# A Framework for Implementing Logical Frameworks

Florian Rabe
Jacobs University, Bremen, Germany

## ABSTRACT

MMT implements a prototypical declarative language in a way that systematically abstracts from theoretical and practical aspects of individual logics. It tries to develop as many solutions as possible generically so that logic designers can focus on the essentials and inherit a large scale implementation at low cost.

For example, advanced generic features include module system, user interface, and change management. Here we focus on MMT's generic type reconstruction component and show how it can be customized to obtain implementations of various logics. As examples, we consider LF as well as its extensions with shallow polymorphism and rewriting.

## 1. INTRODUCTION AND RELATED WORK

### *Generic Large Scale Features.*

At small scales, designing and implementing a new logic is relatively simple: We only need a grammar, an inference system, and the corresponding data structures and algorithm.

However, to be practical, we have to supplement this with a number of advanced features: a human-friendly concrete syntax with user-defined notations, a type reconstruction algorithm that infers omitted subterms, a module system that enables reuse, a smart user interface, and a theorem prover that discharges proof obligations. These features have two things in common: They are essential for practical applications, and they require orders of magnitude more work than the initial implementation.

This impedes evolution. Problems in the initial design may become apparent only in large case studies, which can only be carried out after a major investment into advanced features. Moreover, existing advanced implementations can often not be reused for new languages: As these implementations evolve and acquire users, they tend to become locked to a certain language so that adding a new language feature often requires a fork or a reimplementation.

MMT derives from the hypothesis that (i) the basic features are highly logic-specific, but (ii) many advanced features can be realized generically. This motivates a separation of concerns where small scale definitions of logics are combined with generic implementations.

The MMT meta-language [RK13] focuses on modeling the structural properties shared by the vast majority of languages. And the MMT system [Rab13] provides a rapid prototyping functionality for logics. Moreover, the MMT system can realize many advanced features generically such as module system [RK13] and user interface [Rab14a].

In this paper, we present the practical aspects of one further generic advanced feature: the type reconstruction algorithm employed by MMT.

### *Type Reconstruction.*

By "type reconstruction", we mean the problem of inferring unknown subexpressions that were omitted in a given term. Specifically, we will infer implicit arguments and the types of bound variables (in both of which we include the type parameters of polymorphic constants). These unknown subexpressions can be formally represented as meta-variables that are existentially quantified on the outside of the expression. Then, instead of checking a typing judgment $t : A$, we have to prove $\exists \vec{X}.t(\vec{X}) : A(\vec{X})$, and the witnesses found for the $X_i$ are the solutions for the unknown subexpressions.

This problem is particularly difficult in expressive type theories where unification is undecidable. Therefore, implementations such as Coq [Coq14], Matita [ACTZ06], Agda [Nor05], or Twelf [PS99] employ sophisticated algorithms to obtain practical solutions.

These algorithms are so complex that they are often only understood by a few people close to the original implementors. A formal description or documentation may be incomplete or lacking altogether. [Nor05] describes the algorithm of Agda for a fragment of the language. [Lut01] describes an algorithm for the calculus of constructions. Only recently two major formalizations were presented: [ARCT12] for the algorithm implemented in Matita (there called *refinement*), and [Pie13] for the algorithm implemented in Beluga [PD10] (which is similar to the one of Twelf).

The basic idea of these algorithms is to apply bidirectional type checking to $t(\vec{X}) : A(\vec{X})$. This leads to equality constraints about the $X_i$, most importantly when checking the equality between an inferred type and an expected type. Eventually it produces constraints of the form $X_i = t_i$, which are used to solve $X_i$ as $t_i$.

Our approach follows this recipe but deviates crucially from the above by not fixing the type theory. Instead, our

implementation assumes arbitrary sets of operators/binders, notations, and typing rules. This open-world assumption affects the design in interesting ways, making some aspects more and others less complex. For example, in MMT it becomes very natural to use meta-variables both for terms and for types. This is in contrast with, e.g., Twelf [PS99] and [Pie13], which carefully avoid the use of type variables.

Our main result is a generic implementation that can be customized easily by supplying plugins for specific type theories.

We evaluate our system in 3 examples. Firstly, we instantiate our work with dependent type theory (as in LF [HHP93]). This requires only the straightforward implementation of 9 simple rules, for which the complexity of the reconstruction remains hidden. Secondly, we extend LF with shallow polymorphism by adding only a single rule. Finally, we obtain an instance of LF modulo a user-declared rewrite system; this is inspired by Dedukti [BCH12], which implements the $\lambda\Pi$-modulo calculus [CD07].

### *Logical Frameworks.*

Our results are similar to the ones obtained in logical frameworks like LF [HHP93] or Isabelle [Pau94]. These fix a syntax and a type system for the framework language $F$ and represent the operators, notations, and typing rules of $L$ in terms of their counterparts of $F$. In this sense, the reconstruction algorithms for logical frameworks are generic algorithms similar to ours.

We deviate partially from the logical framework approach. We still represent the operators and notations of $L$ as MMT theories. But we do not fix a type system for MMT. Instead, we assume that the type system for $L$ is provided by a plugin that supplies the code for individual inference rules.

This makes it harder to supply inference rules in practice: Rules can no longer be specified declaratively but must be programmed. But this also gives us more freedom: The rules of $L$ can use case distinctions, side conditions, or arbitrary computations; they can be generated dynamically; and they can provide customized error messages. In any case, our implementation minimizes the programming cost by offering simple plugin interfaces.

