Rapid Prototyping Formal Systems in MMT: 5 Case Studies

Dennis Müller

Computer Science, FAU Erlangen-Nürnberg, Germany

d.mueller@kwarc.info (iD)

Florian Rabe

Computer Science, FAU Erlangen-Nürnberg, Germany

LRI Paris, France

florian.rabe@fau.de (iD)

Logical frameworks are meta-formalisms in which the syntax and semantics of object logics and related formal systems can be defined. This allows object logics to inherit implementations from the framework including, e.g., parser, type checker, or module system. But if the desired object logic falls outside the comfort zone of the logical framework, these definitions may become cumbersome or infeasible.

Therefore, the MMT system abstracts even further than previous frameworks: it assumes no type system or logic at all and allows its kernel algorithms to be customized by almost arbitrary sets of rules. In particular, this allows implementing standard logical frameworks like LF in MMT. But it does so without chaining users to one particular meta-formalism: users can flexibly adapt MMT whenever the object logic demands it.

In this paper, we present a series of case studies that do just that, defining increasingly complex object logics in MMT. We use elegant declarative logic definitions wherever possible, but inject entirely new rules into the kernel when necessary. Our experience shows that the MMT approach allows deriving prototype implementations of very diverse formal systems very easily and quickly.

1 Introduction and Related Work

Motivation 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** provide meta-logics in which the syntax and semantics of object logics can be defined. They are invaluable for **experimentation**: they allow rapidly prototyping formal systems, often to the extent of reducing the design-implement-scale process from person-years to person-days, and thus can speed up the feedback loop by orders of magnitude.

Features realized generically in logical frameworks include reasoning about object logics (Twelf [17], Abella [9]), interactive theorem proving (Isabelle [16]), concurrency (CLF [24]), reasoning about contexts (Beluga [18]), rewriting (Dedukti [2]), side conditions (LLF$_\mathscr{P}$ [12]), or integration with proof assistants (Hybrid [8]). Moreover, many logic-specific systems are investigating how to allow users to experiment with system behavior inside the system, e.g., via meta-programming (for Idris in [4], for Lean in [7]) or unification hints (for Coq in [10], for Matita in [1]).

Within this group, MMT [21] takes an extreme approach: it tries to systematically avoid any commitment to logical foundations while still allowing useful generic results [22, 20, 19]. All kernel algorithms of MMT are parametrized by a set of rules, which are programmed in the underlying programming language (Scala). The set of rules to use is collected from the current context so that entirely different kernel behaviors can coexist in the same development. This makes MMT so flexible that other logical frameworks can be conveniently developed inside it. (This gave rise to the name MMT as an abbreviation of meta-meta-theory/tool.)

This design makes MMT most similar to the ELPI framework [6]. 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 rules. Thus, it offers more freedom for rule definitions than ELPI at the price of disconnecting syntax and rules. This freedom allows users fine-grained control over, e.g., error-reporting or side conditions, and enables the use of support tools like debuggers and IDEs, but makes it harder to reason about the rules.

Overview and Contribution   Our objective is to evaluate the usability of MMT by conducting several rapid prototyping experiments in it. After sketching the basics of MMT language in Sect. 2, using the dependent type theory LF [11] as a "hello-world example", the subsequent sections describe one case study each that extends LF with increasingly complex features: dependently-typed higher-order logic (DHOL) in Sect. 3, Dedukti-style rewriting in Sect. 4, LLF$_\mathscr{P}$ types as described in [12] in Sect. 5, linear logic as defined by the resource semantics in Sect. 6, and homotopy type theory (HoTT) in Sect. 7.

Our main contribution is to answer positively the question whether it is possible to build a framework that allows prototyping such a variety of features both elegantly and rapidly.

Regarding elegance: all prototypes involve only little non-declarative parts (i.e., rules programmed in Scala), and that code is orders of magnitude shorter and arguably easier to understand than a from-scratch implementation would be.

Regarding rapidness, we have tried to estimate the time effort for each case study. DHOL took a few person-hours, rewriting about one person-week (distributed over several years), LLF$_\mathscr{P}$ took two evenings, linear logic took two days (building on an existing failing case study in pure LF). These were done by Rabe, who also built MMT. HoTT, combining various typing features, was done by Müller as a PhD student and in its first development – with little to no prior experience with either MMT or type theories — took several weeks over the course of roughly a year. A recent complete reimplementation was done in a few afternoons.

Each case study is published here for the first time (some were summarized without details in [21]), and some present novel contributions in themselves. For example, to the best of our knowledge no implementation previously existed for LLF$_\mathscr{P}$.

We are not quite able to answer the question about the limitations of the approach: we chose our case studies in increasing order of difficulty, and they have all succeeded. But we speculate on the limitations and future challenges in Sect. 8.

