MMT: The Meta Meta Tool (system description)

Florian Rabe*

University Erlangen-Nuremberg

Abstract. Logical frameworks are meta-logics for defining other logics. MMT follows this approach but abstracts even further: it avoids committing to any primitive features like function types or propositions, resulting in a framework general enough to develop other logical frameworks in it.
Despite this high level of generality, it is possible to develop sophisticated results in MMT: The current release 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

Motivation Despite their 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 to help optimally allocating resources: They provide meta-logics in which the syntax and semantics of object logics can be defined, thus enabling, e.g., reasoning about object logics (Twelf [PS99], Abella [Gac08]) or generic proof assistants (Isabelle [Pau94]). Critically, they are invaluable for experimentation as 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, logical frameworks were extended with, e.g., concurrency (CLF [WCPW02]), reasoning about contexts (Beluga [PD10]), rewriting (Dedukti [BCH12]), and side conditions (LLFP [HLMS17]). Moreover, many logic-specific systems have investigated how to give users more freedom to change the system's behavior, e.g., via meta-programming (Idris [CB16], Lean [EUR+17]) or unification hints (Coq [GM08], Matita [ARCT09]).

Within this trend, MMT takes an extreme approach: it tries to systematically avoid any commitment to logical foundations, while still building a useful

* 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.

generic implementation. That makes Mmt flexible enough to implement other logical frameworks. Mmt is most similar to ELPI [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 (Scala) 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.

Contribution and Originality The originality of this system description may be debated as it does not present a recent, novel system. Instead, it summarizes over 10 years of experience in developing Mmt. Naturally, during this time, it was already presented in publications on, e.g., theoretical aspects [RK13,Rab17,Rab18], specific parts of the implementation [Rab15,Rab14], rapid prototypes developed on top of Mmt, e.g. [Rab19,MR19], or major research projects using Mmt [DIK+16,KR16].

However, there has never a direct system description about the Mmt system as a meta-logical framework. (A previous system description [Rab13] focused on knowledge management aspects and entirely predates the work presented here.) New ideas about logical frameworks generally require years of evaluation, and the author considers the present retroactive system description to be more valuable than any earlier descriptions would have been.

While Mmt comes with ready-to-use implementations of various logical frameworks, it is primarily a library that system developers can use to build new systems. Because Mmt abstracts from individual languages, its development brings universal aspects of language design into focus in a unique way. In particular, every Mmt feature must tease apart universal from language-specific aspects, and the interface between them makes the common expressivity-vs.-simplicity trade-off very clear. Therefore, the author expects this system description to be valuable in its own right benefiting the community in general and any researcher implementing a new language in particular.

Overview We describe the syntax and semantics of Mmt in Section 2 and 3, focusing on the internal syntax and the key design decisions at the implementation level. As a running example, we use Mmt to implement `LFM`, a variant of the logical framework LF [HHP93] modulo rewriting [CD07]. Implementation and documentation of Mmt are available at `https://uniformal.github.io/`.

2 Data Structures

Declarations and Expressions The Mmt syntax distinguishes strictly between two levels in a way that captures an almost ubiquitous distinction in implementations of formal systems. **Declarations** introduce a named object that can be

referenced in later declarations. Implementations fully process each declaration and register it imperatively in the kernel before parsing the next one.[1] On the other hand, **expressions** are anonymous, stored in pure data structures, and are processed using context-sensitive recursive functions.

Implementations of formal systems typically begin with a kernel defining the syntax of declarations and expressions. Over time, new kinds of declarations (module system, rewrite rules, etc.) and support tools (tactic language, user interface, etc.) are added. Thus, changes to the kernel syntax become increasingly difficult. MMT turns this process on its head by making only minimal assumptions about the kernel syntax so that all system components are forced to abstract from the specific choice of kernel syntax. MMT-based implementations of formal systems are MMT plugins that single out the desired fragment of declarations and expressions and defining their semantics.

MMT Declarations An MMT declaration consists of
- its local **name**,
- its optional **type**, which is an arbitrary expression,
- its optional **definiens**, which is an arbitrary expression,
- a **body**, which is a list of declarations with pairwise different local names,
- optional expressions like notations and metadata, which we omit here.

This makes MMT document a tree of named declarations, in which each node is decorated with some expressions (type, definiens, etc.) and has an ordered list of children (the body). The nesting of declarations yields a unique identifier for each declaration by qualifying with the list of local names of all ancestors. Two kinds of declarations are special in MMT:
- A **theory** declaration is of the form $T[: M] = \Sigma$. Here T is the name and Σ the body. The optional type M is called the meta-theory and explained in Sect. 4.
- A **constant** declaration is of the form $c[: A][= t][\#N]$. Here c is the name, A the type, t the definiens, and N the notation. Constants have no body and thus are the leafs of the declaration tree.

