has historically focused on the theoretical study of a few individual logics – mostly first-order logic and some others such as higher-order or modal logic – recent decades have seen increasing specialization into a plethora of different logics. Moreover, during that time advances in technology – e.g., the internet and theorem provers – have dramatically changed the scale of practical applications of logic. For the future, modern logicians envision the routine use of logic for the verification of mathematical theorems and safety-critical software.

However, these approaches 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. For example, successful projects such as the verification of the Kepler conjecture [Hal05] or the L4 microkernel [KAE+10] required double-digit person years of investment.

These scales bring a new set of critical challenges for logics such as building large libraries of logical theorems collaboratively, using multiple logics in the same project and reusing theorems across logics, and the interoperability of logic-based tools. Many of these challenges were not anticipated in the designs of current logics, tools, and libraries:

- Modern logic tools such as ACL2 [KMM00], Coq [Coq14], HOL [Gor88, HOL, Har96, NPW02], Matita [ACTZ06], Mizar [TB85], Nuprl [CAB+86], or PVS [ORS92] are built on **mutually incompatible logical foundations**. These include first-order logic, higher-order logic, set theory, and numerous variants of type theory.
- The respective communities are mostly disjoint and have built **overlapping but mutually incompatible libraries**.
- Each of these tools **lacks urgently-needed features** because no community can afford developing (and maintaining) all desirable features. These include both core logical features such as theorem proving and peripheral features like managing large libraries.
- Novel logics usually have to start from scratch because it is almost impossible to reuse existing tools and libraries. That massively **slows down evolution** because it can take years to evaluate a new idea.
- All but a few logics are **never used beyond toy examples** because it takes dozens of person-years to reach production-ready maturity.

Existing Approaches These problems have been known for several decades (see, e.g., [Ano94]) and have motivated **three major independent developments in different, disjoint communities**:

(1) **Logical frameworks** [Pfe01] are meta-logics in which logics can be defined. Their key benefit is that results about the meta-logic can be inherited by the logics defined in them. Examples include type inference (Twelf tool [PS99] for LF [HHP93]), rewriting (Dedukti tool [BCH12] for LF modulo [CD07]), and theorem proving (Isabelle tool [Pau94] for higher-order logic [Chu40]).

But logical frameworks **do not go far enough**: Like logics, the various meta-logics use mutually incompatible foundations and tools. Moreover, they are not expressive enough for defining the logics of modern logics (such as the ones cited above). This requires the design of more and more complex meta-logics, partially defeating the purpose for which they were introduced.

(2) Categorical frameworks like **institutions** [GB92] give an abstract, uniform definition of what a logic is. Their key benefit is that they capture precisely the central concepts of logics (such as theories and models) and leverage category theory to elegantly formalize reuse across theories and logics. Many results that were originally logic-specific have been generalized to the institution-independent level [Dia08], and institution-level tool support has been developed [MML07].

But institutions **go too far**: They abstract from the syntactic structure of sentences, theories, and proofs that is essential to logic. Thus, they cannot provide substantial logic-independent tool support, instead requiring specific tools for every logic.

(3) **Markup languages for mathematics** such as MathML [ABC+03], OpenMath [BCC+04], and OMDoc [Koh06] allow the representation of syntax trees for logical objects (e.g., theories or formulas). Their key benefit is that they provide a standardized machine-readable interchange format that can be used for any logic. A major achievement is the inclusion of MathML into the definition of HTML5.

But markup languages succeeded only because they **ignore the logical semantics**. They allow the representation of proofs and models but they do not make any attempt to define which proof or models are correct.

Foundation-Independence We introduce the novel paradigm of *foundation-independence* – the systematic abstraction from logic-specific conceptualizations, theorems, and algorithms in order to obtain results that apply to any logic. This yields the flexibility to design new logics on demand and instantiate existing results to apply them right away.

Contrary to logical frameworks, we do not fix any foundation, not even a meta-logic. Contrary to institutions, the reusable results include strong tool support. And contrary to markup languages, the logical semantics is formalized and respected by the tools.

Central to foundation-independence are our MMT language [RK13, Rab14b] and tool [Rab13]. They provide the theoretical and practical environment to define concepts, reason meta-logically, and implement tool support in a foundation-independent way. We can see MMT as the **final step in a progression towards more abstract formalisms**:

Mathematics	Logic	Logical Framework	Foundation-Independence
			MMT
		meta-logic	meta-logic
	logic	logic	logic
domain knowledge	domain knowledge	domain knowledge	domain knowledge

In conventional mathematics, domain knowledge was expressed directly in mathematical notation. Starting with Frege's work, logic provided a formal syntax and semantics for this notation. Starting in the 1970s, logical frameworks provided meta-logics to formally define this syntax and semantics. Now MMT formalizes the foundation-independent level.

