A Generic Type Theory

Florian Rabe

Jacobs University, Bremen, Germany

Abstract. We describe the type theoretical aspects of the MMT language. It systematically abstracts from theoretical and practical aspects of type theories and tries to develop as many solutions as possible generically. This permits type theory designers to focus on the essentials and to inherit a large scale implementation at low cost.
We define a type system for MMT that can be easily instantiated with specific type theories. Our central result is the design and implementation of a generic type reconstruction algorithm.
When instantiating MMT, a major advantage is the flexibility and extensibility. We demonstrate this by obtaining an implementation of the dependently-typed λ-calculus and extending it to an implementation modulo rewriting.

1 Introduction and Related Work

Generic Large Scale Features At small scales, introducing a new type theory is relatively simple: We only have to give a grammar and an inference system. Additionally, we should try to prove the decidability of typing and implement type checking.

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

This impedes evolution. Problems in the small scale design may become apparent only in large case studies, which can only be carried out after a major investment into large scale features. And new type theoretical features can rarely be added easily to an existing large scale implementation.

Our work derives from the hypothesis that (i) the small scale features are highly type theory-specific, but (ii) many large scale features can be realized generically. This motivates a separation of concerns where small scale definitions of type theories are combined with generic large scale implementations.

To that end, we introduced the MMT language in [RK13], which abstracts from language-specific features and focuses on modeling the structural properties shared by the vast majority of languages. MMT already realizes some of the needed large scale features generically: [RK13] introduces a generic module system; and [Rab14] presents a generic IDE-style user interface.

In this paper, we present a further generic large scale feature: We design and implement a generic type reconstruction algorithm. Moreover, by separating the type theory-independent from the type theory-specific aspects, we obtain a relatively clear structure that makes our algorithm easier to present and understand than existing descriptions of similar ones.

Type Reconstruction By "type reconstruction", we mean the problem of inferring unknown subexpressions that were omitted in a given term. Typical examples are implicit arguments, types of bound variables, implicit coercions, or 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 [The14], 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, both our theoretical description and our implementation only fix structural rules and assume arbitrary sets of operators, notations, and typing rules. Theoretically, our main result is that we can give a meaningful type reconstruction algorithm despite this open-world assumption. Practically, our main result is a generic implementation that can be customized easily by supplying plugins for specific type theories.

We evaluate this approach in two examples. Firstly, we instantiate our work with dependent type theory (as in LF [HHP93]). This requires only the straightforward implementation of 10 simple rules while the complexity of the reconstruction remains hidden. Secondly, we obtain an instance of LF modulo a user-declared rewrite system; this is inspired by the Dedukti [BCH12] system which implements the $\lambda\Pi$-modulo calculus of [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. Then we represent the operators, notations, and typing rules of L in terms of their counterparts of F so that large scale implementations of F can be applied to L. In this sense, the reconstruction algorithms for LF-based frameworks [Pie13] 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 an arbitrary set of rules, and to instantiate our framework with L we directly supply the appropriate rules.

This makes it harder to supply typing rules in practice: Rules can no longer 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 or provide customized error messages. For example, this permit adding rules for constraint domains, which are difficult to define declaratively. And our implementation minimizes the programming cost by offering simple plugin interfaces.

Moreover, MMT can serve as a meta-framework in which other logical frameworks are designed. Indeed, our two examples first instantiate MMT with LF and then extend that to LF-modulo. This extension only requires implementing a simple component that dynamically adds the needed rewriting rules.

Overview We present the syntax and the generic parts of our type system in Sect. 2 and 3. Then we present our type reconstruction algorithm in Sect. 4. In Sect. 5, we describe how type reconstruction interacts with other generic large scale features that we have developed based on MMT, namely module system, parsing, and graphical user interface. In Sect. 6, we revisit some related work and conclude.

2 Generic Syntax

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

Fig. 1. MMT Grammar

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.

`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`	#	$\mathcal{A}_1 \to \mathcal{A}_2$

Fig. 2. LF-Syntax in MMT

A **theory** Σ 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** Γ 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, Γ declares the **bound variables**, and the E_i are the **arguments**. Both bound variables and arguments are optional, in particular $\Gamma = \cdot$ yields the case of non-binding operators. 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.