2   Preliminaries

The syntax, type system, and typing algorithms of MMT have been described in detail in [21]. In the sequel, we give a quick example-driven introduction to MMT that builds a prototype implementation of the dependently-typed λ-calculus LF [11].

Theory			$T[:T] = D^*$	LF $=$		
Declarations	D	$::=$	$c[:E][=E][\#N] \mid \text{rule } E$	type	#	type
Expression	E	$::=$	$x \mid c(\Gamma; E^*)$	kind	#	kind
Context	Γ	$::=$	$\cdot \mid \Gamma, x : E$	Pi	#	$\{V1\}1$
Notation	N	$::=$	$(Vn \mid n \mid \text{string})^* [\text{prec } n]$	lambda	#	$[V1]1$
Number	n			apply	#	$1\ 2$
Identifier	T, c, x			arrow	#	$1 \rightarrow 2$

Figure 1: Core MMT Grammar (left) and LF Example (right)

MMT uses no built-in symbols like λ or $\rightarrow$. Prototyping a formal system S in MMT means to define one MMT theory S that declares all the logical primitives of S such as universes, base types, connectives, binders, inference rules, axioms, etc. Once such a meta-theory is fixed, MMT induces an implementation of S, including, e.g., parser, type reconstruction, module system, and IDE.

Fig. 1 shows a simple fragment of MMT's **grammar** that is sufficient for our purposes as well as an example theory for the logical primitives of LF. An MMT **theory** declares **constants** $c[:A][=t][\#N]$, which introduce the identifier c with optional **type** A, **definiens** t, and **notation** N. In a **notation**, Vi indicates the position of a variable binding $x_i : A_i$, i the position of an argument E_i, and arbitrary strings can be used as delimiters; finally $\text{prec } I$ determines its precedence.

LF introduces constants for the syntax of LF and their notations, e.g., the notation for `lambda` introduces λ-abstractions with the concrete syntax to be $[x : A]t$. Internally, **expressions** are represented as $c(x_1:A_1, \ldots, x_m:A_m; E_1, \ldots, E_n)$. Here c is the constant forming the term, the $x_i : A_i$ are variable bindings, and the E_i are arguments, e.g., $[x : A]t$ corresponds to $\text{lambda}(x : A; t)$, and $f\, a$ (for LF function application) to $\text{apply}(\cdot; f, a)$. Each x_i is bound and may occur in $A_{i+1}, \ldots, E_n$. α-renaming and capture-avoiding substitution are defined in the usual way.

We can already use LF to define other languages, e.g., as in Listing 1.[1] Here, $PL : LF = \Sigma$ defines the theory PL (for propositional logic) with **meta-theory** LF, and body Σ. Apart from LF being an MMT theory itself, this definition of propositional logic in LF is standard. Note how the constants `andI` and `impI` (for proof rules) have notations that do not list all their arguments — the other arguments are implicit and have to be reconstructed. Similarly, omitted types of bound variables are reconstructed.

Listing 1: Fragment of an MMT Theory for propositional logic

```
theory PL : LF =
        prop : type
        ded : prop → type  #  ⊢1 prec −5
        and : prop → prop → prop  #  1 ∧2 prec 15
        impl : prop → prop → prop  #  1 ⇒2 prec 10
        equiv : prop → prop → prop  #  1 ⇔2 prec 10
            = [x,y] (x ⇒y) ∧(y ⇒x)
        andI : {A,B} ⊢A → ⊢B → ⊢A ∧B  #  andI 3 4
        impI: {A,B} (⊢ A → ⊢B) → ⊢A ⇒B  #  impI 3

        equivI : {A,B} (⊢ A → ⊢B) → (⊢ B → ⊢A) → ⊢A ⇔B
            = [A,B,p,q] andI (impI [x] p x) (impI [x] q x)
```

[1] from file `https://gl.mathhub.info/MMT/examples/blob/devel/source/logic/pl.mmt`

In $T : M = \Sigma$, the purpose of the **meta-theory** M is to induce the syntax and semantics of Σ. All constants and notations of M are available to form the types and definitions of the constants in Σ. And the rules of M define how Σ is type-checked. To understand how to supply rules for LF, we need a little more background about the MMT type system.