The combination of theories and constants is the simplest possible way to build a formal system. Essentially, constants are the payload, and theories group them into nested scopes. Constants subsume the various atomic declaration used in formal systems (e.g., function, predicates, and axioms/theorems in FOL; types, terms, and axioms/theorems in HOL; or universes and terms in the calculus of constructions), and theories subsume the various scoping constructs (Isabelle locales, Coq modules, PVS theories, etc.).

In addition to theories and constants, MMT plugins can implement Scala interfaces to single out arbitrary other special cases of declarations and give them a special semantics by elaborating them into other declarations. Declarations that have been defined in this way include inductive types (the body declares the type and constructors, and elaborations generated the induction schema),

[1] This happens also in LCF-style system like HOL Light, where this data structure is implicitly provided by the host language.

record types (the body declares the fields, and elaboration generates the type and constructors), and Coq-style sections (the body contains the section, and elaboration generates the quantified declarations).

Example 1. To define `LF` in MMT, all declarations are of the form $c : E$ where E is an LF-kind or an LF-type. Optionally, definiens and notation may be present. All of them are represented as typed constants. In particular, all LF-types and LF-kinds are represented as MMT expressions. We just have to declare special constants that construct these expressions, which we do in Ex. 2.

To extend `LF` to `LFM`, we build a plugin for MMT that defines rewrite declarations by implementing a specific Scala interface in a class R and registering R with `LFM`. R must implement abstract methods for parsing, checking, and elaborating that are called by the respective algorithms described in Sect. 3. The checking function defines the abstract syntax, and we define that a rewrite declaration must be of the form $c : E$ with E representing a directed equality (see also Ex. 2).

MMT Expressions The abstract syntax of MMT expressions E is given by

$$E ::= \ | \ g \mid c \mid C \mid g(E^*)$$

where $C ::= c[: A][= t][\#N]$ has the same syntax as a constant declaration. The role of the productions is as follows:

- g is a global identifier that refers to any declaration that is in scope. The scope of a declaration are all subsequent declarations and their bodies.
- c is a local name that refers to a local declaration C.
- C is a local declaration such as a variable declaration in a binder.
- $g(E^*)$ forms complex expressions as explained in Ex. 2.

Example 2. To build the *abstract* syntax of `LFM`-expressions, we declare five special constants in a special theory: `LFM` = `type`, `Pi`, `lambda`, `apply`, `rewrite`. All five consist just of a local name, without any of the optional components. We will add notations to them in Ex. 3 in order to define *concrete* syntax.

By qualifying with the theory name, we obtain the corresponding global identifiers. Now the complex expression `LFM`?`apply`(f, a) represents the `LFM`-function application $f\,a$, and `LFM`?`rewrite`(E, E') represents a directed equality. Binders are represented by using local declarations to introduce the variables, as in `LFM`?`lambda`$(x : A, t)$. The constant `type` takes no arguments and simply forms the expression `LFM`?`type`().

The precise treatment of scoping and α-renaming of local declarations is a subtle problem. Originally, MMT defined every local declaration to be subject to α-renaming and to be visible in all subsequent children of the complex expression. That is the right definitions for all variable-binding operators. But to represent, e.g, records `record`$(a : E, b : F)$, it must be possible to declare non-α-renamable local names a and b. Therefore, MMT now uses multiple local declaration types to fine-tune this behavior.

3 Algorithms

We restrict attention to the main processing chain of
- parsing: a declaration in concrete syntax (string) is converted into an MMT declaration, possibly with gaps for information that still needs to be inferred,
- checking: the declaration is type-checked and missing gaps are filed in,
- elaboration: the declaration is registered with the kernel and induced declarations are generated and registered.

MMT splits parser, checker, and elaborator into two components each for declarations resp. expressions. To maximize extensibility, the resulting 3×2 components are given as abstract classes with default implementations. For example, a plugin defining a new declaration parser is an easy way to adapt MMT's look and feel to a specific domain.

Parsing The declaration parser is a relatively simple one-pass parser that delegates to subparsers based on the initial keyword of each declaration. Special attention was paid to extensibility: MMT plugins can register new keywords and parse them into arbitrary declarations. The declaration parser delegates as necessary to the expression parser, which uses a much more complex design to handle precedence-ordered mixfix notations. The expression parser also inserts a fresh meta-variable for any component of the abstract syntax that is not found in the concrete syntax (e.g. omitted variable types, implicit arguments).

```
theory LFM =
  type    # type
  Pi      # Π V1,… . 2
  lambda  # λ V1,… . 2
  apply   # 1 %w…
  rewrite # 1 ⟿ 2
```

Example 3. To parse `LFM`-expressions, we add notations to the theory `LFM` as shown in the screenshot on the right. For example, the notation for `lambda` defines $\lambda x_1 : A_1, \ldots, x_n : A_n.B$ to be the concrete syntax for `LFM?lambda`$(x_1 : A_1, \ldots, x_n : A_n, B)$. In particular, `V1,...` says that the first argument is a comma-separated list of bound variable declarations. The notation for `apply` defines the concrete syntax for application to be a whitespace-separated list of all arguments.