In particular, MMT can serve as a meta-framework in which logical frameworks are developed. Indeed, our examples easily implement powerful extensions of LF.

### *Overview.*

We present the MMT language and system in Sect. 2 and 3. Then we describe the type reconstruction algorithm and our examples in Sect. 4 and 5. In Sect. 6, we describe how type reconstruction interacts with other generic advanced features of MMT, namely module system, parsing, and graphical user interface. In Sect. 7, we revisit some related work and conclude.

The implementation of the MMT system are available at `https://svn.kwarc.info/repos/MMT/doc/html/index.html`.

All examples given throughout the paper are reconstructed successfully.

## 2. THE MMT LANGUAGE

In this section, we fix the syntax of MMT. The grammar is given in Fig. 1. This is essentially a fragment of the grammar we used in [RK13] except that notations are novel.

| | | | |
|---|---|---|---|
| Theory | $\Sigma$ | ::= | $\cdot \mid \Sigma,\ c[: E][= E][\#N]$ |
| Context | $\Gamma$ | ::= | $\cdot \mid \Gamma,\ x : E$ |
| Term | $E$ | ::= | $c \mid x \mid c(\Gamma; E^*)$ |
| Notation | $N$ | ::= | $(\mathcal{V}_n \mid \mathcal{A}_n \mid \texttt{string})^*$ |

**Figure 1: MMT Grammar**

| | | |
|---|---|---|
| `type` | # | `type` |
| `kind` | # | `kind` |
| `Pi` | # | $\{\,\mathcal{V}_1\,\}\,\mathcal{A}_2$ |
| `lambda` | # | $[\,\mathcal{V}_1\,]\,\mathcal{A}_2$ |
| `apply` | # | $\mathcal{A}_1\ \ \mathcal{A}_2$ |
| `arrow` | # | $\mathcal{A}_1 \to \mathcal{A}_2$ |

**Figure 2: Syntax of LF in MMT**

A **theory** $\Sigma$ is a list of **constant** declarations. A **constant** declaration is of the form $c[: A][= t][\#N]$ where $c$ is an identifier, $A$ is its **type**, $t$ its **definiens**, and $N$ its **notation**, all of which are optional. A **context** $\Gamma$ is very similar to a theory and declares typed variables.

Type and definiens are **terms**, which are formed from constants $c$, variables $x$, and complex terms $c(\Gamma; E_1, \ldots, E_n)$.

Our complex terms are a very general construct that subsumes binding and application: $c$ is called the **head** of the term, $\Gamma$ declares the **bound variables**, and the $E_i$ are the **arguments**.

The variables declared in $\Gamma$ are bound in the arguments $E_i$, and the usual definition of **substitution** can be generalized easily. If $E$ is a term using the free variables $x_1, \ldots, x_l$ and $\gamma = t_1, \ldots, t_l$ is a list of terms, we write $E[\gamma]$ for the result of substituting each $x_i$ with $t_i$.

In a complex term, both bound variables and arguments are optional. For example, $\Gamma = \cdot$ yields the case of $n$-ary non-binding operators; and if the numbers of bound variables and arguments are both 1, we obtain the usual binding operators such as $\lambda$ or $\forall$. In particular, MMT does not fix the set of binding operators, which yields very high level of generality.

If a constant $c$ has a notation $N$, then $N$ is used for the concrete representation of complex terms with head $c$: $\mathcal{V}_n$ refers to the $n$-th bound variable, $\mathcal{A}_n$ to the $n$-th argument, and these are interspersed with arbitrary delimiters. Arguments that are not mentioned by the notation are **implicit** and inferred by reconstruction.

| | | |
|---|---|---|
| `tp` | $: \texttt{type}$ | $\#\ \texttt{tp}$ |
| `tm` | $: \texttt{tp} \to \texttt{type}$ | $\#\ \texttt{tm}\,\mathcal{A}_1$ |
| `Sigma` | $: \{A\}(\texttt{tm}\,A \to \texttt{tp}) \to \texttt{tp}$ | $\#\ \Sigma\,\mathcal{A}_2$ |
| `pair` | $: \{A, B\}\{a : \texttt{tm}\,A\}\texttt{tm}\,(B\,a) \to \texttt{tm}\,(\Sigma[x]B\,x)$ | $\#\ (\mathcal{A}_3, \mathcal{A}_4)$ |
| `pi1` | $: \{A, B\}\texttt{tm}\,(\Sigma[x : \texttt{tm}\,A]B\,x) \to \texttt{tm}\,A$ | $\#\ \texttt{pi1}\,\mathcal{A}_3$ |
| `pi2` | $: \{A, B\}\{t : \texttt{tm}\,(\Sigma[x : \texttt{tm}\,A]B\,x)\}\texttt{tm}\,(B\,(\texttt{pi1}\,t)$ | $\#\ \texttt{pi1}\,\mathcal{A}_3$ |

**Figure 3: Syntax of Sigma Types in MMT/LF**

The syntax of MMT is generic in the sense that MMT has no built-in constants $c$. Therefore, to form any terms at all, we first have to declare some constants for the primitive operators of the desired logic:

*Example* 2.1 (LF as an MMT Theory). To obtain the syntax of the logical framework LF, we use the theory shown in

Fig. 2. It declares one constant for each primitive concept of LF and a notation for it.

For example, in a $\lambda$-abstraction $\texttt{lambda}(x : A; t)$, `lambda` is the head, $x : A$ the single variable binding, and $t$ the single argument. The notation declares $[x : A]t$ as its concrete representation. Similarly, in an application $\texttt{apply}(\cdot; f, t)$, `apply` is the head, no variables are bound, and $f$ and $t$ are the arguments. The concrete representation is $f\ t$.