Judgment	Intuition
All judgments are relative to fixed theory Θ and its meta-theories.	
$\Gamma \vdash T \; \texttt{inh}$	T is inhabitable (may occur as the type of a constant)
$\Gamma \vdash t \Leftarrow T$	t checks against inhabitable term T.
$\Gamma \vdash t \Rightarrow T$	the principal type of term t is inferred to be T
$\Gamma \vdash t_1 \equiv t_2 \; : T$	t_1 and t_2 are equal at type T
$\Gamma \vdash t_1 \rightsquigarrow t_2$	t_1 computes to t_2 (preserving type)

Figure 2: Judgments

The **judgments** are given in Figure 2. For simplicity, we assume that a set of theory definitions has been fixed and that all judgments are relative to one theory Θ, without making this explicit in the notation. No constant-specific rules are built in: the only rules fixed by MMT are lookup rules (e.g., to infer the type of a variable x), equivalence and congruence of equality, and α-renaming of bound variables. The four judgments for equality and typing are standard for bidirectional type-checking. The unusual judgment $\Gamma \vdash T \; \texttt{inh}$ is used to check constant declarations: the declaration $c : T$ is allowed if $\Gamma \vdash T \; \texttt{inh}$, and the declaration $c : T = t$ additionally requires $\Gamma \vdash t \Leftarrow T$.

Finally, we add **rule declarations** $\texttt{rule}\, E$ to the theory LF. Here the expression E is interpreted as an object of the underlying programming language (Scala). When checking a judgment, MMT uses only those rules that are visible to the current context. Each rule is a self-contained Scala object that is loaded dynamically when needed. When prototyping systems in MMT, users usually use a logical framework like LF and only define declarative, statically checked rules like we did for PL above. But, critically, users have the option to add a Scala-based rule whenever necessary.

Each rule implements a specific Scala interface. For example, rules implementing *InferenceRule* are used whenever the judgment $t \Rightarrow T$ is encountered; this interface includes a method *inferType* that receives the current context and t and must try to return T. It also receives callbacks to recursively check the premises of the rule. Listing 2 shows an example rule inferring the type of a lambda term.[2]

Listing 2: A Rule Inferring the Type of a Lambda Term

```scala
object LambdaTerm extends InferenceRule(Lambda.path) {
   def apply(solver: Solver)(tm: Term, covered: Boolean)(implicit stack: Stack, history: History)
    : Option[Term] = tm match {
      case Lambda(x,a,t) =>
         if (!covered) isTypeLike(solver,a)
         val (xn,sub) = Common.pickFresh(solver, x)
         solver.inferType(t ^? sub, covered)(stack ++ xn % a, history) map {b => Pi(xn,a,b)}
      case _ => None // should be impossible
   }
}
```

---

[2]in file `https://github.com/UniFormal/MMT/blob/devel/src/mmt-lf/src/info/kwarc/mmt/lf/Rules.scala`

For LF, we use 5 inference rules applicable to different terms: one each for terms formed from `type`, `kind`, `Pi` (or arrow), `lambda`, and `apply`. Additionally, we define 2 rules that check $t \Leftarrow T$ and $t \equiv t' : T$ whenever T is a `Pi`-expression, and a computation rule for β-reduction. For type-reconstruction, MMT uses some variables to represent omitted subexpressions that must be reconstructed. For LF, only one additional rule is needed that is aware of these meta-variables — a special rule for pattern unification.

Together with two rules for inhabitability, our LF prototype requires 11 Scala rules, taking about 200 lines of Scala code. This is a tiny fraction compared to the logic-independent code in MMT that builds a full-fledged application from these 11 rules. Moreover, these 200 lines that carry the semantics of LF are all in one place and much easier to read and verify than a from-scratch implementation.

3 Dependently-Typed Higher-Order Logic

> By adding a single new typing rule to LF, we enable shallow polymorphism. This allows developing dependently-typed higher-order logic.

LF users have often lamented the lack of polymorphism to define, e.g., polymorphic equality, lists, product types, etc. An important observation is that shallow polymorphism, i.e., the binding of kinded variables but only at toplevel, is much simpler than full polymorphism but already enables many important applications.

To enable shallow polymorphism, we extend our LF prototype with a single rule[3] for the judgment $\{x : A\}B$ inh. This rule infers the type of A, checks that it is a universe (i.e., `type` or `kind`) and then recursively checks that B is inhabitable. Because the judgment T inh is only called at toplevel, this yields exactly the desired effect. Maybe surprisingly, except for a few minor adaptations, all existing rules of LF can remain unchanged, e.g., LF's Scala rule to check $t \Leftarrow \{x : A\}B$ worked immediately if A is a kind.

We call the resulting logical framework PLF. Using the MMT module system, it is defined by including LF and adding one rule declaration.[4]

The resulting case study is shown in Listing 3. Starting with polymorphic equality, we can define higher-order logic in the usual way, including the definition of polymorphic quantifiers and their proof rules. Note how kinded variables are treated in the same way as typed ones because MMT does not care about the difference anyway. Similarly, type reconstruction of variable kinds and implicit type arguments works out of the box: the new Scala rule does not even mention reconstruction.

