MMT: A Foundation-Independent Logical Framework

Florian Rabe

FAU Erlangen and LRI Paris

Abstract. Logical frameworks like Twelf and Isabelle are meta-formalisms in which the syntax and semantics of object logics and related formal systems can be defined. This allows developing support services like theorem proving generically and reuse them for many object logics.
MMT follows this tradition but abstracts even further than previous frameworks: it avoids foundational commitments as much as possible and does not build in any primitive features like function types or propositions. Instead, all MMT algorithms are parametric in a set of rules, which are supplied by the language designer as self-contained objects in the underlying programming language. That makes it easy to define language features that existing frameworks cannot express. In particular, MMT allows designing and implementing new logical frameworks.
Despite this high level of generality, it is possible to develop sophisticated results in MMT. The current release, in development since 2006, includes, e.g., parsing, type reconstruction, module system, IDE-style editor, and interactive library browser. It is systematically designed to be extensible, providing multiple APIs and plugin interfaces, and thus provides a versatile infrastructure for system development and integration.

1 Introduction and Related Work

Despite its potential and successes, the automation of formal systems proceeds very slowly because designing them, implementing them, and scaling these implementations to practical tools is extremely difficult and time-consuming. **Logical frameworks** have been recognized as an important method for optimally allocating resources. They provide meta-logics in which the syntax and semantics of object logics can be defined. This enables, e.g., reasoning about object logics (Twelf, Abella) or generic proof assistants (Isabelle). Moreover, logical frameworks are invaluable for **experimentation**: they can reduce the design-implement-scale process from person-years to person-days and thus speed up the feedback loop by orders of magnitude.

Recently, the **limitations** of logical frameworks have received attention. Recent systems added concurrency (CLF), reasoning about contexts (Beluga), rewriting (Dedukti), side conditions (LLFP), or proof-assistant support (Hybrid). Moreover, many logic-specific systems are investigating how to give users more freedom to change the system's behavior, e.g., via meta-programming (Idris, Lean) or unification hints (Coq, Matita).

Within this trend, MMT takes an extreme approach: it tries to systematically avoid any commitment to logical foundations, while still allowing useful generic results. It is so flexible that it cannot only be used as a logical framework but also to implement other logical frameworks. In fact, it does not fix any meta-logic at all — every MMT development starts with the definition of a logical framework. (This gave rise to the name MMT as an abbreviation of meta-meta-theory/tool.) MMT achieves this flexibility by exposing an abstract class `Rule` of the underlying programming language (Scala). All core algorithms are parametrized by a set of `Rule` instances, which are collected from the current context. Thus, entirely different system behaviors can coexist in the same development.

This makes MMT most similar to the ELPI framework [DGCT15]. Both use an untyped expression language into which object language syntax is embedded and a programming language for writing rules. But while ELPI uses the same language (λ-Prolog) for defining syntax and rules, MMT uses a purely declarative language for the syntax and a general purpose programming language for the latter. That way, it exerts more control over syntax definitions and less control over rule definitions than ELPI. The latter is important in particular when implementing logical frameworks in MMT, where fine-grained control over, e.g., error-reporting or side conditions is needed.

Overview We sketch the definition of the MMT language in Sect. 2 and describe the implementation in Sect. 3. As a running example, we build `DHOL`, a dependently-typed higher-order logic modulo rewriting — a formal system that to our knowledge is in itself novel. Sect. 4 summarizes libraries developed in MMT so far, and Sect. 5 concludes with a description of future work. Implementation and documentation are available at `https://uniformal.github.io/`.

Acknowledgments MMT has been designed and developed since about 2006, and various contributions to language, system, and libraries were made by members of the Kwarc group at FAU Erlangen-Nürnberg (previously at Jacobs University Bremen), in particular Michael Kohlhase, Fulya Horozal, Mihnea Iancu, and Dennis Müller. Some developers were partially supported by grants KO-9 LATIN, RA-18723-1 OAF, KO-2428/13-1 OAF (all DFG), Leibniz-SAW MathSearch, and Horizon 2020 ERI 676541 OpenDreamKit.