The syntax of Mmt is generic in the sense that Mmt does not have any predefined constants c. To form any terms, we first have to declare some (untyped) constants for the primitive operators of the desired type theory.

$$\begin{array}{lll}
\texttt{u} & : \texttt{type} & \#\ \texttt{u} \\
\texttt{equal} & : \texttt{u} \to \texttt{u} \to \texttt{type} & \#\ \mathcal{A}_1 \doteq \mathcal{A}_2 \\
\texttt{f} & : \texttt{u} \to \texttt{u} & \\
\texttt{idemp} & : \{x : \texttt{u}\}(\texttt{f}\,(\texttt{f}\,x)) \doteq (\texttt{f}\,x) &
\end{array}$$

Fig. 3. An LF-Theory in Mmt

Example 1 (LF as an Mmt *Theory).* To obtain 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 λ-abstraction $\texttt{lambda}(x : A; t)$, $\texttt{lambda}$ 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 $f\ t$.

Fig. 3 gives some example declarations relative to LF. They declare a type with an equality predicate and an idempotent unary function on it.

Finally, we define **substitution** generically at the Mmt level: 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.

3 Generic Type System

Main judgments	
$\Gamma \vdash_\Sigma t : A$	t has type A
$\Gamma \vdash_\Sigma E = E'$	E and E' are equal
$\Gamma \vdash_\Sigma E\ \texttt{UNIV}$	E is a universe
$\vdash \Sigma$	Σ is a valid theory
$\vdash_\Sigma \Gamma$	Γ is a valid Σ-context
Abbreviations	
$\Gamma \vdash_\Sigma \_ : A$	$\Gamma \vdash_\Sigma A : E$ and $\Gamma \vdash_\Sigma E\ \texttt{UNIV}$
$\Gamma \vdash_\Sigma t :!$	$\Gamma \vdash_\Sigma t : A$ for some A

Fig. 4. Judgments

Now we go beyond [RK13] by defining a generic type system for MMT terms. We use the **judgments** given in Fig. 4. Terms subsume all types, kinds, universes, etc., and typing and equality are simply binary relations between terms – all further details are left to the rules supplied by an individual type theory.

Because the MMT syntax introduces no predefined constants, the type system only contains structural **rules**. These are given in Fig. 5. The rules for terms are straightforward: constants and variables have types according to their declaration, constants are equal to their definiens, binding admits α-conversion, and equality is an equivalence and congruence relation. The rules for theories and contexts make use of the universe judgment: A may occur as the type of a constant or variable if A is typed by a universe. We use the judgment $\Gamma \vdash_\Sigma \_ : A$ to abbreviate this.

valid theories (ignoring notations) and contexts:

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

lookup in theories and contexts:

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

α-conversion:

$$\frac{A'_i = A_i[y'_1, \dots, y'_{i-1}] \quad E'_i = E_i[y'_1, \dots, y'_n]}{\Gamma \vdash_\Sigma c(\overrightarrow{y_m : A_m}; E_1, \dots, E_n) = c(\overrightarrow{y'_m : A'_m}; E'_1, \dots, E'_n)}$$

equality is equivalence:

$$\frac{}{\Gamma \vdash_\Sigma E = E} \qquad \frac{\Gamma \vdash_\Sigma E = E'}{\Gamma \vdash_\Sigma E' = E} \qquad \frac{\Gamma \vdash_\Sigma E = E' \quad \Gamma \vdash_\Sigma E' = E''}{\Gamma \vdash_\Sigma E' = E''}$$

equality is congruence:

$$\frac{\Gamma, \overrightarrow{y_{i-1} : A_{i-1}} \vdash_\Sigma A_i = A'_i \quad \Gamma, \overrightarrow{y_n : A_n} \vdash_\Sigma E_i = E'_i}{\Gamma \vdash_\Sigma c(\overrightarrow{y_m : A_m}; E_1, \dots, E_n) = c(\overrightarrow{y_m : A'_m}; E'_1, \dots, E'_n)}$$

$$\frac{\Gamma \vdash_\Sigma t : A \quad \Gamma \vdash_\Sigma t = t' \quad \Gamma \vdash_\Sigma A = A'}{\Gamma \vdash_\Sigma t' : A'} \qquad \frac{\Gamma \vdash_\Sigma \_ : A \quad \Gamma \vdash_\Sigma A = A'}{\Gamma \vdash_\Sigma \_ : A'}$$

Notation: $\overrightarrow{y_m : A_m} = y_1 : A_1, \dots, y_m : A_m$

Fig. 5. Structural Rules

A type theory can now be represented as a pair of an MMT theory and a set of additional rules.