One might expect that this meta-meta-level, at which MMT works, is too abstract to develop deep results. **But not only is it possible to generalize many existing results to the foundation-independent level, MMT-based solutions are also often simpler and stronger than foundation-specific ones.** Moreover, MMT is very well-suited for modularity and system integration and thus better prepared for the large scale challenges of the future than any foundation-specific system. In particular, it systematically separates concerns between logic designers, tool developers, and application developers.

We argue this point by surveying the author's work of the past 10 years. This includes numerous foundation-independent results, which we summarize in a coherent setting:

- Section 2: the MMT language, which provides the foundation-independent representation of any logical object,
- Section 3: the foundation-independent definition of logic-related concepts based on the MMT language,
- Section 4: the foundation-independent algorithms implemented in the MMT system,
- Section 5: the scalable knowledge management support developed on top of the MMT system.

In all cases, we pay special attention to the foundation-independent nature of the results and discuss the differences from and benefits over foundation-specific approaches.

2 Foundation-Independent Representation

Key Concepts The MMT language uses a small set of carefully chosen orthogonal primitive concepts: *theories*, *symbols*, and *objects*, which are related by *typing* and *equality* and acted on by theory *morphisms*. The main insight behind foundation-independence is that these concepts are at the same time

- universal in the sense that they occur in virtually all formal systems in essentially the same way,
- complete in the sense that they allow subsuming virtually all logics and related languages.

MMT **theories** subsume any kind of formal system such as logical frameworks, mathematical foundations (e.g., ZF set theory), type theories, logics, domain theories, ontology languages, ontologies, specification languages, specifications, programming languages, etc.

Every theory consists of a list of **symbol declarations** $c[: t][= d][\#N]$ where c is the symbol identifier, the objects t and d are its type and definiens, and N is its notation. Symbol declarations subsume any kind of basic declaration common in formal systems such as type/constant/function/predicate symbols, binders, type operators, concepts, relations, axioms, theorems, inference rules, derived rules, etc. In particular, theorems are just a special case of typed symbols: They can be represented via the propositions-as-types correspondence [CF58, How80] as declarations $c : F = p$, which establish theorem F via proof p.

MMT **objects** subsume any kind of complex expressions common in formal systems such as terms, values, types, literals, individuals, universes, formulas, proofs, derivations, etc. The key property of theories is that they define the scope of identifiers: An object o over a theory T may use only the symbols visible to T.

Objects o and o' over a theory T are subject to the **typing and equality judgments** $\vdash_T o : o'$ and $\vdash_T o \equiv o'$. These judgments subsume any kind of classification used to define the semantics of formal systems: typing subsumes, e.g., well-formedness, typing, kinding, sorting, instantiation, satisfaction, and proving; equality subsumes, e.g., equality, interpretation, and computation. The central step to represent a specific foundation is (i) to give a theory F that declares one symbol for each primitive operator of the foundation, and (ii) to fix the rules for typing and equality judgments on F-objects.

The above concepts are related by **theory morphisms**, and theories and morphism form a category. Morphisms subsume all structural relations between theories including logic translations, denotational semantics, interpretation functions, imports, instantiation, inheritance, functors, implementations, and models.

For theories S and T, a morphism $m : S \to T$ maps every S-symbol to a T-object, which induces a homomorphic translation $m(-)$ of all S-objects to T-objects. MMT guarantees that $m(-)$ preserves the MMT judgments, i.e., $\vdash_S o : o'$ implies $\vdash_T m(o) : m(o')$ and accordingly for equality. This includes the preservation of provability as a special case, which allows moving theorems between theories along morphisms.

$$\begin{tikzcd}
 & & \mathtt{LF} \arrow[r, hook] \arrow[dl, dotted, hook] \arrow[d, dotted, hook] & \mathtt{LF}^{\leq} \\
 & \mathtt{FOL} \arrow[r, "\mathtt{Trans}"] \arrow[dl, dotted, hook] \arrow[dr, dotted, hook] & \mathtt{HOL} \arrow[r, "\mathtt{Sem}"] & \mathtt{ZF} \\
\mathtt{Monoid} \arrow[rr, bend left, "\mathtt{mult}"] \arrow[rr, bend right, "\mathtt{add}"'] & & \mathtt{Ring} \arrow[ur, "\mathtt{Int}"'] &
\end{tikzcd}$$

For example, consider the representation of various theories in diagram on the right. Here dotted morphisms represent the meta-theory relation where one language is imported to define another; `Trans` is a language translations; `Sem` is the semantics of higher-order logic in ZF set theory. `add` and `mult` are the two instantiations of `Monoid` used to build the theory `Ring` modularly. `Int` represents the integers as a model of `Ring`, and the compositions `Int` $\circ$ `add` and `Int` $\circ$ `mult` are the additive and multiplicative monoid of the integers. Finally, $\mathtt{LF}^{\leq}$ is the extension of `LF` with subtyping that we can use to represent more complex logics.