All declarations are straightforward except for the congruence rule `cong`. We stated it here for simply-typed functions only due to the inherent difficulties of combining dependent types with equality. But this is a well-known theoretical problem and not an MMT issue. In fact, MMT can now help with the problem: It is often unclear how well a potential solution for this deep problem works in practice. By building variants of `DHOL`, possibly with additional Scala rules to handle equality specially, we can quickly prototype and evaluate possible solutions.

Listing 3: An MMT Theory for Dependently-Typed HOL

```
theory DHOL : PLF =
  bool : type
  equal : {A:type} A → A → bool # 2 ≐ 3 prec 5
  ded : bool → type # ⊢1 prec −5
```

---

[3]in file `https://github.com/UniFormal/MMT/blob/devel/src/mmt-lf/src/info/kwarc/mmt/lf/ShallowPolymorphism.scala`

[4]in file `https://gl.mathhub.info/MMT/urtheories/blob/devel/source/primitive_types/bool.mmt`

```
refl : {A,X:A} ⊢X ≐X # refl %I1 %I2
cong : {A, B: type} {F: A → B} {X, Y: A} ⊢X ≐Y → ⊢(F X) ≐(F Y) # cong 3 6

extensionality : {A:type,B:A → type}{F:{x:A} B x, G:{x:A} B x} ({x: A} ⊢F x ≐G x) → ⊢F ≐G
   # ext 5
boolEqIntro : {B1,B2:bool} (⊢ B1 → ⊢B2) → (⊢ B2 → ⊢B1) → ⊢(B1 ≐B2) # beqI 3 4
boolEqElim : {B1,B2:bool} ⊢B1 ≐B2 → ⊢B1 → ⊢B2 # beqE 3 4
symmetry : {A : type}{a,b : A} ⊢a ≐b → ⊢b ≐a
   = [A][a,b][p] (cong ([x] x ≐a) p) refl # symm 4
eqFun : {A : type,B : type}{F,G : A → B} ⊢F ≐G → {a} ⊢F a ≐G a
   = [A,B][F,G][p][a] cong ([H: A → B] H a) p # eqfun 5 6

true : bool = ([x:bool] x) ≐([x:bool] x)
trueI : ⊢true = refl
forall : {A:type} (A → bool) → bool = [A,P] P ≐([x] true) # ∀2
forallI : {A:type, P : A → bool}({x} ⊢P x) → ⊢∀[x] P x
   = [A,P][p] ext ([x: A] beqI ([pf: ⊢P x] trueI) ([pt: ⊢true] p x))
forallE : {A:type, P : A → bool} (⊢ ∀[x]P x) → {a} ⊢P a
   = [A,P][p][a] beqE (eqfun (symm p) a) trueI
```

4   LF Modulo Rewriting

> We build an MMT plugin that interprets certain declarations as rewrite rules and adds them on the fly. This yields LF modulo rewriting akin to Dedukti.

LF modulo rewriting [5] as implemented in Dedukti [2] was recently identified as a sweet spot in logical framework design — a simple feature with huge practical benefit. Inspired by this, we want to extend our LF prototype to LF modulo.

In LF, a rewrite rule can be seen as an LF constant $c : \{\Gamma\}l \doteq r$, which rewrites l to r in context Γ. (For simplicity, we reuse the equality predicate $\doteq$ from Section 3.) In plain LF, this is just a normal constant that has no bearing on computation. In LF modulo, this rule must result in an additional computation rule $\Gamma \vdash l \rightsquigarrow r$.

This naturally leads to the main idea of our prototype study: we build an MMT plugin that automatically adds a rule declaration when a constant with an appropriate type is encountered. For this we use some machinery that is available to MMT plugins but that we have not mentioned in Section 2.

Firstly, plugins can listen to MMT events. Whenever a constant c is added, our plugin inspects it and possibly adds a new computation rule declaration to MMT's internal data structures. The rule declaration is added right after the declaration of c so that it is in scope whenever c is.

Secondly, like in Dedukti, we do not check confluence or termination so that the user needs fine-grained control over which declarations induce rewrite rules. For that, we exploit that every MMT declaration may carry metadata. Most importantly, it provides convenient syntax to attach a *role* to each constant. The role has no effect unless plugins choose to act on it. Our plugin picks up on the role `Simplify` — any constant with this role becomes a rewrite rule.

The implementation of our plugin – including reusable abstractions for matching – takes about 450 lines of code[5]. To evaluate it, we formalize Σ-types. In general, there are three different ways to do that:

- As new primitive features akin to how we defined Π-types in LF. But that requires new Scala rules.