Example 2 (Rules of LF). To represent LF, we extend Ex. 1 with the following rules where U ranges over $\{\mathtt{type}, \mathtt{kind}\}$

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

The latter legitimizes typed constant declarations $c : A$ for $A : \mathtt{type}$ and kinded constant declarations $c : K$ for $K : \mathtt{kind}$.

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

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

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

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

which correspond to the rules $(*, *)$ and $(*, \Box)$ of pure type systems, as well as rules for β and η-equality.

4 Generic Type Reconstruction

It is good practice in the meta-theory of formal languages to favor declarative over algorithmic presentations because they are easier to understand and reason about. However, after several attempts, we found that in this particular case the algorithmic form actually works better for two reasons. Firstly, by using a few stateful global variables, we can dramatically simplify the notation compared to, e.g., [Nor05] or [Pie13]. Secondly, we can still reason quite intuitively about the soundness, although of course not formally. An additional advantage is that our presentation is very close to our implementation and thus easy to recreate.

Our type reconstruction algorithm consists of 3 levels described in Sect. 4.1, 4.2, and 4.3. The **high level** provides the interface visible to the outside and maintains the global state. After initializing the global state, it relegates processing to the respective mid level component. The **mid level** consists of several mutually recursive algorithms: one for each judgment about terms (inhabitability checking, type checking, and equality checking) as well as type inference and simplification. It implements the structural rules of and relegates to the low level rules as needed. The **low level** consists of the type theory-specific rules, which must be provided separately.

4.1 The High Level

Overview Let $M = X_1, \dots, X_n$ be a theory and let J be an inhabitability, typing, or equality judgment over the theory Σ, M. We call the X_i the **meta-variables** of J.

Our goal is to find definitions for the X_i such that J holds. More precisely, a **partial solution** is a theory S that declares one constant for each X_i. X_i is **solved** in S if S declares a definiens for X_i, and S is a **total solution** if all its variables are solved.

The high level maintains the **global state** as given in Fig. 6. D is the list of judgments that must be proved (we can think of them as open goals), and *act* contains one boolean flag for each. We call $J_i \in D$ **activatable** if $act_i =$ `true`. Initially, D and *act* contain only one entry for an activatable judgment J.

The global variable S maintains the partial solution found so far. Initially no X_i is solved. S is called **correct** if all judgments in D hold over Σ, S. Our goal is to transform D and S such that D is empty and S is a total solution, while maintaining the invariant that the set of correct solutions does not change. In that case, we can conclude that S is the unique correct solution.

Variable	Stateful?	Intuition
J	no	the initial judgment to check
Σ	no	the theory of J
D	yes	the list of judgments to prove, initially contains only J
act	yes	the boolean flags remembering which judgment in D are activatable, initially contains only `true`
S	yes	the partial solution found so far, initially $X_1, \dots, X_n$

Fig. 6. Input and Global State

Delay, Solve, Transform, and Activate Due to the presence of meta-variables in J, we may get stuck while checking J. For example, if j is the judgment $\cdot \vdash_\Sigma t : X$, we may get stuck because we do not know which rule to apply at the unknown type X. In that case, we **delay** j by appending it to D and continue with the next judgment in D. If other branches of the derivation encounter the judgment $\cdot \vdash_\Sigma X = A$, we can **solve** X as A. At this point j becomes $\cdot \vdash_\Sigma t : A$ and can be **activated**. Finally, we are occasionally able to solve a meta-variable partially, e.g., if we find out the first projection of an unknown pair. In that case, we perform a meta-variable **transformation**.

More precisely, the high level provides the following stateful operations:

- **delay** takes a judgment j, appends it to D, appends `false` to *act*, and returns `true`.
- **solve** takes a judgment $\cdot \vdash_\Sigma X = E$ and:
  - if S already contains a definiens for X: checks $\cdot \vdash_\Sigma E = E'$.

  - otherwise: adds E as the definiens of X and makes every judgment in D activatable that refers to X

  Accordingly, solve uses a judgment $\cdot \vdash_\Sigma X : E$ to add E as the type of X.
- **transform** takes a list $Y = Y_1, \ldots, Y_n$ of new meta-variables, an existing meta-variable X, and a term $E[Y_1, \ldots, Y_n]$. It inserts Y into S in front of X, and solves X as E.