To parse `LFM`-rewrite declarations, our class R must implement the parsing method of the interface for new declarations. However, the default implementation is already sufficient: it induces the concrete syntax **rewrite** $c : E$ for our rewrite declarations.

Checking The structure checker implements the semantics of theories and constants. This includes in particular the generation of appropriate typing judgments that are delegated to the expression checker. For a constant declaration $c : A$, this is simply the check whether A is inhabitable, i.e., whether A may occur as the type of a constant.

The details of the object checker have already been described in [Rab18]. It uses the identifier g in complex expressions $g(E^*)$ to delegate judgments to specific rules. A rule is a self-contained Scala snippet provided by an MMT plugin

that is internally stored like a declaration in a theory. When invoking the object checker, MMT passes all rules in scope. This allows different type systems to co-exist and interact in MMT.

Example 4. We already indicated in Ex. 1, how to check `LFM`-rewrite declarations **rewrite** $c : E$. It remains to check that E is indeed a well-typed directed equality.

Concretely, we need to accept declarations with concrete syntax $c : \Pi\vec{d}.l \rightsquigarrow r$ (where d declares the free variables of l and r) iff l and r have the same type. When checking the inhabitability of the type of c, the object checker repeated delegates to the inhabitability rule for `Pi` until it reaches the judgment that $l \rightsquigarrow r$ is inhabitable. This would fail because there is no inhabitability rule yet for the expression constructed from `rewrite`. In our plugin, we implement such a rule using the following (slightly simplified) Scala code:

```
object RewriteRule extends InhabitableRule(LFM.rewrite) {
 def apply(checker: Checker, E: Expression, C: context): Boolean = {
  val LF.rewrite(l,r) = E
  val lT = checker.inferType(l,C)
  val rT = checker.inferType(r,C)
  checker.checkEqual(lT,rT,C)
 }
}
```

Here `checker` is the current instance of the object checker, which provides callbacks for recursing into other judgments. We use these to infer the MMT-types of l and r and check that they are equal.

Elaboration The declaration elaborator delegates any declaration that is not a constant or a theory to the respective plugin. That plugin then has full freedom to generate additional declarations or rules, or flag errors.

Example 5. The declaration elaborator calls the elaboration method of our class R from the `LFM`-plugin on any declaration rewrite declaration **rewrite** $c : \Pi\vec{d}.l \rightsquigarrow r$. We implement it create a Scala object that implements a simplification rule s that performs that matches its input against l and rewrites it to r. It then injects s into the current theory right after the declaration of c. Thus, s is in scope for any future invocations of the object checker, which uses s to normalize expressions just like it uses the `LF`-rule for β-conversion.[2]

4 Content

A final, critical feature of MMT is the use of meta-theories: every theory may carry as its type a reference to some other theory. Typical MMT developments use two levels of meta-theories, which inspired the name *meta-meta-tool*: logical frameworks F are implemented via MMT plugins and use no meta-theory. `LFM` is one such case. Logics L are defined declaratively in a logical framework F, which

[2] Like in the Dedukti [BCH12] implementation of [CD07], it remains the users obligation to ensure termination of the rewrite system.

serves as the meta-theory. And the payload content C is formalized as theories with meta-theory L. Only the F-levels requires writing Scala plugins — authors of theories with meta-theory usually see no Scala code because the meta-theory defines the semantics.

At the F-level, the main MMT library[3] implements a variety of freely combinable language features including, e.g., dependent function types, predicate subtypes, intersection types, records, quotation, and LLFP-style locks. Our `LFM`-example adds rewrite rules as one such feature. At the L-level, the LATIN library[4] defines a suite of logics in LF-based frameworks.

Additionally, after defining the respective logics, C-level libraries have been imported automatically from a variety of proof assistants, including the libraries of HOL Light in [KR14], Mizar in [IKRU13], PVS in [KMOR17], Coq in [MRS19], and Isabelle [KRW20]. Being able to represent these advanced logics was one of the original motivations of MMT, and experience has shown that it was critical to be able to flexibly extend and experiment with the logical frameworks as MMT now allows.

```
theory Test : LFM =
  nat  : type
  zero : nat
  succ : Πx:nat.nat          # 1 '
  plus : Πx:nat,y:nat.nat    # 1 + 2
  rewrite plus_zero : Πx.x+0 ⟿ x
  rewrite plus_succ : Πx,y.x+(y') ⟿ (x+y)'
```

Example 6. To write `LFM`-content we create new MMT theories with meta-theory `LFM`. The screenshot on the right gives as an example the addition of natural numbers defined by rewriting. If our plugin for `LFM` is loaded, future uses of addition are automatically simplified.