---

[5]in   file   `https://github.com/UniFormal/MMT/blob/devel/src/mmt-lf/src/info/kwarc/mmt/lf/`
`SimplificationRuleGenerator.scala`

- By declaring an object logic inside LF. But that does not allow building pairs of arbitrary LF types.

- By defining polymorphic Σ-types in PLF.

All three ways have their merit, and the latter two can make use of rewriting in essentially the same way: they turn the two β-style rules into rewrite rules $\mathtt{pi}_i(\mathtt{pair}\,x_1\,x_2) \rightsquigarrow x_i$. Here we choose the third way, both for the sake of example and because it is a neat trade-off between the other two options. The resulting formalization is given in Listing 4[6].

Listing 4: Σ-types defined in dependently-typed HOL

```
theory Sigma : DHOL =
    Sigma : {A:type} (A → type) → type # Σ2
    pair : {A : type, B: A → type, a : A } B a → ΣB # pair 3 4
    pi1 : {A : type, B: A → type} ΣB → A # pi1 3
    pi2 : {A : type, B: A → type} {p : ΣB} B (pi1 p) # pi2 3

    conv_pair : {A: type, B: A → type, u: ΣB} ⊢u ≐(pair (pi1 u) (pi2 u))
    conv_pi1 : {A : type, B: A → type}{a : A, b: B a} ⊢(pi1 (pair A B a b)) ≐a role Simplify
    conv_pi2 : {A : type, B: A → type}{a : A, b: B a} ⊢(pi2 (pair A B a b)) ≐b role Simplify
```

All declarations are straightforward except for `conv_pi2`. In plain LF, this declaration does not type-check: the two sides of the equality have different types, namely `B (pi1 (pair A B a b))` on the left and `B b` on the right. But in LF modulo, it type-checks because `conv_pi1` is registered as a rewrite rule.

The combination of shallow polymorphism and rewriting is an amazingly powerful extension while retaining the look-and-feel of LF. For example, it allows defining polymorphic lists with their induction principles and rewrite rules to compute recursive functions. The separation of concerns that MMT introduces allowed us to build this extension of LF within days without changing any line in MMT itself.

5   External Side Conditions

The previous case studies considered well-understood extensions of LF, which makes prototyping simpler. Therefore, the next case study considers an experimental formal system that was introduced only recently, as a theoretical result without mature implementation. Such new systems abound, and it is here where rapid prototyping is most urgently needed.

> Lock types were introduced in [12] to allow for arbitrary external side conditions in LF. Using MMT, we obtain a prototype implementation within a few hours.

The central idea of $\mathrm{LLF}_{\mathscr{P}}$ is to extend LF with a monadic type operator $\mathscr{L}^K$ that is parametrized by an arbitrary extra-linguistic side condition K, which is of the form $p(t,A)$ for a term $t \Leftarrow A$ and an identifier p. Within the monad, K can be assumed at no cost. But to access a monadic value from the outside, the side condition has to be evaluated.

Our case study represents $\mathrm{LLF}_{\mathscr{P}}$ and applies it to one of the examples described in [12] — a call-by-value reduction of the λ_v calculus. Here the side condition is of the form $K = \mathtt{Val}(t,\mathtt{term})$, and $\mathtt{Val}$ checks that t is a λ_v-value (i.e., a variable or a λ-abstraction). Such syntactic side conditions are quite common but cannot be formalized in LF because they are not preserved by substitution.

We formalize the syntax of $\mathrm{LLF}_{\mathscr{P}}$ in theory LLFP in Listing 5: we include LF and extend it with one constant each for the formation, introduction, and elimination form of the new type constructor. We only

---

[6]in file `https://gl.mathhub.info/MMT/examples/blob/devel/source/sigma.mmt`

need to write 6 rules in Scala that require roughly 100 lines of code. Additionally, we introduce a new abstract rule interface ExternalConditionRule for rules that check our side conditions. ExternalConditionRule is a new abstract interface, and such rules can evaluate K in any way they want. Their behavior is as follows: (i) The formation rule infers $\mathscr{L}\,K\,\langle A\rangle \Rightarrow \texttt{type}$ from $A \Rightarrow \texttt{type}$. (ii) The introduction rule infers $L\,K\,\langle t\rangle \Rightarrow \mathscr{L}\,K\,\langle A\rangle$ from $t \Rightarrow A$. Both rules use the flexibility of MMT expressions to put the assumption K into the context — something that plain LF would not allow because K is not a well-typed LF expression. (iii) The elimination rule infers $U\,\langle t\rangle : A$ if $t : \mathscr{L}\,K\,\langle A\rangle$ and if K holds. This rule is called in InferUnlock in Scala, and a fragment of its code is given in Figure 6. It establishes K of the form $p(t,A)$ in one of two ways: if K is in the context (i.e., we are in the monad), it is assumed; otherwise, we look up an ExternalConditionRule for p and call it on K. (iv-vi) The remaining 3 rules (check against monadic type, compare equality at monadic type, reduce unlock of lock) are straightforward.