Algorithm We can now define the high level algorithm as the following loop:

1. If D is empty:
   - If S is a total solution, return `true`.
     Intuition: Success – J was proved and all meta-variables solved.
   - Otherwise, return `false`.
     Intuition: Underspecified – J was proved but not all meta-variables solved.
2. Otherwise:
   - If D contains no activatable judgment, return `false`.
     Intuition: Unproved – not all judgments were derived, and we give up.
   - Otherwise, remove the first activatable judgment j from D and
     Intuition: j gets activated.
     (a) obtain j' from j by substituting all solutions of meta-variables and simplifying all terms
     (b) call the respective mid level algorithm on j'. If this returns,
       - `true`: go to step (1).
         Intuition: The judgment is proved, and we activate the next one.
       - `false`: return `false`.
         Intuition: Disproved – J does not hold.

4.2 The Mid Level

Algorithm	Input	Output	
		Success	Failure
universe checking	$\Gamma \vdash_\Sigma E$ `UNIV`	`true`	`false`
type checking	$\Gamma \vdash_\Sigma t : A$	`true`	`false`
equality checking	$\Gamma \vdash_\Sigma E = E'$	`true`	`false`
type inference	t	A such that $\Gamma \vdash_\Sigma t : A$ is covered	nothing
simplification	E	E' such that $\Gamma \vdash_\Sigma E = E'$ is covered	nothing

Fig. 7. Mid Level Algorithms

The mid level consists of the mutually recursive algorithms given in Fig. 7. Here we call a judgment j **covered** if it is implied by the judgments in D and the judgments about the meta-variables that we store in S. Thus, a covered judgment holds about every correct solution.

Each mid level algorithm is parametrized by certain low level rules. In the sequel, we describe each algorithm and its rules.

Universe Checking A **universe rule** for c has the following form:

input	term E with head c
output	boolean

Relative to a set of universe rules, **universe checking** of $\Gamma \vdash_{\Sigma} E$ `UNIV` proceeds as follows:

– if E has a head for which there is a universe rule, relegate to it,
– otherwise, return `false`.

Type Checking A **typing rule** for c has the following form:

input	terms t and A such that the head of A is c
precondition	$\Gamma \vdash_{\Sigma} \_ : A$ is covered
output	boolean

Relative to a set of typing rules, **type checking** of $\Gamma \vdash_{\Sigma} t : A$ proceeds depending on t:

– If t is a meta-variable X: solve the type of X as A
– If t is a constant of Σ or a variable of Γ: look up its type as A' and relegate to $\Gamma \vdash_{\Sigma} A = A'$.
– If t is a term with head c:
  - If there is a typing rule for c, relegate to it.
  - Otherwise, infer the type of t. If this returns:
    - a term A', relegate to $\Gamma \vdash_{\Sigma} A = A'$.
    - nothing: delay the input judgment.

Equality Checking An **equality rule** for c has the following form:

input	terms E, E', A such that A has head c
precondition	$\Gamma \vdash_{\Sigma} A$ `UNIV`, $\Gamma \vdash_{\Sigma} E : A$ and $\Gamma \vdash_{\Sigma} E' : A$ are covered
output	boolean

A **solution rule** for c has the following form:

input	terms E, E' such that E has head c and minor head X
output	optional terms F, F'
postcondition	if (F, F') is returned, then $\Gamma \vdash_{\Sigma} E = E'$ is covered only if $\Gamma \vdash_{\Sigma} F = F'$ is no change to the global state

Here the **minor head** of E is the first atomic term we find if we recursively take the first argument of E. For example, X is the minor head of $\texttt{apply}(\cdot; \texttt{apply}(\cdot; X, t_1), t_2)$.

Relative to a set of equality and solution rules, **equality checking** of $\Gamma \vdash_{\Sigma} E = E'$ proceeds as follows:

1. If E and E' are identical, return `true`.
2. Otherwise, if E is a meta-variable X, solve X as E.
3. Otherwise, if there is a solution rule for $\Gamma \vdash_\Sigma E = E'$ or $\Gamma \vdash_\Sigma E' = E$ that returns (F, F'), relegate to $\Gamma \vdash_\Sigma F = F'$.
4. Otherwise:
   (a) Infer the type of E and if that fails of E'. If either returns A continue, otherwise delay the input judgment.
   (b) If A has a head for which there is an equality rule, relegate to it. Otherwise, if A can be simplified, simplify A and repeat this step.
   (c) Otherwise, if E and E' cannot be simplified, if there is no activatable judgment, and if E and E' have the same head, recursively check equality of all arguments.
   (d) Otherwise, delay the input judgment.