Note how theories can be moved along morphisms via pushout: All developments over `LF` can be moved $\mathtt{LF}^{\leq}$ so that we do not lose any work when migrating to a new meta-logic. Similarly, `Monoid` and `Ring` can be moved to `HOL` by pushout along `Trans` or to `ZF` along $\mathtt{Sem} \circ \mathtt{Trans}$.

Dia	$::=$	$(Thy \mid Mor)^*$	diagram
Thy	$::=$	$c = \{Dec, \ldots, Dec\}$	theory declaration
Mor	$::=$	$c : o \to o = \{Ass, \ldots, Ass\}$	morphism declaration
Dec	$::=$	$c[: o][= o][\#N]$	symbol declaration
Ass	$::=$	$c := o$	assignment to symbol
o	$::=$	$c \mid x \mid \mathcal{L}^c(Str) \mid \mathcal{C}^c((x[: o])^*; o^*)$	object
N	$::=$	$(\mathtt{A}_{Int} \mid \mathtt{V}_{Int} \mid Str)^*$	notation
c	$::=$	URIs	identifiers
Str	$::=$	Unicode strings	literals or delimiters
Int	$::=$	integers	argument positions

Formal Language The above grammar contains all the essentials of the Mmt language. A **diagram** is a list of theory and morphism declarations. The declaration of a **theory** t is of the form $t = \{\ldots, c[: o][= o'][\#N], \ldots\}$. The declaration of a **morphism** m from S to T is of the form $m = \{\ldots, c := o, \ldots\}$ such that every S-symbol c is mapped to a T-object o.

Mmt uses only 4 productions for T-**objects**. 3 of these yields the basic leafs of syntax trees: symbol identifiers c that are visible to T, previously bound variables x, and literals $\mathcal{L}^c(s)$ of type c with string representation s. The remaining production $\mathcal{C}^c(x_1[: o_1], \ldots, x_m[: o_m]; a_1, \ldots, a_n)$ subsumes the various ways of forming complex objects: It applies a symbol c, binding some variables x_i, to some arguments a_j. For example, $\mathcal{C}^{\mathtt{and}}(; F, G)$ represents the conjunction of F and G, and $\mathcal{C}^{\mathtt{forall}}(x; F)$ represents the universal quantification over x in F. We obtain the usual notations $F \wedge G$ and $\forall x.F$ if we declare the symbols with notations such as $\mathtt{and}\,\#\mathtt{A}_1 \wedge \mathtt{A}_2$ and $\mathtt{forall}\,\#\forall \mathtt{V}_1.\mathtt{A}_2$.

Specific Foundations Individual foundations arise as fragments of Mmt: A foundation singles out the well-formed theories and objects. We refer to [Rab14b] for the details and only sketch the key idea. Mmt provides a minimal inference system for the typing and equality judgments that fixes only the foundation-independent rules, e.g.,

- the well-formedness of theories relative to the well-formed of objects,
- the typing rules for the leafs of the syntax tree, e.g., we have $\vdash_T c : o$ whenever $c : o$ is declared in T,
- the equality rules for equivalence, congruence, and α-equality.

Further rules are added by the foundations. For example, we can declare a symbol `o` for the type of well-formed formulas and add a typing-rule for `and` that derives $\vdash_T \mathcal{C}^{\mathtt{and}}(; F, G) : \mathtt{o}$ from $\vdash_T F : \mathtt{o}$ and $\vdash_T G : \mathtt{o}$.

Crucially, the foundations only restrict attention to fragments of Mmt and do not change the Mmt grammar. Therefore, many definitions can be stated foundation-independently once and for all by induction on the Mmt grammar, whereas foundation-specific approaches have to repeat these tediously, often less elegantly. Examples include the category of theories and universal constructions such as pushout, the homomorphic extension $m(-)$ and the proof that it preserves the judgments, free/bound variables and substitution, the use of notations to relate concrete and abstract syntax, the management of change through differencing and patching, and the modular development of large theories. As an example, we consider the module system.

Module System The foundation-independent module system of MMT allows forming complex theories and morphisms in two ways.

(1) *Object-based formation* uses symbols to form objects that denote complex theories or morphisms. For example, we declare a unary symbol `identity` $\#$`id`($\mathtt{A}_1$) and add inference rules that make `id`(T) is the identity morphism $T \to T$. Similarly, we declare a symbol `comp` $\# \mathtt{A}_1 \circ \mathtt{A}_2$ that maps two morphisms to their composition. [Rab15b] discusses further constructions, most importantly pushouts.