5 Conclusion and Future Work

MMT explores how much logic implementation development can be relegated to a logic-independent framework. If the right abstractions are found, this offers a huge potential for reuse, and MMT can be seen as a running experiment to find these. Both on the theoretical and practical side, this has so far been confirmed by all experiments.

The author conjectures that this effect will also hold true for advanced features that have not been attacked within MMT yet. Most importantly, MMT offers a platform for the logic-independent implementation of many important theorem proving techniques such as substitution tree indexing, tactic languages, machine learning–based premise selection, or external hammering systems. These have already been recognized as largely logic-independent but have so far mostly been implemented in logic-specific systems.

[3] https://gl.mathhub.info/MMT/urtheories/
[4] https://gl.mathhub.info/MMT/LATIN2/

References

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.

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.

CB16. D. Christiansen and E. Brady. Elaborator reflection: extending idris in idris. In J. Garrigue, G. Keller, and E. Sumii, editors, *International Conference on Functional Programming*, pages 284–297. ACM, 2016.

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.

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.

DIK+16. P. Dehaye, M. Iancu, M. Kohlhase, A. Konovalov, S. Lelièvre, D. Müller, M. Pfeiffer, F. Rabe, N. Thiéry, and T. Wiesing. Interoperability in the OpenDreamKit Project: The Math-in-the-Middle Approach. In M. Kohlhase, L. de Moura, M. Johansson, B. Miller, and F. Tompa, editors, *Intelligent Computer Mathematics*, pages 117–131. Springer, 2016.

EUR+17. G. Ebner, S. Ullrich, J. Roesch, J. Avigad, and L. de Moura. A Metaprogramming Framework for Formal Verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):34:1–34:29, 2017.

Gac08. A. Gacek. The Abella interactive theorem prover (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning*, pages 154–161, 2008.

GM08. G. Gonthier and A. Mahboubi. A Small Scale Reflection Extension for the Coq system. Technical Report RR-6455, INRIA, 2008.

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.

HLMS17. F. Honsell, L. Liquori, P. Maksimovic, and I. Scagnetto. LLFP: a logical framework for modeling external evidence, side conditions, and proof irrelevance using monads. *Logical Methods in Computer Science*, 13(3), 2017.

IKRU13. M. Iancu, M. Kohlhase, F. Rabe, and J. Urban. The Mizar Mathematical Library in OMDoc: Translation and Applications. *Journal of Automated Reasoning*, 50(2):191–202, 2013.

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.

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

KR16. M. Kohlhase and F. Rabe. QED Reloaded: Towards a Pluralistic Formal Library of Mathematical Knowledge. *Journal of Formalized Reasoning*, 9(1):201–234, 2016.

KRW20. M. Kohlhase, F. Rabe, and M. Wenzel. Making Isabelle Content Accessible in Knowledge in Representation Formats. see https://kwarc.info/people/frabe/Research/KRW_isabelle_19.pdf, 2020.

MR19. D. Müller and F. Rabe. Rapid Prototyping Formal Systems in MMT: Case Studies. In D. Miller and I. Scagnetto, editors, *Logical Frameworks and Meta-languages: Theory and Practice*, 2019. To appear.

MRS19. D. Müller, F. Rabe, and C. Sacerdoti Coen. The Coq Library as a Theory Graph. In C. Kaliszyk, E. Brady, A. Kohlhase, and C. Sacerdoti Coen, editors, *Intelligent Computer Mathematics*, pages 171–186. Springer, 2019.

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

PD10. B. Pientka and J. Dunfield. Beluga: A Framework for Programming and Reasoning with Deductive Systems (System description). In J. Giesl and R. Hähnle, editors, *Automated Reasoning*, pages 15–21. Springer, 2010.

PS99. F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Automated Deduction*, pages 202–206, 1999.

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.

Rab15. F. Rabe. Generic Literals. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge, editors, *Intelligent Computer Mathematics*, pages 102–117. Springer, 2015.

Rab17. F. Rabe. How to Identify, Translate, and Combine Logics? *Journal of Logic and Computation*, 27(6):1753–1798, 2017.

Rab18. F. Rabe. A Modular Type Reconstruction Algorithm. *ACM Transactions on Computational Logic*, 19(4):1–43, 2018.

Rab19. F. Rabe. MMTTeX: Connecting Content and Narration-Oriented Document Formats. In C. Kaliszyk, E. Brady, A. Kohlhase, and C. Sacerdoti Coen, editors, *Intelligent Computer Mathematics*, pages 205–210. Springer, 2019.

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

WCPW02. K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A Concurrent Logical Framework I: Judgments and Properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002.