The λ_v case study is shown in the bottom part of Listing 5[7]. We declare the side condition Val as an untyped constant and a rule ValRule of type ExternalConditionRule. The Scala code of ValRule is given in Listing 6[8]. It checks the condition $\texttt{Val}(t,a)$, i.e., that t is of the form $\texttt{free}\,n$ or $\lambda\,F$; if that fails, it registers an error message that the IDE will show to the user.

The remainder then follows [12] in a straightforward way. We added the declaration check2 and fail, where InferUnlock calls ValRule. This succeeds in the first case because the argument 'n, which appears as LF.Apply(CallByValue.free, n) in the Scala code, is a λ_v-value; it (correctly) fails in the second case where the argument is not a λ_v-value.

Listing 5: An MMT Theory for the Syntax of Lock Types

```
theory LLFP =
    include LF
    locktype # 𝓛 1 ⟨ 2 ⟩ // type formation: 𝓛K is the monadic operator
    lockterm # L 1 ⟨ 2 ⟩ // introduction form
    unlock # U ⟨ 1 ⟩ // elimination form
    // 6 rules omitted

theory CallByValueExample : ?LLFP =
    term : type
    app : term → term → term # 1 @ 2 prec 50
    lam : (term → term) → term # λ1
    ... // (natural numbers omitted)
    free : nat → term # ' 1 prec 100

    eq : term → term → type # 1 ≐2
    eq_app: {M,N,X,Y} M ≐N → X ≐Y → M@X ≐N@Y # eq_app 5 6
    ... // (additional equality judgment and its rules omitted)

    Val
    rule rules?ValRule

    // reduction rules using Val condition
    betav : {M,N} 𝓛 (Val N term) ⟨(λ M)@N≐(M N)⟩ # betav 1 2
    csiv : {M,N}({x} 𝓛 (Val x term) ⟨(M x)≐(N x)⟩) → (λ M) ≐(λ N) # csiv 3

    // example from the end of the section
    goal = λ[x]'0 @ ((λ[y] y) @ x) ≐λ[x] '0 @ x
```

---

[7] in file `https://gl.mathhub.info/MMT/urtheories/blob/devel/source/llfp.mmt`
[8] in file `https://github.com/UniFormal/MMT/blob/devel/src/mmt-lf/src/info/kwarc/mmt/lf/externals/ExternalCheck.scala`

```
// The following check succeeds without ever calling the ValRule because the unlock is under a lock.
check: goal
  = csiv [x] L (Val x term) ⟨
     eq_app (refl '0) U⟨betav ([y] y) x⟩
  ⟩
// The following example does not guard the unlock but still succeeds because 'n is indeed a value.
check2 = [n] eq_app (refl '0) U⟨betav ([y] y) 'n⟩
// The following negative example fails because x is any term and thus not necessarily a value.
fail = [x] eq_app (refl '0) U⟨betav ([y] y) x⟩
```

Listing 6: A Rule Inferring the Type of a Lock Term

```
object InferUnlock extends InferenceRule(Unlock.path) {
        def apply(solver: Solver)(tm: Term)(context: Context) : Option[Term] = {
                ... // side condition is p(t,a)
                solver.rules.getByHead(classOf[ExternalConditionRule], p).headOption match {
                        case Some(rule) =>
                          if (! rule(solver)(context, t, a))
                                return None // side condition failed
                        case None =>
                          solver.error("no rule for condition " + p + " found")
                }
                ...
        }

object ValRule extends ExternalConditionRule(CallByValue.Val) {
  def apply(solver: Solver)(context: Context, t: Term, a: Term) : Boolean = {
    t match {
      case LF.Apply(CallByValue.free, _) => true
      case LF.Apply(CallByValue.lam, _) => true
      case _ => solver.error(solver.presentObj(t) + " must be lambda−abstraction or variable")
    }
  }
}
```

During this case study, we found a few possible simplifications to the LLF$_{\mathscr{P}}$ calculus — minor tweaks that make the language and implementation easier but do not appear to change its essence. That shows rapid prototyping at its best: with MMT's parser, type reconstruction, and IDE working out of the box, we could tweak the LLF$_{\mathscr{P}}$ grammar and update the rules and the λ_v example with just a few minutes of turnaround time. That was critical to conduct the experimentation needed to find these simplifications.