Several aspects of language and implementation have been published previously, most importantly the core language [Rab17b], the type reconstruction algorithm [Rab17a], the module system [RK13], and the user interface [Rab14]. A previous system description [Rab13] focused on knowledge management features and is essentially disjoint from the present paper.

2 Language

Fig. 1 shows the core MMT language, where we omit the entire module system for simplicity, as well as a theory for LF, which starts off our running example.

$T : M = \Sigma$ defines a **theory** T with body Σ. M is an optional **meta-theory**; if M is absent, we call T a **foundation**. A typical use of MMT is three-tiered:

Theories	Θ	::=	$\cdot \mid \Theta,\ T : [T] = D^*$
Declarations	D	::=	$c[: E][= E][\#N] \mid \texttt{rule}\ E$
Expression	E	::=	$c \mid c(\Sigma; E^*) \mid \mathcal{L}^E(\text{string})$
Notation	N	::=	$(\mathcal{V}_n \mid \mathcal{A}_n \mid \text{string})^*$ [prec integer]

```
LFSyntax =
   type    # type
   kind    # kind
   Pi      # { V1 } A1
   lambda  # [ V1 ] A1
   apply   # A1  A2
   arrow   # A1 → A2
```

Fig. 1. Core MMT Grammar and LF Example

foundations define logical frameworks such as LF, which then serve as the meta-theories of logics such as FOL, which then serve as the meta-theories of the domain knowledge such as algebraic structures.

Constant declarations $c[: A][= t][\#N]$ introduce an identifier c with **type** A, **definiens** t, **notation** N, all of which are optional. In a **rule** declaration $\texttt{rule}\ E$, the expression E is interpreted as an object of the underlying programming language (Scala) evaluating to an instance of `Rule`. Sect. 3 describes how declaring rules can almost arbitrary customize the behavior of MMT.

In particular, rules are used to define the semantics of untyped constants. These are especially needed in foundations: because they have no meta-theory and MMT has no built-in constants, foundations can only declare untyped constants (as in `LFSyntax`). For example, we add about 10 rules (for type inference, β-reduction, etc., see [Rab17a] for details) to `LFSyntax` to obtain a theory for LF. By fixing LF as the meta-theory of all other theories, MMT can be used like an implementation of LF, at which point no further rules have to be provided. Users typically add more rules for two reasons: to define new logical frameworks; or to add a feature that the used logical framework cannot express.

A key insight to obtain foundation-independence was to fix the abstract syntax for **expressions** in a way that other languages can be seen as fragments. Besides identifiers c, only two constructors are used: Firstly, expressions $c(x_1 : A_1, \ldots, x_m{:}A_m; E_1, \ldots, E_n)$ subsume typical binding constructors and operators. **Notations** are used to relate these expressions to user-facing concrete syntax. In a notation, $\mathcal{V}_i$ indicates the position of the variable binding $x_i : A_i$, $\mathcal{A}_i$ the position of the argument E_i, and arbitrary strings can be used as delimiters; finally $\texttt{prec}\ I$ determines its precedence. For example, the notations in `LFSyntax` relate the abstract syntax $\texttt{lambda}(x : A; t)$ and $\texttt{arrow}(\cdot; A, B)$ to the concrete syntax $[x : A]t$ and $A \to B$. Secondly, literals $\mathcal{L}^A(l)$ represent arbitrary primitive values of type A with string representation l, e.g., $\mathcal{L}^{\texttt{nat}}(1)$ for the natural number 1. $\mathcal{L}^A(l)$ is rendered as the string l in concrete syntax, and special lexing rules must be provided for the inverse direction.

3 Implementation

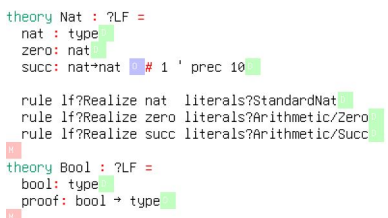
MMT processes input in **three phases**: parsing, checking, and elaboration. Each phase is split into the **structural** level for theories and declarations and the **ex-**

pression level. To maximize extensibility, the structural levels expose various plugin interfaces, and the expression level algorithms use the set of visible to the current theory. All 3×2 components are independent, and if multiple implementations for a component are available, MMT chooses based on, e.g., the file extension. For example, plugging in a new structure parser is an easy way to adapt the look and feel to a specific domain.