(2) *Declaration-based formation* uses special declarations that elaborate into sets of symbol declarations. Most importantly, the declaration **include** T elaborates into the set of all declarations of T. And the declaration $q : T = \{\ldots, c := o, \ldots\}$ declares an instance T where T-symbols are qualified by q and some T-symbols c are substituted with objects o. The details can be found in [RK13].

This yields a very simple and expressive syntax for forming large theories from small components, where the semantics of theory formation is independent of the chosen foundation.

3 Foundation-Independent Logic

In MMT, we can give concise foundation-independent definitions of the central concepts regarding logics. This has been described in detail in [Rab14b], and we will only give examples here.

Meta-Logic Meta-logics are represented as foundations in MMT. For example, the theory for LF is shown on the right. It introduces one symbol for each primitive concept along with notations for it. For example, the abstract syntax for a λ-abstraction is $\mathcal{C}^{\mathtt{lambda}}(x : A; t)$, and the concrete syntax is $[x : A]t$.

Additionally, LF declares one symbol for each rule that is added to MMT's inference system. These are the usual rules for a dependently-typed λ-calculus such as

$$\frac{\Gamma, x : A \vdash_\Sigma t : B}{\Gamma \vdash_\Sigma [x : A]t : \{x : A\}B}\mathtt{infer}_{\mathtt{lambda}}$$

LF = {

`kind`		
`type`	:	`kind`
`Pi`	#	{ $\mathtt{V}_1$ } $\mathtt{A}_2$
`lambda`	#	[$\mathtt{V}_1$] $\mathtt{A}_2$
`apply`	#	$\mathtt{A}_1$ $\mathtt{A}_2$
`arrow`	#	$\mathtt{A}_1 \to \mathtt{A}_2$
$\mathtt{infer}_{\mathtt{Pi}}$		
$\mathtt{infer}_{\mathtt{lambda}}$		
$\mathtt{infer}_{\mathtt{apply}}$		
`funcExten`		
`beta`		

}

Of course, these rules cannot themselves by declared formally.[1] Nonetheless, they are declared as symbols of the MMT theory LF. Thus, e.g., $\mathtt{infer}_{\mathtt{lambda}}$ can only be used to prove $\vdash_T o : o'$ if T imports LF.

[1]The simplest reasonable meta-logic in which we can formalize these rules is LF itself. So we have to supply these rules extra-linguistically to get off the ground. Once a sufficiently expressive meta-logic is represented in this way, we never have to add inference rules again.

Syntax and Proof Theory For the special case of using LF as a meta-logic, a logic **syntax** is any MMT theory that includes LF and the declarations $\mathtt{o} : \mathtt{type}$ and $\mathtt{ded} : \mathtt{o} \to \mathtt{type}$. For example, the theory PL for propositional logic adds the declarations on the left and the theory FOL for first-order logic the ones on the right (where we omit the usual notations):

$$\begin{array}{lll@{\qquad\qquad}lll} \top & : & \mathtt{o} & & & \\ \bot & : & \mathtt{o} & \mathtt{i} & : & \mathtt{type} \\ \neg & : & \mathtt{o} \to \mathtt{o} & \doteq & : & \mathtt{i} \to \mathtt{i} \to \mathtt{o} \\ \wedge & : & \mathtt{o} \to \mathtt{o} \to \mathtt{o} & \forall & : & (\mathtt{i} \to \mathtt{o}) \to \mathtt{o} \\ \vee & : & \mathtt{o} \to \mathtt{o} \to \mathtt{o} & \exists & : & (\mathtt{i} \to \mathtt{o}) \to \mathtt{o} \\ \Rightarrow & : & \mathtt{o} \to \mathtt{o} \to \mathtt{o} & & & \end{array}$$

Given a logic syntax L, an L-**theory** is a theory that extends L. For example, the FOL-theory of semi-groups adds $\mathtt{comp} : i \to i \to i \# \mathtt{A}_1 \circ \mathtt{A}_2$ and $\mathtt{assoc} : \mathtt{ded}\big(\forall[x : \mathtt{i}]\forall[y : \mathtt{i}]\forall[z : \mathtt{i}](x \circ y) \circ z \doteq x \circ (y \circ z)\big)$.

A **sentence** over an L-theory T is any object F such that $\vdash_T F : \mathtt{o}$.