6   Linear Logic

The previous case studies focused on systems relatively near to LF, for which MMT was originally designed. The next case study takes us out of this comfort zone by considering a substructural logic, which tends to be incompatible with LF.

> Linear logic is tricky to represent in standard (i.e., not substructural) frameworks. The resource semantics provides an appealing solution, but it is only elegant if a free monoid is available. In MMT, we can simply add one by writing appropriate equality rules.

Substructural frameworks omit one or several of the basic principles *exchange*, *weakening* or *contraction*. Since LF uses all of them, this makes elegant representations of substructural logics difficult.

The resource semantics of linear logic interprets formulas in a Kripke frame, modeling the linear contexts as worlds. Consequently, this requires a free commutative monoid structure for the type of worlds. Our formalization follows Frank Pfenning's lecture notes.

Listing 7: Resource Semantics of Linear Logic

```
theory ResourceSemantics : LF =
        o : type
        ⊗: o → o → o # 1 ⊗2
        ⊸: o → o → o # 1 ⊸2 prec 80
        ... // other connectives omitted

        // worlds contain resources
        world : type
        // composition of worlds, corresponds to multiset union of resources
        union : world → world → world # 1 * 2 prec 100
        // empty world with no resources
        empty : world # ε
        // truth at a world
        At : o → world → type # 1 @ 2 prec 50
        // validity is truth in the empty world
        valid : o → type = [A] A @ ε

        ... // axioms for commutative monoid omitted

        // proofs of the linear sequent A1, ..., An |− A are represented as terms of type
        // (A1 ⊗... ⊗An) ⊸A @ ε, where @ is the binary holds−at relation between
        // propositions and worlds
        ⊗_R : {A,B,a,b} A @ a → B @ b → A ⊗B @ a∗b
        ⊗_L : {A,B,C,u,v} A ⊗B @ u → ({a}{b} A @ a → B @ b → C @ a∗b∗v) → C @ u∗v
        # ⊗_L 6 7
        ⊸_R : {A,B,w} ({a} A @ a → B @ a∗w) → A ⊸B @ w
        ... // other rules omitted
```

We give a fragment of our formalization in Listing 7[9]. It uses plain LF as the meta-theory and declares the linear propositions, the free commutative monoid of worlds, the holds-at relation between propositions and worlds, and the proof rules.

This formalization has the problem that proofs in linear logic now require equational reasoning about the worlds. The resulting proofs become inelegant and cumbersome. Therefore, we add two rules[10] that automate some of the equational reasoning. This is undecidable. But in linear logic proofs, the worlds are usually formed from bound variables only so that we can find reasonably good heuristics: (i) The rule NormalizeWorlds for $v \leadsto w$ at type `world` removes brackets (associativity) and units (neutrality) and normalizes the order of bound variables (commutativity). (ii) The rule EquateWorlds for judgment $v \equiv w$: `world` normalizes both v and w and then tries to equate them. Here we make use of the freeness property: in the free monoid, if v, v', w are bound variables, $v * w \equiv v' * w$: `world` is equivalent to $v \equiv v'$: `world`. This is important to make type reconstruction recurse into equality checks that eventually solve meta-variables of type `world`. This is necessary to make the world parameters of the proof rules implicit arguments, which in turn is necessary to write proofs in practice. Both rules are straightforward to implement and take just a few dozen lines of Scala code.

If we add these rules, we can write linear logic proofs elegantly. An example is given in Listing 8.

---

[9]see file https://gl.mathhub.info/MMT/examples/blob/devel/source/logic/linear.mmt
[10]in   file   https://gl.mathhub.info/MMT/examples/blob/devel/scala/info/kwarc/mmt/examples/
ResourceSemantics.scala

Listing 8: A Proof for Associativity of ⊗

```
// reconstruction must solve abc * X = abc and ab * Y * X = ab * c with X = epsilon, Y = c
⊗_assoc1 : {A,B,C} valid (A⊗B)⊗C ⊸A⊗(B⊗C)
    = [A,B,C] ⊸_R [abc,h] ⊗_L h
        [ab,c,pq,r] ⊗_L pq
            ([a,b,p,q] ⊗_R p (⊗_R q r))
```

Prototypes of other substructural logics can be obtained accordingly, e.g., by using idempotence to model the contraction rule.

7   Homotopy Type Theory

The previous case studies considered relatively simple formal systems that use very few language features and thus require only a few rules. Therefore, the next case study considers a more complex system combining several typing features.

> Homotopy Type Theory (HoTT,[23]) is a foundation of mathematics based on a dependently typed Martin-Löf type theory and the univalence axiom. We implement it in MMT using ≈ 45 rules.

HoTT is a relatively young formal system that has received significant attention and multiple variants are actively investigated. That makes it a prime candidate for rapid prototyping.