3.1 Parsing

Our running example `DHOL` starts in the screenshot on the right with LF-theories for natural numbers and booleans. The **structure parser** uses keywords at the beginning and special delimiters at the end of of declarations (shown as colored rectangles in the screenshot below). A plugin interface allows adding new keywords for arbitrary new declarations, whose semantics is defined by elaboration into the core language. We get back to this in Sect. 3.3.

The **expression parser** uses the available notations without hard-coding any special character. It performs multiple left-to-right passes, each one applying notations of a certain precedence. This complex design was necessary to support MMT's precedence-ordered mixfix notations where even the scope of bound variables is determined by notations.



To add natural number literals, our **example** uses the rule `lf?Realize`[1], which takes an MMT expression and a Scala object as arguments. In the first rule, the LF-type `nat` is tied to the Scala object `literals?StandardNat`, which defines the set of valid literals of type `nat`. In the other two rules, the LF-functions `zero` and `succ` are tied to corresponding Scala functions operating on these literals. In both cases, the Scala objects may supply additional rules: in our case (not shown in the screenshot), `StandardNat` supplies a lexing rule for natural numbers, and `Zero` and `Succ` provide unification rules. Together with the notation for `succ`, these rules enable MMT to parse the string `5'` into $\texttt{succ}(\cdot; \mathcal{L}^{\texttt{nat}}(5))$, compute it into $\mathcal{L}^{\texttt{nat}}(6)$, and unify $\mathcal{L}^{\texttt{nat}}(6)$ with $\texttt{succ}(\cdot; x)$.

3.2 Checking

$T \vdash A\ \texttt{inh}$	A is a type (is inhabitable)
$T \vdash t \Leftarrow A$	t checks against type A
$T \vdash t \Rightarrow A$	type of t infers to A
$T \vdash E \equiv E'$	E and E' are equal
$T \vdash E \rightsquigarrow E'$	E computes to E'
$T \vdash A <: A'$	A is subtype of A'

The structure checker implements the semantics of theories and the module system as described in [RK13]. Moreover, it determines which expression judgments must be verified when checking a constant declaration, e.g., the declaration $c : A$ can be added to theory T if $T \vdash A\ \texttt{inh}$.

[1] This example is simplified by omitting namespaces. All Scala objects can be found by searching for, e.g., `StandardNat` at https://uniformal.github.io/apidoc.

The expression parser inserts meta-variables for all subexpressions that are expected by the notations but not present in the input (such as implicit arguments and omitted variable types). The **expression checker** then uses the bidirectional type checking algorithm described in [Rab17a] to solve the meta-variables. It consists of 6 mutually recursive subalgorithms corresponding to the judgments in the table above.[2] Foundations are free to supply any rules for these judgments. In addition to applying these rules, the expression checker implements all the bureaucracy, which thus remains transparent to foundation developers. The bureaucracy includes the congruence closure of equality, lazy expansion of definitions, substitution and α-renaming, delaying constraints, error reporting, and backtracking as well as various optimizations such as caching for every term its free variables, inferred type, and whether it is normal. Notably, substitution introduces and preserves structure sharing, e.g., reducing $([x : A]f(x, x))\, a$ does not duplicate a and later calls only recurse into a once.