The **proof theory** of a logic is defined as a theory that extends the syntax. For example, some proof rules for a natural deduction calculus for propositional logic are declared as:

$$\begin{array}{lll} \vee I_l & : & \{A : \mathtt{o}\}\{B : \mathtt{o}\}\mathtt{ded}\, A \to \mathtt{ded}\,[A \vee B] \\ \vee I_r & : & \{A : \mathtt{o}\}\{B : \mathtt{o}\}\mathtt{ded}\, B \to \mathtt{ded}\,[A \vee B] \\ \vee E & : & \{A : \mathtt{o}\}\{B : \mathtt{o}\}\{C : \mathtt{o}\}\mathtt{ded}\,[A \vee B] \to \\ & & \quad (\mathtt{ded}\, A \to \mathtt{ded}\, C) \to (\mathtt{ded}\, B \to \mathtt{ded}\, C) \to \mathtt{ded}\, C \end{array}$$

Finally, a **proof** of the T-sentence F under the assumptions $F_1, \ldots, F_n$ is any object such that $\vdash_T p : \mathtt{ded}\, F_1 \to \ldots \to \mathtt{ded}\, F_n \to \mathtt{ded}\, F$.

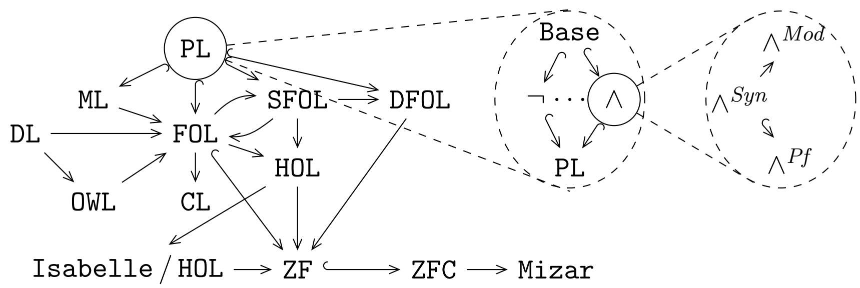
Model Theory and Logic Translations There is an intuitive similarity between model theory and logic translations: Both are inductive translations from one formalism into another. In particular, model theory translates the syntax and proofs into the semantic domain. In MMT level, we can capture this similarity formally: Both are represented as theory morphisms.

In particular, we represent the semantic domain, in which models are defined, as a theory itself. Traditional logic assumed an implicit mathematical domain such as axiomatic set theory. But modern approaches in computer science often use other domains such as constructive type theory or programming languages. Therefore, it is very advantageous that MMT allows choosing different domains. Comprehensive examples of foundations and model theory are given in [IR11, HR11].

For example, the set theoretical model theory of `HOL` is represented as a morphism $\mathtt{Sem} : \mathtt{HOL} \to \mathtt{ZF}$, where `ZF` formalizes axiomatic set theory. `Sem` maps every logical symbol to its interpretation – this corresponds to the cases in the inductive definition of the interpretation function. The different ways of extending `Sem` to the non-logical symbols declared in `FOL`-theories T correspond to the different possible T-models. Thus, models are represented as theory morphisms as well.

Finally we can show once and for all and for any meta-logic that every logic defined in MMT induces an institution. This ties together the concrete representations in logical frameworks and the abstract representations as institutions. We obtain an according result for logic translations, which are represented as theory morphisms $L \to L'$ such as from `FOL` to `HOL`.

The LATIN project [CHK+11] leveraged the MMT module system and the above conceptualization to systematically develop a comprehensive library of logics from small reusable components. The whole library inlcudes > 1000 moduels, a small fragment is visualized below. The left part relates, e.g., propositional, first-order, higher-order, modal, and description logics as well as set theories via imports and translation. The middle part shows how propositional logic PL is build up from individual features for, e.g., negation and conjunction. The right part shows how the module for conjunction consists of formalizations of syntax, model theory, and proof theory.



4 Foundation-Independent Algorithms

The MMT system provides foundation-independent implementations of the typical algorithms needed to obtain tool support for a logic. These are obtained by systematically differentiating between the *foundation-independent* and the *foundation-specific* aspects of existing solutions, and implementing the former in MMT.

Each implementation is relative to a set of rules via which the foundation-specific aspects are supplied. This yields extremely general algorithms that can be easily instantiated for specific foundations. The details of each rule are implemented directly in MMT's underlying programming language (Scala). Practical experience has shown this to be not only feasible but extremely simple and elegant. For example, the foundation-independent parts of MMT consist of $> 30,000$ lines of Scala code. But, e.g., the LF plugin provides only 10 simple rules of a few hundred lines of Scala code. Of course, for logics defined using a meta-logic like LF, no rules have to be provided at all.

Moreover, rules are declared as regular MMT symbols (as we did for the LF in Sect. 3), and the implementations use exactly those rules that are imported into the respective context. That makes it very easy to recombine and extend rules to implement new languages. For example, it took a single rule of 10 lines of code to add shallow polymorphism to LF, and a single rule schema to add rewriting. Similar extensions of foundation-specific systems can require whole PhD theses if they are possible at all.

Below we sketch MMT's solutions for parsing and type-checking as examples and discuss applications to deduction and computation.