Then we can extend such a theory with the usual typed declarations:

*Example* 2.2 (Sigma Types as an MMT/LF Theory). Fig. 3 gives a Church-style encoding of dependent product types in LF, which we revisit in Ex. 5.3.

Here we already use implicit arguments and omitted variable types to be reconstructed later.

| | |
|---|---|
| $\vdash \Sigma$ | $\Sigma$ is a valid theory |
| $\vdash_\Sigma \Gamma$ | $\Gamma$ is a valid $\Sigma$-context |
| $\Gamma \vdash_\Sigma t : A$ | $t$ has type $A$ |
| $\Gamma \vdash_\Sigma E = E'$ | $E$ and $E'$ are equal |
| $\Gamma \vdash_\Sigma \_ : A$ | $A$ is inhabitable |
| $\Gamma \vdash_\Sigma U\ \texttt{UNIV}$ | $\Gamma \vdash_\Sigma \_ : A$ whenever $\Gamma \vdash_\Sigma A : U$ |

**Figure 4: Judgments**

MMT uses the **judgments** given in Fig. 4. The rules for theories and contexts are given in Fig. 5. Theories and contexts are valid if all declarations are. If a type is provided, it must be inhabitable; if a definiens is provided, it must be typed by the type. Variables in contexts may shadow other variables or constants, but constant names must be unique within the theory. The corresponding lookup rules are straightforward.

The rules for theories and context are generic in the sense that they are fixed by MMT and are the same for every language represented in MMT. The judgments for terms, on the other hand, can be defined (almost) arbitrarily.

More concretely, we define: An MMT **language** is any definition of the judgments for typing, equality, universe, and inhabitability that is closed under the rules given in Fig. 6. These rules specify the behavior of equality: binding admits $\alpha$-conversion, and equality is an equivalence and congruence relation.

*Example* 2.3 (LF as an MMT Language). We extend Ex. 2.1 with the following rules where $U$ ranges over $\{\texttt{type}, \texttt{kind}\}$

$$\frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma \texttt{type} : \texttt{kind}}\texttt{type} \qquad \frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma U\ \texttt{UNIV}}\texttt{UNIV}$$

The latter uses the universe judgments, which is an abbreviation to easily make many terms inhabitable: It says that any term typed by `type` or `kind` is inhabitable, i.e., LF-types and LF-kinds may appear as the types of MMT constants and variables.

Moreover, we add the well-known typing rules for dependent function types

$$\frac{\Gamma \vdash_\Sigma A : \texttt{type} \qquad \Gamma, x : A \vdash_\Sigma B : U}{\Gamma \vdash_\Sigma \{x : A\}B : U}\texttt{Pi}$$

$$\frac{\Gamma \vdash_\Sigma \{x : A\}B : U \qquad \Gamma, x : A \vdash_\Sigma t : B}{\Gamma \vdash_\Sigma [x : A]t : \{x : A\}B}\texttt{lambda}$$

valid theories:

$$\frac{}{\vdash \cdot} \qquad \frac{\vdash \Sigma \qquad [\cdot \vdash_\Sigma \_ : A] \qquad [\cdot \vdash_\Sigma t : A] \qquad c \notin \Sigma}{\vdash \Sigma,\ c[: A][= t]}$$

valid contexts:

$$\frac{\vdash \Sigma}{\vdash_\Sigma \cdot} \qquad \frac{\vdash_\Sigma \Gamma \qquad \Gamma \vdash_\Sigma \_ : A}{\vdash_\Sigma \Gamma,\ c : A}$$

lookup in theories:

$$\frac{\vdash \Sigma \qquad c : A \text{ in } \Sigma}{\vdash_\Sigma c : A} \qquad \frac{\vdash \Sigma \qquad c = t \text{ in } \Sigma}{\vdash_\Sigma c = t}$$

lookup in contexts:

$$\frac{\vdash_\Sigma \Gamma \qquad x : A \text{ in } \Gamma, \text{ rightmost if multiple}}{\Gamma \vdash_\Sigma x : A}$$

**Figure 5: Rules for Theories and Contexts**

$$\frac{\Gamma \vdash_\Sigma f : \{x : A\}B \qquad \Gamma \vdash_\Sigma t : A}{\Gamma \vdash_\Sigma f\ t : B[t]}\texttt{apply}$$

which correspond to the rules $(*, *)$ and $(*, \Box)$ of pure type systems. Note that these ensure that only typed but not kinded variables may be bound by `lambda` and `Pi`.

In addition to the equality rules from Fig. 6, we use the rules for $\beta$ and $\eta$-equality.

### *Assumptions MMT Makes.*

A completely generic framework would be useless. Instead, it is important to ask exactly which assumptions a framework makes. We discuss the most interesting ones.

The lookup rules imply that weakening and exchange are admissible rules, i.e., MMT cannot represent substructural languages. Because MMT focuses on representing theories, a substructural version of MMT would be extremely weak (and we suspect hardly useful).

The lookup of the definiens implies that constants are always equal to their definiens. Some systems reject this rule in order to avoid inefficient definition expansion steps. However, because MMT does not automatically normalize terms, the presence of this rule only legalizes definition expansion and does not enforce it.

The congruence rule for typing implies subject reduction. This assumption is usually reasonable, but it is typically treated as an admissible. It is necessary to make sure that MMT can perform equality reasoning in any term.