Type Inference An **inference rule** for c has the following form:

input	term t in context Γ with head c
precondition	$\Gamma \vdash_\Sigma t :!$ is covered
output	an optional term
postcondition	if a term A is returned, then $\Gamma \vdash_\Sigma t : A$ is covered

Relative to a set of inference rules, **type inference** of t in context Γ proceeds as follows:

– If t is meta-variable, returns its type in S.
– If t is a constant or a variable, look up its type and return it.
– If t has a head for which there is an inference rule, relegate to it.
– Otherwise, return nothing.

Simplification A **simplification rule** for c has the following form:

input	term E with head c
precondition	$\Gamma \vdash_\Sigma E :!$ is covered
output	optional term
postcondition	if the output is E', then $\Gamma \vdash_\Sigma E = E'$ is covered
	no change to the global state

Relative to a set of simplification rules, **simplification** of E proceeds as follows:

– If E is a variable, return nothing.
– If E is a constant or meta-variable, return its definiens (possibly nothing).
– If E is a complex term $c(x_1 : A_1, \ldots, x_n : A_n; E_1, \ldots, E_n)$:
  - If there is a simplification rule for c that returns E', return E'.
  - Otherwise, iteratively simplify $E_1, \ldots, E_n, A_1, \ldots, A_n$ (in the respective context) until one of them returns a term. Replace the result in E and return the resulting term.

4.3 The Low Level

The low level is not generic anymore because it provides the language-specific universe, typing, equality, solution, inference, and simplification rules.

Type Reconstruction for LF We supply all rules from Ex. 2 except for `apply` and η. The rules `type`, `Pi`, and `lambda` are inference rules for the respective constant, the rule `UNIV` is a universe rule, and β is a simplification rule.

Moreover, we supply the following derivable rules:

- an inference rule for `apply`, which implements the corresponding rule from Ex. 2 with the following modification. If the type of $f = X$ cannot be inferred, we perform a meta-variable transformation: We introduce new meta-variables X^1 and X^2, solve X as $\{x : X^1\}X^2\,x$. In case $f = X\,x_1\,\ldots\,x_n$, we proceed accordingly. This has the effect that meta-variables of function type are decomposed into meta-variables of smaller types, which can then be solved separately.
- a typing rule for `Pi`:

$$\frac{\Gamma, x : A \vdash_\Sigma f\,x : B \qquad x \notin \Gamma}{\Gamma \vdash_\Sigma f : \{x : A\}B}$$

- extensionality (which is equivalent to η) as an equality rule:

$$\frac{\Gamma, x : A \vdash_\Sigma f\,x = f'\,x : B \qquad x \notin \Gamma}{\Gamma \vdash_\Sigma f = f' : \{x : A\}B}$$

- a simplification rule to eliminate `arrow` in favor of `Pi`:

$$\frac{x \notin \Gamma}{\Gamma \vdash_\Sigma A \to B = \{x : A\}B}$$

- a solution rule for `apply`:

$$\frac{\Gamma \vdash_\Sigma X\;x_1\;\ldots\;x_n = E' \qquad x_i : A_i \in \Gamma \qquad \text{all } x_i \text{ distinct}}{\Gamma \vdash_\Sigma X = [x_1 : A_1, \ldots, x_n : A_n]E'}$$

The implementation of the plugin with all rules and error reporting requires only about 200 lines of code.

Type Reconstruction for LF-modulo We extend LF to LF-modulo in the style of [CD07]. The main idea is to treat certain constant declarations as simplification rules and add them dynamically. Specifically, we use constants c whose type is of the form

$$\{x_1\}\ldots\{x_n\}\;r\,s\,t \qquad (*)$$

for a constant r and terms s and t. The intuition is that r is a (user-declared) judgment for rewriting and c represents the rewrite rule $s \rightsquigarrow t$. In our implementation, we restrict attention to the case where s is of the form $c\,\ldots\,(d\,\ldots)\,\ldots$

where c and d are constants and all ellipses are lists of bound variables. Like in Dedukti, the confluence of the rewrite system remains the user's responsibility.

To implement this, we make use of two additional features of the MMT implementation. Firstly, users can ascribe metadata to a constant declaration. Secondly, plugins can register change listeners that are called whenever a constant declaration is added or removed.

Thus, we define a tag `Rewrite` and register a change listener that acts on any change to a constant c with that tag. If c is added, we match its type against $(*)$, create the run time object representing the corresponding simplification rule, and register the rule. If c is deleted, we delete the generated rule. The whole plugin requires only about 100 lines of code.