Parsing and Presenting The MMT algorithms for parsing and presenting are almost completely foundation-independent. (Rules are only needed for literals.) They use the notations available to T to parse/present T-objects. Notably, because the grammar for MMT objects is so simple, they are easier to design and implement than their foundation-specific counterparts.

Therefore, the foundation-independent MMT algorithms are actually stronger than those of many state-of-the-art logic tools.

As seen in Sect. 3, MMT notations subsume not only the usual fixity-based notations but also complex notations like the ones of LF binders. But MMT also supports additional advanced features, each only present in very few foundation-specific systems:

- Sequence arguments and sequences of bound variables. For LF, we can actually use notations such as $[\mathtt{V}_1, \ldots]\mathtt{A}_2$ and $(\mathtt{A}_1 * \ldots) \to \mathtt{A}_2$. The former expresses that `lambda` binds a comma-separated list of variables; the latter expresses that `arrow` takes a star-separated list of arguments.
- Implicit arguments. Notations may declare some arguments to be implicit, i.e., omitted in concrete syntax and inferred from the context.
- 2-dimensionality. Notations can declare over-/under-/sub-/superscripts as well as fraction-style notations. These are used for presentation in 2-dimensional output formats such as HTML.

Typing Another advantage of MMT's grammar for objects is that it can express both human-written and machine-checked syntax trees in the same format. For example, human-written syntax often omits certain objects (e.g., implicit arguments and variable types), which must be inferred during type-checking.

The MMT type-checker takes an MMT judgment such as $\Delta \vdash_U o : o'$ where Δ declares some variables for omitted subobjects. Then it solves for the unique substitution σ such that $\vdash_U o[\sigma] : o'[\sigma]$. This problem is undecidable for most foundations, and at the moment only a few foundation-specific solutions (e.g., the one in Coq [Coq14]) are substantially stronger than MMT's foundation-independent one.

As an example, we consider the subproblem of type inference, where T and o are given and o' such that $\vdash_T o : o'$ is needed. If o is an identifier c, MMT returns the type of c in T (if any); this is foundation-independent. But if o is a λ-abstraction $\mathcal{C}^{\mathtt{lambda}}(x : a; t)$, MMT delegates to a foundation-specific rule. For example, if T imports LF, this is the rule $\mathtt{infer}_{\mathtt{lambda}}$, which is implemented in the LF plugin as the following Scala function:

```
object InferLambda extends TypeInferenceRule(LF.lambda) {
 def apply(solver:Solver,o:Object,context:Context):Option[Object]={
  o match {
   case LF.lambda(x,a,t) =>
    solver.check(Typing(context, a, LF.type))
    solver.inferType(t, context ++ (x oftype a)).map{
     b => LF.Pi(x,a,b)
    }
   case _ => None
  }
 }
}
```

Here `TypeInferenceRule` is the interface for rules that may be used during type inference. Its constructor argument `LF.lambda` indicates that it is applicable to `lambda`-objects. When applied, it receives the object o and the current context (a pair T and the list of bound variables) and returns the inferred type if possible. It also receives the current solution environment `solver`, which maintains the unknown subobjects Δ and offers callbacks for recursively checking the assumptions of the rule.

Note how foundation-independence yields a clear separation of concerns. The foundation-specific core – e.g., the rule $\texttt{infer}_{\texttt{lambda}}$ – is supplied by the developer of the LF plugin. But all the difficult aspects of type-checking – e.g., constraint propagation and error reporting – are handled foundation-independently by MMT. Similarly, the module system remains transparent: The rules are not aware of the modular structure of T.

MMT implements objects as an inductive data type with four constructors for constants, variables, literals, and complex objects. One might expect that this foundation-independent representation of objects makes it awkward to implement foundation-specific rules. For example, for LF, one would prefer an inductive data type with one constructor for each LF symbol.

MMT makes this possible: It uses the notations to generate constructor/destructor abbreviations. Thus, e.g., LF-rules can be defined as if there were a foundation-specific inductive data type for LF-objects. In particular, for LF, MMT generates abbreviations `LF.Pi` and `LF.lambda`. Above, `LF.Pi(x,a,b)` expands into $\mathcal{C}^{\texttt{Pi}}(x : a; b)$. Notably, these abbreviations are also available during pattern-matching: Above, `case LF.lambda(x,a,t)` matches any MMT object of the form $\mathcal{C}^{\texttt{lambda}}(x : a; t)$.

Deduction and Computation The semantics of logics and related languages can be defined either deductively or computationally. Both paradigms are well-suited for foundation-independent implementation. Moreover, the integration of typing, deduction, and computation is an open research question. And because MMT can state these problems generically, it provides a good environment for developing solutions.

In MMT, deduction means to find some object p such that $\vdash_T p : F$, in which case p proves F. Computation means to find some simple object o' such that $\vdash_T o \equiv o'$, in which case o evaluates to o'. MMT provides very promising basic foundation-independent implementations for both.