For the same reason, the congruence rule for complex terms permits equality conversion under a binder. Specialized to `lambda`, this is called the $\xi$-rule. $\xi$ is sometimes rejected, but we argue that its nature as a congruence rule justifies assuming it. (Recall that in the presence of $\beta$, $\eta$ and $\xi$ together are equivalent to extensionality. We hold that $\eta$ should be rejected if extensionality is to be avoided.)

### *Assumptions MMT Does Not Make.*

MMT terms subsume all types, kinds, universes, propositions, proofs, etc. None of these are distinguished by MMT.

Consequently, typing and equality can be arbitrary binary relations between terms, and inhabitability can be any unary

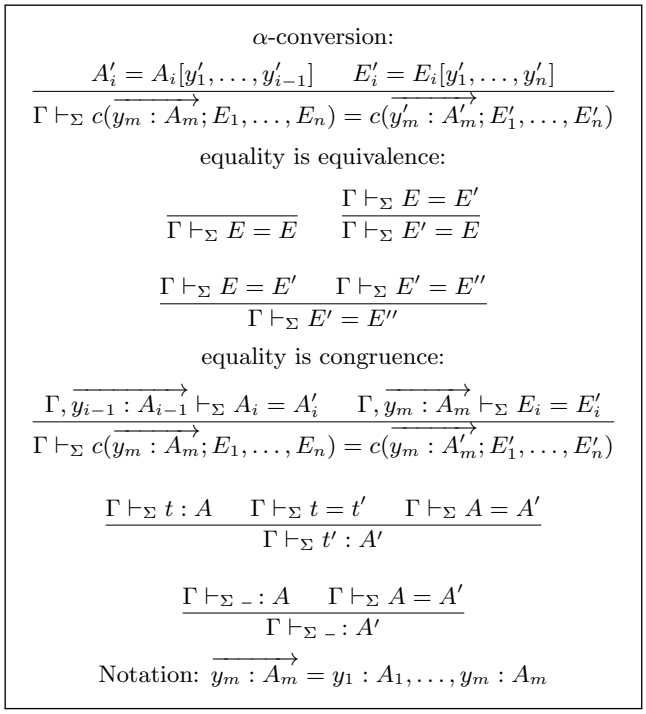


**Figure 6: Rules for Equality**

property of terms (as long as they admit the rules from Fig. 6.

MMT does not restrict how terms are formed – theories can declare any constant and use it as an operator or binder of any arity.

It is always possible to add constants without type or definiens (as in Ex. 2.1). However, as no lookup rules apply, $c$ can only be used validly by an MMT language that postulates judgments about $c$ (as we do in Ex. 2.3).

We do not assume that canonical forms exists. In fact, the judgments may even be undecidable, and MMT languages may perform arbitrary theorem proving or computation to define the judgments.

MMT hardly restricts what declarations can occur in theories or contexts. Instead, languages are free to define the inhabitability judgment. Declarations of inductive and record types, however, are not directly possible. This is intentional because we will provide for them in an orthogonal way in the future.

# 3. THE MMT SYSTEM

The kernel of the MMT system implements the data structures for theories, contexts, terms, and judgments. Moreover, it provides the infrastructure for authoring, analyzing, and maintaining them.

The system is designed to be maximally generic: All algorithms work in the same way for every MMT language. Whenever language-specific knowledge (e.g., inference rules) is required, MMT encapsulates it in abstract interfaces (e.g., an inference rule could provide a function that takes a goal and returns subgoals). Language-specific implementations arise as MMT plugins that register appropriate instances of these interfaces (e.g., one instance for every needed inference rule).

The main achievement of MMT is to demonstrate that the generic parts of complex algorithms can dramatically outweigh the language-specific ones. In Sect. 6, we briefly showcase some examples, in particular the module system and the user interface.

In the following, we give an overview of those abstract interfaces that are most relevant for defining MMT languages.

### *Lexing and Parsing Rules.*

MMT plugins may register additional lexing and parsing rules. However, because all operators must be declared as MMT constants and can have very flexible notations, MMT's generic implementation of lexing and parsing is almost always sufficient.

The only exception are literals, in which case additional lexing rules can be provided as described in [Rab14b].

### *Inference Rules.*

The type reconstruction algorithm we will describe in Sect. 4 provides several new abstract interfaces for various kinds of inference rules. These include, e.g., rules for type inference and equality checking. The central step of implementing an MMT language is to register the appropriate rules.

### *Substitution.*

MMT's default implementation applies substitutions immediately. But every subterm caches its free variables so that substitution only recurses where necessary.

Using an appropriate constant for an additional binder, plugins can provide alternative implementations that use explicit substitutions. In that case, additional inference rules have to be supplied.

### *Change Listeners.*

While MMT validates a theory (by applying the rules in Fig. 5), it calls change listeners registered by plugins.

In particular, a change listener can provide functions to be called when a constant declaration is added, deleted, or updated. A typical application of change listeners is to elaborate high-level into low-level declarations.

# 4. GENERIC TYPE RECONSTRUCTION

The main component of MMT's type reconstruction algorithm is the **solver**. It takes a judgment with some unknown meta-variables and tries to derive the judgment and find solutions for the meta-variables. It does so by performing a bidirectional type-checking algorithm in which goals are delayed if the presence of unsolved meta-variables precludes progress.

The solver consists of 6 mutually recursively algorithms and 7 abstract interfaces for inference rules. Plugins may provide any number of instances for these 7 interfaces. No instances are hard-coded by MMT.