While LF only allows *typed* variables, we want shallow polymorphism in our `DHOL` **example**, i.e., *kinded* variables at the outside of expressions. Then we can declare the usual polymorphic infix equality operator of HOL as `equal` : $\{a : \texttt{type}\}a \to a \to \texttt{bool}$ $\#2 = 3$. We achieve this with the rule declaration `rule lf?ShallowPolymorphism`, whose (slightly simplified) Scala source is

```
object ShallowPolymorphism extends InhabitableRule(LF.Pi) {
 def apply(checker: Checker, tp: Term, C: context, H: History):Boolean={
  val LF.Pi(x,s,t) = tp
  checker.inferTypeAndThen(s, C, history + "inferring type") {si =>
   if (si != LF.type && si != LF.kind)
    solver.error("variable type must be a universe", H)
  } && checker.check(Inhabitable(C++x%s, t))
 }
}
```

Here checker is the instance of the expression checker proving the judgment $C \vdash tp\,\texttt{inh}$; and H is passed along to trace error messages. For example, if $tp = \{a : \texttt{type}\}a \to a \to \texttt{bool}$, this rule infers the type of $s = \texttt{type}$ to be $si = \texttt{kind}$, reports an error if si is neither `type` nor `kind`, and then continues checking $C, a : \texttt{type} \vdash t\,\texttt{inh}$ where $t = a \to a \to \texttt{bool}$ (which will succeed because t is now a plain LF-type). No further rules have to be added to LF to obtain shallow polymorphism.

3.3 Elaboration

The **expression elaborator** performs normalization by exhaustively applying computation rules. Because it is already called by the expression checker, it distinguishes between checked and unchecked expressions, where the former are simplified more efficiently.

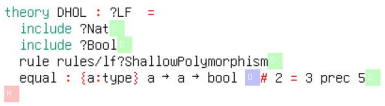
The **structure elaborator** computes for all declarations of the module systems as well as derived declarations provided by plugins their semantics in the core language. A typical use is to add derived declarations for inductive types and recursive functions.

[2] Subtyping is included here for completeness even though it is still under development.

In our **example**, we use a plugin to add derived declarations corresponding to the production $D ::= \texttt{rewrite}\, c : E$ to the structure parser, checker, and elaborator. Our plugin checks that E is of the form $\texttt{Pi}(\Gamma; \vdash l = r)$ and elaborates $\texttt{rewrite}\, c : E$ into the declarations $c : E$ and $\texttt{rule}\, R$. Here R is a `Rule` object built by the plugin that applies to the computation judgment $T \vdash e \rightsquigarrow e'$: it matches e against l with unification variables Γ and, if successful, computes e' by applying the appropriate substitution to r.

Together with this plugin, our `DHOL` foundation lets users inject arbitrary rewrite rules into the expression checker. This is exemplified in Sect. 3.5. An example of such a plugin in the MMT code is `lf?SimplificationRuleGenerator`.

3.4 Module System

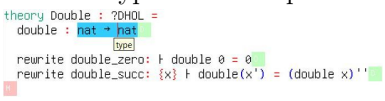


The MMT module system [RK13] is based on the category of theories and theory morphisms. Users can use theory morphisms to build large theories from small components and to translate between theories. Because foundations, logics, and domain knowledge are all represented as theories, the module system works uniformly for all levels. It is particularly useful for experimenting with language extensions: it allows encapsulating a set of rules in a theory so that multiple different rule sets can be used in different places of the same development. Moreover, developments based on one language can be easily carried over to an extended variant.

In our **example**, we build the DHOL foundation by merging the features developed so far, e.g., as in the screenshot on the right.

3.5 User Interface

The MMT user interface [Rab14] is realized as a plugin for the Java-based text editor jEdit. Besides the usual features (e.g., outline tree for the abstract syntax, tooltips, error highlighting, navigation), the UI includes several advanced features such as context-sensitive auto-completion (i.e., show only completions that return the expected type), interactive type inference, and change management. For example, the screenshot on the right applies our DHOL foundation to implement an inductive functions for doubling as a set of two rewrite rules (a feature inspired by Dedukti). The user has double-clicked on the arrow, which selected the surrounding subexpression and showed the inferred type as a tooltip.



Two features have proved critical for the UI development. Firstly, the MMT parser produces source references: every node in the syntax tree carries its file-line-column location in the input. Because of a subtlety of Scala, this information can be attached even to the immutable data structures for MMT expressions without interfering with pattern-matching or equality. That makes it easy to carry through source references during checking and elaboration. Secondly, the parse-check-elaborate pipeline can recover from almost

all errors. Thus, even declarations that caused errors are partially checked, which is invaluable to help users fix the errors.