MMT's foundation-independent theorem prover implements, e.g., the search space, structure sharing, and backtracking. Similarly, its foundation-independent computation engine implements the congruence closure and rewriting. In both cases, the individual proving/-computation steps are delegated to rules imported by T. Both algorithms are transparently integrated with typing.

Notably, the LF plugin defines only three deduction rules and already yields a simple theorem prover for any logic defined in LF. Computation rules for specific theories can be provided by giving models whose semantic domain is a programming language [Rab15a]. That allows integrating arbitrary computation with logics.

However, contrary to parsing and type-checking, deduction and computation often require foundation-specific optimizations such as search strategies and decision procedures. Here MMT's foundation-independent implementations are still rather basic. But there is no indication they will not be improved dramatically in future work.

5 Foundation-Independent Knowledge Management

Developing knowledge management services and applications is usually too expensive for individual logic communities, especially if it requires optimization for large scale use cases. Indeed, even the logics with the strongest tool support fare badly on knowledge management.

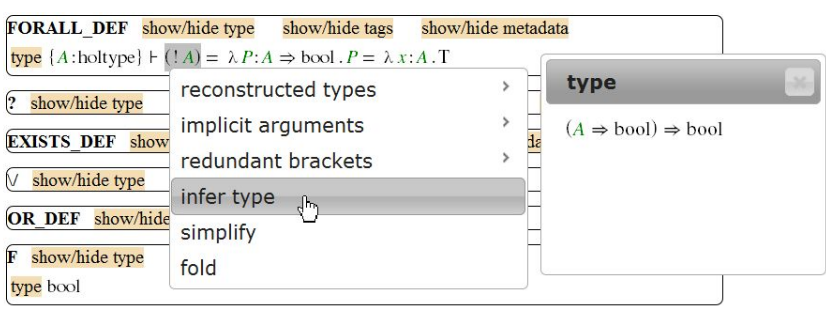
Fortunately, this support can be obtained completely foundation-independently – we do not even have to supply any foundation-specific rules as we did in Sect. 4. Thus, maybe surprisingly, foundation-independence helps solve problems in general at relative ease that have proved very hard in each foundation-specific instance.

User Interfaces MMT provides two foundation-independent user interfaces that go beyond the state-of-the-art of all but very few foundation-specific solutions. It uses jEdit, a mature full-fledged text editor, as the host system for a foundation-independent IDE. And it uses web browsers as the host system for a library browsing environment.

The user interfaces are described in detail in [Rab14a], and we only list some of the advanced features. Both include hyperlinking of symbol to their declaration and tooltips that show the type of the selected subobject. The IDE displays the list of errors in and the abstract syntax tree of a source file. Both are cross-referenced with the respective source locations. Moreover, it provides context-sensitive auto-completion, which uses the available proving rules to suggest operators that can return an object of the required type. This already yields a basic interactive theorem prover. The HTML5 interface includes 2-dimensional presentations, e.g., for proof trees, and the folding and hiding of subobjects. It also allows dynamically displaying additional information such as SVG graphs of the modular structure of theories.

Moreover, both interfaces are highly scalable. For example, the IDE employs change management to recheck a declaration only when it was changed or affected by a change. And the multi-threaded HTTP server loads and unloads theories into main memory dynamically so that it can serve very large libraries.

Services The simplicity of the MMT syntax makes it easy to develop advanced foundation-independent services. We mention only some examples here. [Rab12] defines a **query language** that allows retrieving sets of declarations based on relational (RDF-style) and tree-based (XQuery-style) criteria. Queries may refer both to the MMT concepts and to user-annotated metadata. MathWebSearch [KŞ06] is a massively optimized **substitution tree index** of objects. It performs unification-based search over extremely large libraries almost-instantaneously. [IR12] develops **change management** support for MMT. It creates differences and patches that only include those changes in an MMT file that are semantically relevant.



All services are independent and exposed through high-level interfaces so that they can be easily reused when building MMT-based applications such as the user interfaces mentioned above. Of particular importance is the MathHub system [IJKW14]. Based on git and MMT, it uses the above services in a highly scalable project management and archiving solution for logic libraries.

The viability and strength of these approaches has been demonstrated by instantiating the above services for several major logics such as for Mizar in [IKRU13] and for HOL Light in [KR14]. For example, the screenshot above shows the result of dynamic inference in a web browser: Browsing the HOL Light library, the user selected the subexpression $!A$ (universal quantification at type A) and called type inference, which returned $(A \Rightarrow \text{bool}) \Rightarrow \text{bool}$. No major foundation-specific system is designed in a way that would make it easy to offer such an interactive behavior in a web browser.