4 checking algorithms each take a judgment (typing, equality, inhabitability, or universe judgment) and return a boolean. Type inference and simplification take a term in context and optionally return another term. Each algorithm performs some reasoning generically and relegates to rules for language-specific reasoning. For example, type inference of constants and variables is handled generically by lookup;

type inference of a complex term uses the head to choose a type inference rule.

Below we describe the solver as well as the 6 algorithms and the 7 rule interfaces.

*Parsing.*

Type reconstruction is implemented independently of parsing. However, the generic notation-based parser of MMT can provide the input for reconstruction.

In that case, the parser uses the notations to determine the positions of implicit arguments and omitted variable types and inserts fresh meta-variables for them.

Because meta-variables represent closed terms, each meta-variable is applied to the list of bound variables in whose scope it occurs.

## 4.1 The Solver

The input to the solver consists of a theory $\Sigma$ and a $\Sigma, M$-judgment $J$ that is to be proved. Here $M = X_1, \ldots, X_n$ is a theory declaring one constant for each unknown meta-variable. Each $X_i$ represents an unknown closed term that occurs in $J$ and is to be solved.

The solver maintains $M$ statefully. Initially, the $X_i$ have neither type nor definiens. Once these are inferred, they are added to $M$.

*Check-Delay-Activate Loop.*

Additionally, the solver statefully maintains a list $G$ of open goals, which initially contains only the judgment $J$. Each open goal may be activatable or non-activatable.

The solver performs a loop, which picks the next activatable open goal and calls the respective checking algorithm on it. If this returns false, $J$ has been disproved. Otherwise, the loop repeats. If no open goal is left to pick, $J$ has been proved. If open goals are left but no activatable ones, the solver is unable to prove or disprove $J$.

*Open Goals.*

There are two kinds of open goals. A **delayed judgments** $j$ is a judgment for which no progress is possible because unsolved meta-variables preclude picking an appropriate rule. $j$ becomes activatable if a meta-variable is solved that occurs in $j$.

A **delayed inference** is a function that must be called on the inferred type of a term $t$, but which cannot be called because unsolved meta-variables preclude inferring the type of $t$. A delayed inference becomes activatable if a meta-variable is solved that occurs in $t$.

*Solver Interface.*

All algorithms and rules have access to the following functions to change the solver's state:

- **check** takes a judgment and calls the respective checking algorithm.
- **delay check** takes a judgment and appends it to $G$.
- **infer** takes a term in context and tries to infer its type.
- **delay inference** takes a pair of a term in context and a continuation function and appends it to $G$.
- **simplify** takes a term in context and simplifies it.
- **solve definiens** takes a judgment $\cdot \vdash_\Sigma X = E$ and (i) if $M$ already contains a definiens $E'$ for $X$, checks $\cdot \vdash_\Sigma E = E'$; or (ii) sets $M$ as the definiens of $X$ in $M$.
- **solve type** takes a judgment $\cdot \vdash_\Sigma X : E$ and proceeds accordingly.
- **new meta-variable** takes a list $Y = Y_1, \ldots, Y_n$ of fresh meta-variables and an existing meta-variable $X$ and inserts $Y$ into $M$ before $X$.

**check**, **infer**, and **simplify** give access to the 6 algorithms mentioned above. If **check** or **infer** gets stuck due to unsolved meta-variables, we can call **delay check** or **delay inference** instead.

**solve definiens** and **solve type** are called whenever new information about a meta-variable is inferred.

**new meta-variable** is usually called together with **solve definiens** to perform a variable transformation. For example, in a language with product types, if we find out that $X$ must be a pair, we can add new meta-variables $Y_1$ and $Y_2$ before $X$ and solve $X$ as $(Y_1, Y_2)$.

## 4.2 The Algorithms and Rule Interfaces

Each of the 6 algorithms is partnered with a rule interface that has the same input/output signature. Each algorithm determines if an applicable rule is available; if so, it relegates to it. Therefore, each of these 6 rule interfaces names an MMT constant $c$ and is applicable to all complex terms with head $c$. For example, an inference rule for `lambda` is applicable to all terms whose head is `lambda`.

Additionally, a 7th rule interface is used for *solution rules*, which come up during equality checking.

*Inhabitability Checking.*

To check $\Gamma \vdash_\Sigma \_ : A$, we relegate to an inhabitability rule applicable to $A$. If no rule exists, we infer the type of $A$, say $U$, and check $\Gamma \vdash_\Sigma U$ `UNIV`. If type inference fails, we delay the judgment.

*Universe Checking.*

To check $\Gamma \vdash_\Sigma U$ `UNIV`, we relegate to a universe rule applicable to $U$. If no rule exists, we delay the judgment.

*Type Checking.*

To check $\Gamma \vdash_\Sigma t : A$, if $t$ is a meta-variable, we call **solve type**. Otherwise, we simplify $A$ until a typing rule is applicable to $A$ and relegate to it. If there is no rule, we infer the type of $t$ as $A'$ and check $\Gamma \vdash_\Sigma A = A'$.

*Type Inference.*

Type inference takes a term $t$ in context $\Gamma$ and optionally returns a term. We proceed based on $t$:

- If $t$ is constant, meta-variable, or variable, we look up its type in $\Sigma$, $M$, or $\Gamma$, respectively. If no type is given, we return nothing.
- If there is an inference rule applicable to $t$, we relegate to it.
- Otherwise, we return nothing.

*Equality Checking.*

Equality checking takes a judgment $j$ of the form $\Gamma \vdash_\Sigma E = E'$ and optionally the known type $A$ of $E$ and $E'$.