Example 3. Consider the declarations from Ex. 1. If we tag `idemp` with `Rewrite`, our implementation generates a rewrite rule $\mathtt{f}\,(\mathtt{f}\,x) \rightsquigarrow \mathtt{f}\,x$. In subsequent declarations, this rule will be a part of the simplification algorithm.

4.4 Soundness

We sketch an intuitive argument for the soundness of our algorithm.

Definition 1 (Soundness). *The high level is* ***sound*** *if at step (1) the set of correct solutions of D is always the same.*

Theorem 1. *If the high level is sound and returns* `true` *for a judgment J, then S is the unique correct solution of J (up to provable equality).*

Proof. Initially D is just $\{J\}$. When returning `true`, D must be empty and S total. Then the result follows from the soundness.

Definition 2. *A universe, typing, or equality rule R is* ***sound*** *if the following holds: If j satisfies the precondition and $R(j)$ succeeds, then the set of correct solutions of $D \cup \{j\}$ before calling A is the same as the set of correct solutions of D after calling A.*

An inference, solution, or simplification rule R is ***sound*** *if it satisfies its pre-/post-condition and the sets of correct solutions of D are the same before and after calling R.*

Theorem 2. *If all rules are sound, then the high level is sound.*

Proof. The basic idea is to prove soundness lemmas for each mid level algorithm, which correspond closely to the soundness of rules.

5 Generic Large Scale Features

We have implemented the generic syntax, judgments, and type reconstruction within the MMT system.[1] The value of our implementation increases dramatically if it is seen together with other large scale features that MMT provides. All of them are generic and immediately available for any type theory we represent. Therefore, we sketch the interplay of these features with the present work.

[1] https://svn.kwarc.info/repos/MMT/doc/api/index.html

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 modfied language.

The module system is orthogonal to type reconstruction: When implementing individual rules, modularity can be ignored completely. And if a rule is sound in the non-modular setting presented here, it is guaranteed to be sound in the modular setting as well.

Parsing We have developed a parsing algorithm for MMT using an extended version of the notation language presented here. The parser inserts fresh meta-variables for unknown subterms, which can then be solved by reconstruction.

This applies to the omitted types of bound variables and to all arguments that are not mentioned by the notation. The latter subsumes implicit arguments and type parameters of polymorphic constants.

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

User Interface MMT includes a IDE-style user interface for MMT [Rab14] 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 displayed in a sidebar and in tooltips. For example, hovering over a constant shows its implicit arguments. All IDE functionality degrades gracefully: If reconstruction cannot find a total solution, the IDE uses the partial solution. This is important to help users find errors.

All rule applications are traced, and rules can insert custom messages into the trace. If reconstruction fails, the IDE shows all branches of the derivation that led to errors. This works even it there were delay-activate interruptions on the branch. This is important because early messages on the trace are often more helpful than later ones.

6 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 type theories. 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 demonstrated the feasibility of this approach by describing a generic type reconstruction algorithm at the MMT level. Moreover, we have integrated it with other generic features of MMT: module system, parser, and IDE.

We have applied our system to obtain implementations of LF and LF-modulo as examples. The latter is particularly notable because we obtained it as a simple extension of the former – a corresponding extension of other implementations of LF would be very hard.

The same approach can now be used to easily implement other extensions of LF, e.g., LF with product types, shallow polymorphism, or number literals.

Related Work We revisit some related work that is specific to our two examples. The closest relatives of our LF implementation are Twelf [PS99], Beluga [PD10], and Delfin [PS08]. These employ similar type reconstruction algorithms [Pie13] that are stronger than ours. While we assume that all unknown meta-variables are known before hand, they generate additional meta-variables along the way, which are eventually bound in an abstraction step. This is important in practice because meta-variables in dependently-typed languages often occur only in the types of other meta-variables. We have not yet investigated whether abstraction can be generalized to MMT.

Moreover, these 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 MMT level algorithm becomes simpler by using meta-variables for LF types.

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.

In both cases, we expect our systems to be slower because our design prevents most language-specific optimizations.

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 also includes the reconstruction of implicit coercions. Here the description in [ARCT12] is a good starting point.

We have already 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 eventually yield support for unification in the style of canonical instances [GGMR09] or unification hints [ARCT09].

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 $\lambda\Pi$-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.
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.
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.
Rab14.	F. Rabe. A Logic-Independent IDE. see `http://kwarc.info/frabe/Research/rabe_ui_14.pdf`, 2014.
RK13.	F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, 230(1):1–54, 2013.
The14.	The Coq Development Team. The Coq Proof Assistant: Reference Manual. Technical report, INRIA, 2014.