6 Conclusion

The success of logic in the future depends on the solution of one major problem: the proliferation of different logics with incompatible foundations and imperfect and expensive (if any) tool support. These logics and tools are competing instead of collaborating, thus creating massive duplication of work and unexploited synergies. Moreover, new logics are designed much faster than tool support can be developed, e.g., in the area of modal logic. This inefficient allocation of resources must be overcome for scaling up applications of logic in the future.

Over several decades, three mostly disjoint research communities in logic have independently recognized this problem, and each has developed a major solution: logical frameworks, institutions, and markup languages. All three solutions can be seen as steps towards foundation-independence, where conceptualizations, theorems, and tool support are obtained uniformly for an arbitrary logic.

But these solutions have been developed separately, and each can only solve some aspects of the problem. Logical frameworks are too restrictive and ignore model theoretical aspects. Institutions are too abstract and lack proof theoretical tool support. And markup languages do not formalize the semantics of logics.

The present author has picked up these ideas and coherently re-invented them in a novel framework: the foundation-independent MMT language and system. We have described the existing results, which show that MMT allows obtaining simple and powerful results that apply to any logic. Notably, these results range from the theoretical definitions to large scale implementations. Within MMT, new logics can be defined extremely easily, and mature, scalable implementations can be obtained at extremely low cost.

The vast majority of logic-related problems can be studied within MMT – in extremely general settings and with tight connections to both theoretical foundations and practical applications. The MMT language is very simple and extensible, and the MMT system is open-source, well-documented, and systematically designed to be extensible. Thus, it provides a powerful universal framework for the future of logic.

References

[ABC+03] R. Ausbrooks, S. Buswell, D. Carlisle, S. Dalmas, S. Devitt, A. Diaz, M. Froumentin, R. Hunter, P. Ion, M. Kohlhase, R. Miner, N. Poppelier, 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.

[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.

[Ano94] Anonymous. The QED Manifesto. In A. Bundy, editor, *Automated Deduction*, pages 238–251. Springer, 1994.

[BCC+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.

[BCH12] M. Boespflug, Q. Carbonneaux, and O. Hermant. The $\lambda\Pi$-calculus modulo as a universal proof language. In D. Pichardie and T. Weber, editors, *Proceedings of PxTP2012: Proof Exchange for Theorem Proving*, pages 28–43, 2012.

[CAB+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.

[CD07] D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In S. Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications*, pages 102–117. Springer, 2007.

[CF58] H. Curry and R. Feys. *Combinatory Logic.* North-Holland, Amsterdam, 1958.

[CHK+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.

[Chu40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.

[Coq14] Coq Development Team. The Coq Proof Assistant: Reference Manual. Technical report, INRIA, 2014.

[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.

[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.

[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.

[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.

[HOL] HOL4 development team. `http://hol.sourceforge.net/`.

[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.

[IJKW14] M. Iancu, C. Jucovschi, M. Kohlhase, and T. Wiesing. System Description: MathHub.info. In S. Watt, J. Davenport, A. Sexton, P. Sojka, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 431–434. Springer, 2014.

[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+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.

[KMM00] M. Kaufmann, P. Manolios, and J Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[Koh06] M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*. Number 4180 in Lecture Notes in Artificial Intelligence. Springer, 2006.

[KR14] C. Kaliszyk and F. Rabe. Towards Knowledge Management for HOL Light. In S. Watt, J. Davenport, A. Sexton, P. Sojka, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 357–372. Springer, 2014.

[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.

[MML07] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *Tools and Algorithms for the Construction and Analysis of Systems 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.

[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.

[Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[Pfe01] F. Pfenning. Logical frameworks. In J. Robinson and A. Voronkov, editors, *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. In H. Ganzinger, editor, *Automated Deduction*, pages 202–206, 1999.

[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.

[Rab13] F. Rabe. The MMT API: A Generic MKM System. In J. Carette, D. Aspinall, C. Lange, P. Sojka, and W. Windsteiger, editors, *Intelligent Computer Mathematics*, pages 339–343. Springer, 2013.

[Rab14a] F. Rabe. A Logic-Independent IDE. In C. Benzmller and B. Woltzenlogel Paleo, editors, *Workshop on User Interfaces for Theorem Provers*, 2014.

[Rab14b] F. Rabe. How to Identify, Translate, and Combine Logics? *Journal of Logic and Computation*, 2014. doi:10.1093/logcom/exu079.

[Rab15a] F. Rabe. Generic Literals. under review, see `http://kwarc.info/frabe/Research/rabe_literals_14.pdf`, 2015.

[Rab15b] F. Rabe. Theory Expressions (A Survey). under review, see `http://kwarc.info/frabe/Research/rabe_theoexp_15.pdf`, 2015.

[RK13] F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, 230(1):1–54, 2013.

[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. Morgan Kaufmann, 1985.