A *solution rule* $r$ takes an equality judgment $j$ and optionally returns a different equality judgment $j'$.

- If $E$ and $E'$ are identical, we return `true`.
- If $E$ is a meta-variable $X$, we call **solve definiens**.
- If there is a solution rule, which returns $j'$ when ap-

plied to $j$ (or to the symmetric case of $j$), we check $j'$.

- If none of the above applies, we proceed as follows:
  1. If $A$ is not provided, we determine it using type inference of $E$ or $E'$. If both fail, we delay $j$.
  2. We simplify $A$ until there is an equality rule applicable to $A$. If so, we relegate to it.
  3. If there is no applicable rule, we simplify $E$ or $E'$ and repeat the equality check.
  4. If neither $E$ nor $E'$ can be simplified further:
     (a) If $G$ has an activatable goal, we delay $j$.
     (b) Otherwise, if $E$ and $E'$ have the same head, we return the conjunction of the results of checking the equality of all arguments. If the heads differ, we return `false`.

The condition in (a) is important because solving additional meta-variables followed by simplification may change the heads of $E$ and $E'$.

*Simplification.*

Simplification takes a term $t$ in context $\Gamma$. If it can simplify $t$, it returns the simplified term. Simplification is not exhaustive and applies at most one simplification rule.

If $E$ is a variable, we return nothing. If $E$ is a constant or meta-variable with definiens $d$, we return $d$. If there is no definiens, we return nothing.

If $E$ is a complex term $c(x_1 : A_1, \ldots, x_m : A_m; E_1, \ldots, E_n)$, we return the first term returned by an applicable simplification rule.

If no applicable simplification rule returns a term, we iteratively call simplification on $E_1, \ldots, E_n, A_1, \ldots, A_m$ (in that order) until one of them can be simplified. In that case, we return the result of replacing the respective subterm in $E$. If no $E_i$ or $A_i$ can be simplified, we return nothing.

# 5. SPECIFIC LANGUAGES

## 5.1 LF

To implement LF, we build a plugin that adds the following 9 rules to Mmt. The whole plugin takes about 200 lines of code.

We provide one **inference rule** each for `type`, `Pi`, `lambda`, and `apply`. Each takes a term $E$ in context $\Gamma$. For example, the rule for `Pi` matches $E$ against $\{x : A\}B$, checks $\Gamma \vdash_\Sigma A : \texttt{type}$ and returns the result of inferring $B$ in context $\Gamma, x : A$.

Only the inference rule for `apply`, which matches $E$ against $f\,t$, is not straightforward. It tries to infer the type of $f$, matches it against $\{x : A\}B$, checks $\Gamma \vdash_\Sigma t : A$, and returns $B[t]$. But if $f = X$ is a meta-variable, whose type is not solved yet, we do not know $A$ and $B$. Therefore, we perform a meta-variable transformation: We introduce new meta-variables $X^1$ and $X^2$ before $X$ and solve the type of $X$ as $\{x : X^1\}X^2\,x$. Thus, meta-variables of function type are decomposed into meta-variables of smaller types, which can then be solved separately. In the more complex case where $f = X\,x_1\,\ldots\,x_n$ and the type of $X$ is not solved yet, we proceed accordingly.

We add a straightforward **typing rule** for `Pi`, which reduces $\Gamma \vdash_\Sigma E : \{x : A\}B$ to $\Gamma, x : A \vdash_\Sigma E\,x : B$.

We add an **equality rule** for `Pi` that implements extensionality (and thus implies $\eta$). It takes $\Gamma \vdash_\Sigma E = E'$ at type $\{x : A\}B$ and checks $\Gamma, x : A \vdash_\Sigma E\,x = E'\,x$ at type $B$.

We add two **simplification rules**. The relation between `arrow` and `Pi` is implemented by a rule that simplifies $A \to B$ to $\{x : A\}B$ for a fresh variable $x$. And $\beta$-reduction takes $([x : A]t)\,a$ in context $\Gamma$, checks $\Gamma \vdash_\Sigma a : A$, and returns $t[a]$.

Finally, we add one **solution rule**. If works on equality judgments of the form $\Gamma \vdash_\Sigma X\,x_1\,\ldots\,x_n = E$ where $X$ is an unknown meta-variable and the $x_i$ are distinct bound variables of type $A_i$. In that case, it returns $\Gamma \vdash_\Sigma X = [x_1 : A_1, \ldots, x_n : A_n]E'$. Intuitively, this rule applies $\eta$ and congruence in order to isolate $X$.

## 5.2 Polymorphic LF

Full polymorphism is a relatively complex language feature: It allows `lambda` and `Pi` to bind kinded variables as well. By comparison shallow (or ML-style) polymorphism is rather harmless and can be combined with most type theories. It restricts polymorphism to be used only at toplevel: In a declaration $c : A[= t]$, $A$ may `Pi`-bind and $t$ may `lambda`-bind kinded variables.

In Mmt, maybe surprisingly, adding shallow polymorphism to LF requires only a single rule: the following inhabitability rule for `Pi`. To check $\Gamma \vdash_\Sigma \_ : \{x : A\}B$, this rule makes two recursive checks. Firstly, it calls **delay inference** on $A$ with the continuation function that takes the type $U$ of $A$ and checks $\Gamma \vdash_\Sigma U$ `UNIV`. Secondly, it checks $\Gamma, x : A \vdash_\Sigma \_ : B$. We call the resulting language PLF.