The overall design is similar to Isabelle/jEdit except that MMT's programming language Scala can interface with Java seamlessly. Because MMT already exposes all functionality that is interesting for UIs, the glue code between MMT and jEdit comprises only a few hundred lines, and integrations with other frontends such as IntelliJ would be straightforward.

4 Libraries

MMT uses a decentralized library design, and the MMT repository does not include a single standard library. Instead, MMT provides adapters to retrieve libraries from, e.g., GitHub or MathHub [IJKW14], and manages them and their dependencies. Concrete libraries exist for logical frameworks, logics, and domain knowledge. MMT libraries use the OMDoc XML format [Koh06], which can be produced via the parse-check-elaborate pipeline or by importing other system's libraries.

Libraries have been imported among others from Twelf (where the author built a library of logics in LF) and various proof assistants such as Mizar, HOL Light, and PVS. In all cases, the logic of the other system (e.g., PVS in the case of [KMOR17]) is developed as an MMT foundation, which then serves as the meta-theory of all imported theories.

Libraries have been manually written in two lines of research. Firstly, the `urtheories` library[3] contains various foundations that can be used as logical frameworks. These are developed as modular features, most of which can be combined arbitrarily. Example features beyond our running example include sequence arguments, predicate subtypes, intersection types, records, quotation, and LLFP-style locks. Experimenting with such features was the original motivation of MMT, and many are under active development. Secondly, the OpenDreamKit project is using MMT to build a foundation for mathematics that can be used as a standard interface library for exchanging knowledge between mathematical systems; MMT has proved very appropriate for this long-term experimental process and now yields prototype implementations almost as fast as the foundation developer can conceive of potential features.

5 Conclusion and Future Work

Pushing further the ideas of logical frameworks, MMT was developed to explore the hypothesis that foundation-independent aspects of implementing formal systems dramatically outweigh foundation-specific ones, which would offer a huge potential for reuse if the right abstractions are found. This has been confirmed by all experiments so far: On the theoretical side, the author could generalize many important aspects such as type reconstruction to a foundation-independent

[3] `https://gl.mathhub.info/MMT/urtheories/`

description at the MMT level. On the practical side, the rules of the various foundations built in MMT comprise only a few hundred lines of code in contrast to tens of thousands of lines of foundation-independent code in the MMT system. The MMT system capitalizes on this effect and offers a rapid prototyping environment approach where developers can focus on the logical foundation and obtain a major implementation at extremely low cost.

The author conjectures that this effect also applies to features that have not been developed at the MMT level yet: Most importantly, many important theorem proving techniques such as substitution tree indices, tactic languages, machine learning–based premise selection, or external hammering systems have already been recognized as largely foundation-independent. But they have so far been mostly implemented in foundation-specific systems, partially because no good alternative was available. Similarly, it is promising to study the integration of logics with literate programming–style languages, computer algebra systems, or mathematical databases at the MMT level.

References

DGCT15. C. Dunchev, F. Guidi, C. Sacerdoti Coen, and E. Tassi. ELPI: Fast, Embeddable, lambda-Prolog Interpreter. In M. Davis, A. Fehnker, A. McIver, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 460–468, 2015.

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.

KMOR17. M. Kohlhase, D. Müller, S. Owre, and F. Rabe. Making PVS Accessible to Generic Services by Interpretation in a Universal Format. In M. Ayala-Rincon and C. Munoz, editors, *Interactive Theorem Proving*, pages 319–335. Springer, 2017.

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

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.

Rab14. F. Rabe. A Logic-Independent IDE. In C. Benzmüller and B. Woltzenlogel Paleo, editors, *Workshop on User Interfaces for Theorem Provers*, pages 48–60. Elsevier, 2014.

Rab17a. F. Rabe. A Modular Type Reconstruction Algorithm. *ACM Transactions on Computational Logic*, 2017. accepted pending minor revision; see https://kwarc.info/people/frabe/Research/rabe_recon_17.pdf.

Rab17b. F. Rabe. How to Identify, Translate, and Combine Logics? *Journal of Logic and Computation*, 27(6):1753–1798, 2017.

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