Inspecting the rules provided for LF above, we see that this is indeed all we need. The checking and equality rules for `Pi` as well as the solution rule and the simplification rule for $\beta$-reduction can be easily reused for kinded variables. The inference rules for `lambda` and `Pi` reject kinded variables. But at toplevel (i.e., when checking a declaration $c : A[= t]$) these inference rules are never applied: When checking $\cdot \vdash_\Sigma \_ : A$, we apply the new rule above; when checking $\cdot \vdash_\Sigma t : A$, $A$ must be functional if we apply the typing rule for `Pi`.

Note that we can omit the kind of kinded variables in the same way as we omit the type of typed variables. We can also make kinded arguments implicit in the same way as typed arguments. This does not require any changes to the type reconstruction algorithm.

Above, it is important that we use *delay inference* instead of *inference*. If we omit the kind of a type variable, $A$ will be an unknown meta-variable so that the inference will only succeed later point.

*Example* 5.1 (Polymorphic Equality). Representing languages with equality in LF is tedious because we have to represent equality separately at every type, including a congruence rule for every pair of types. In PLF, we can use, e.g.,

$$\begin{array}{ll}
\texttt{equal} & : \{a : \texttt{type}\}a \to a \to \texttt{type} \quad \#\mathcal{A}_2 \doteq \mathcal{A}_3 \\
\texttt{refl} & : \{a\}\{x : a\}x \doteq x \\
\texttt{cong_term} & : \{a : \texttt{type}, x : a, y, b, f : a \to b\}x \doteq y \to f\,x \doteq f\,y \\
\texttt{cong_type} & : \{a, x : a, y, b : a \to \texttt{type}\}x \doteq y \to b\,x \to b\,y
\end{array}$$

Symmetry and transitivity are now definable.

*Example* 5.2 (Dependently-Typed Higher-Order Logic). We obtain a dependently-typed analog DHOL of higher-order logic HOL such that DHOL is to LF as HOL is to simple type theory:

$$\begin{array}{ll}
\texttt{prop} & : \texttt{type} \\
\texttt{ded} & : \texttt{prop} \to \texttt{type} \\
\texttt{equal} & : \{a : \texttt{type}\}a \to a \to \texttt{prop}
\end{array}$$

Now the propositional connectives and polymorphic quantifiers can be defined from equality in the same way as for Andrews-style HOL. Prawitz-style DHOL, where universal quantification and implication are primitive, can be defined accordingly.

### 5.3 LF-modulo

We now extend LF to LF-modulo in the style of [CD07]. To implement it within Mmt, we reflect certain constant declarations as simplification rules at run time.

We achieve this by registering a change listener $L$. When Mmt has validated a constant declaration $c : A[= t]$, it calls $L$ on it. If $A$ can be interpreted as a simplification rule, $L$ generates the rule and dynamically adds it to Mmt. Thus, later declarations can make use of it. Similarly, when the declaration is deleted, $L$ unregisters the generated rule. The whole plugin requires only about 100 lines of code.

We have some freedom in which declarations we reflect because we may not be able or willing to reflect any type $A$. In the sequel, we describe a very simple solution.

We make two conventions how users can tell our plugin which constants should be reflceted. Firstly, constants whose name ends in *_equal* are equality judgments, e.g., we can extend Fig.3 with

$$\_equal : \{a : \mathtt{tp}\}\mathtt{tm}\, a \to \mathtt{tm}\, a \to \mathtt{type} \qquad \#\mathcal{A}_2 \doteq \mathcal{A}_3$$

Secondly, constants whose name ends in *_rule* are reflected. These must have a type of the form $\{x_1, \ldots, x_n\}\ e\, l\, r$ where $e$ is an equality judgment $l$ is a term of pattern shape and $r$ is any term.

The intuition is that such a constant represents the rule $l \rightsquigarrow r$. Our plugin implements that by adding a simplification rule that takes $l[t_1, \ldots, t_n]$ and returns $r[t_1, \ldots, t_n]$.

*Example* 5.3 (Dependent Product Types). Consider the declarations from Fig. 3 and the declaration of *_equal* above. We add the following declarations for the appropriate conversion rules

$$\begin{aligned}
&\mathtt{pi1}_rule &&: \{A, B, a : \mathtt{tm}\, A, b : \mathtt{tm}\,(B\, a)\}\mathtt{pi1}\,(a, b) \doteq a\\
&\mathtt{pi2}_rule &&: \{A, B, a : \mathtt{tm}\, A, b : \mathtt{tm}\,(B\, a)\}\mathtt{pi2}\,(a, b) \doteq b\\
&\mathtt{pair}_rule &&: \{A, B, t : \mathtt{tm}\,(\Sigma[x : \mathtt{tm}\, A]B\, x)\}t \doteq (\mathtt{pi1}\, t, \mathtt{pi2}\, t)
\end{aligned}$$

This encoding is *not* well-typed in LF. The problem is that the arguments of $\doteq$ in $\mathtt{pi2}\_rule$ do not have the same type, namely $\mathtt{tm}\,(B\,(\mathtt{pi1}\,(\mathtt{pair}\, a\, b)))$ and $\mathtt{tm}\,(B\, a)$.

But in LF-modulo, the simplification rule generated from $\mathtt{pi1}\_rule$ makes those two types equal.

## 6. GENERIC FEATURES

The value of our implementation increases dramatically if it is seen together with other advanced features that Mmt provides generically. Therefore, we sketch the interplay of some of these features with the present work.

### *Module System.*

The Mmt module system [RK13] permits building composed theories using union, instantiation, and translation. This applies both to the Mmt theories we use to represent languages and to the theories of these languages. Importantly, this makes it possible to translate developments between languages, e.g., we can retain the old library when migrating to a modified language.

The module system is orthogonal to type reconstruction: When implementing individual rules, modularity is transparent to the developer.

### *User Interface.*

Mmt includes a IDE-style user interface [Rab14a] based on the jEdit text editor. It includes advanced features such as context-sensitive auto-completion, search, and change management.

The results of type reconstruction are added to the abstract syntax tree, which is displayed in a sidebar in the editor. Moreover, inferred types and implicit arguments are shown as tooltips when hovering over a variable or constant, respectively. Hovering over a selected subterm dynamically runs type inference and displays the result. All IDE functionality degrades gracefully: If reconstruction cannot solve all unknowns, the IDE works with the partial solution, is crucial to find errors.

The reconstruction algorithm traces all steps, and individual rules may insert custom messages into the trace tree. For example, the inference rule for `apply` adds a message about the expected type to the branch in which it checks the argument. If reconstruction finds a typing error, the IDE shows the list of all trace messages on the branch that led to the error. This is important because early messages on the trace are often more helpful than later ones.

## 7. CONCLUSION

### *Contribution.*

We have investigated the type theoretical properties of the Mmt language and system. Our work aims at a systematic separation of concerns between (i) the small scale definition of a type theory by giving operators, notations, and typing rules, and (ii) a large scale implementation that is realized generically for all languages. This enables a rapid prototyping effect where developers can focus on the type theoretical essentials and obtain a major implementation at extremely low cost.

We have applied our system to obtain implementations of LF, LF with shallow polymorphism, and LF-modulo. (Due to the modular nature of Mmt, we obtain a fourth example by combining the latter two.) We also mention that [Rab14b] recently developed an abstract interface that permits adding any literals to any Mmt language.

We have demonstrated the feasibility of this approach by describing a generic type reconstruction algorithm at the Mmt level integrating it with other generic features of Mmt: module system and IDE. In all cases, the generic code is several orders of magnitude larger than the language-specific generic code.

### *Related Work.*

We revisit some related work specific to our examples. The closest relatives of our LF implementation are Twelf [PS99], Beluga [PD10], and Delfin [PS08]. These employ similar type reconstruction algorithms [Pie13].

The main difference is that they allow free variables, which are abstracted once their types are inferred (including additional variables that only occur in these inferred types). Our algorithm does not support that at this point and requires all variables to be bound a priori. This makes it more

similar to the ones in Agda [Nor05] and Matita [ACTZ06].

The importance of abstraction already becomes clear in our examples. For example, in Ex. 5.1, we have to tediously bind $a$, $b$, etc. Therefore, we plan to strengthen our algorithm by generalizing Twelf-style abstraction to MMT. Our idea is that if no open goals are left and for some meta-variables only the type but not the definiens has been solved, we can abstract over these meta-variables and make them additional implicit arguments.

Moreover, LF-specific algorithms exploit the canonical form theorem of LF. This is not possible at the generic MMT level because MMT cannot tell whether canonical forms exist in a specific case. Finally, our algorithm differs by using kinded meta-variables, which works very well at the MMT level but can be unnecessarily complex for a pure LF algorithm.

The closest relative of our LF-modulo implementation is the Dedukti system [BCH12]. Dedukti does not support type reconstruction and focuses on fast type checking. It permits more general rewrite rules than we but requires all rewrite rules for the same toplevel symbol to be grouped together.

Our polymorphic variant of LF can be seen as combining features of two logical frameworks: the dependent types of LF and the polymorphic higher-order logic of Isabelle [Pau94].

### *Future Work.*

We expect that our approach can be generalized substantially. We plan to add a subtyping judgment and generalize existing reconstruction algorithms that support subtyping to the MMT level. This should also include the reconstruction of implicit coercions. Here the description in [ARCT12] is a good starting point.

We have tentatively added support for a second kind of equality rule to handle a goal $E = E' : A$: Here, instead of the head of $A$, we use the pair of heads of $E$ and $E'$ to choose an applicable rule. We expect that this will generalize unification in the style of canonical instances [GGMR09] or unification hints [ARCT09] to MMT.

In orthogonal work, we are developing a generic treatment of inductive and record types in MMT. Once available, we plan to generalize our implementation of type reconstruction accordingly. We cannot say yet how difficult this will be.

## 8. REFERENCES


[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.

[ARCT09] A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. Hints in unification. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 84–98. Springer, 2009.

[ARCT12] A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions. *Logical Methods in Computer Science*, 8(1), 2012.

[BCH12] M. Boespflug, Q. Carbonneaux, and O. Hermant. The λΠ-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.

[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.

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

[GGMR09] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 327–342. Springer, 2009.

[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.

[Lut01] M. Luther. More On Implicit Syntax. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning*, pages 386–400. Springer, 2001.

[Nor05] U. Norell. The Agda WiKi, 2005. `http://wiki.portal.chalmers.se/agda`.

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

[PD10] 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.

[Pie13] B. Pientka. An insider's look at LF type reconstruction: everything you (n)ever wanted to know. *J. Funct. Program*, 23(1):1–37, 2013.

[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 Metalanguages: Theory and Practice*, pages 135–141. ENTCS, 2008.

[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. see `http://kwarc.info/frabe/Research/rabe_ui_14.pdf`, 2014.

[Rab14b] F. Rabe. Generic Literals. see `http://kwarc.info/frabe/Research/rabe_literals_14.pdf`, 2014.

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