HoTT uses Π-types, Σ-types, coproduct types, finite types, W-types (which represent inductive types), a countable cumulative hierarchy of type universes, and intentionally proof-relevant equality types. We formalize each feature as a theory in the same way in which we formalized Π-types in LF and eventually obtain our HoTT prototype as the union of all theories.

We only present our definition of W-types here because it is the most unusual from the MMT perspective. The type $W_{x:A} B(x)$ represents the inductive type whose constructors are the terms $x : A$ whose arity is represented by $B(x)$. Typically, types A and $B(x)$ are finite. For example, the natural numbers correspond to $W_{x:\text{Bool}} x \text{ match}\{\text{true} \Rightarrow \varnothing, \text{false} \Rightarrow \text{Unit}\}$.

The elimination rule, for example, is: For $C' := C\left[c/_{\sup_c \left\{ x \Longrightarrow_{W_{x:A} B} g(x) \right\}} \right]$:

$$\frac{\Gamma \vdash w \Rightarrow \underset{x:A}{W} B \quad \Gamma, c : \underset{x:A}{W} B \vdash C \Rightarrow U \quad \Gamma, c : A, g : B[x/_c] \to \underset{x:A}{W} B, h : \prod_{y:B[x/_c]} C\left[c/_{g(y)} \right] \vdash e \Leftarrow C'}{\Gamma \vdash \text{rec}(w)\{(c,g,h) \Longrightarrow_C e\} \Rightarrow C[c/_w]}$$

W-types are a relatively simple representation of induction in a type-theoretical setting and are well-understood from an implementation point of view. Indeed, implementing the Scala rules is tedious but straightforward. But from a user perspective, even formalizing seemingly simple inductive types like natural numbers or lists using W-types is inconvenient and unintuitive. For example, Listing 9[11] shows an implementation of natural numbers and addition.

Listing 9: Natural Numbers as W-types

```
theory Wtest :?LFW =
    beta : ENUM 2 → type = [x] x match y. ∅|UNIT to type
    ℕ: type = W x:ENUM 2 . (beta x)

    zeroN : ℕ= sup (CASE 0) , x ⟹ (x ∅f ℕ) to ℕ
    S : ℕ→ ℕ= [x : ℕ] sup (CASE 1) , y ⟹ x to ℕ
```

---

[11]in file https://gl.mathhub.info/MMT/LFX/blob/devel/source/test.mmt

```
plusN : {n:ℕ,m:ℕ} ℕ= [n][m] (rec m,c,g,h ⟹ (c match (ENUM 2), x. n | S (h x) to ℕ) to ℕ)
    # 1 + 2 prec 5
```

In order to provide more convenient syntax, we write a plugin that extends the MMT language. This uses a powerful extension mechanism of MMT— called *structural features* — that we omitted in Figure 1 for simplicity. It is described in [13]. Our structural features are intended to enable the definitions shown in Listing 10[12], which is immediately recognizable as an inductive type and a recursive function.

Listing 10: Natural Numbers using Structural Features

```
theory FeatureTest : LFX/WTypes?Inductive =
        induct Nats () =
                Nat : type # ℕ
                Zero : ℕ# O
                Succ : ℕ→ ℕ# S 1
        def addition (n : ℕ) =
                add : ℕ→ ℕ# 1 + 2
                Zero = n
                Succ = [m] S (add m)
```

Structural features can be seen as an elegant logic-independent version of Isabelle's definitional packages. Syntactically, they allow using arbitrary nested theories as declarations. Semantically, these nested theories are elaborated into a list of declarations generated from its contents. Each such nested theory is introduced by a special keyword that indicates which structural feature provides the elaboration function.

For W-types, we provide two structural features for the keywords `induct` and `def`. Both are conservative in the sense that the elaboration contains only *defined* constants. The former generates a constant with a W-type as definiens and analogously defined constants for all constructors of the type. The latter generates an inductively defined function on a W-type using the cases given in the body. For example, Listing 11[13] shows the relevant part of code that generates the W-type.

Listing 11: Code that elaborates the Body of a Theory into a W-type

```
implicit val tpname : (GlobalName,Term) = (tpc.path,if (params.isEmpty)
    OMS(parent.path.module ? tpc.name) else
    ApplySpine(OMS(parent.path.module ? tpc.name),params.variables.map(_.toTerm):_*))
val cases = consts.tail.zipWithIndex.map(p => makeCase(p._1,p._2))

val tpA = Coprod(cases.map(_.getcptype):_*)
val (xn,_) = Context.pickFresh(dd.getInnerContext,LocalName("x"))
val (yn,_) = Context.pickFresh(dd.getInnerContext,LocalName("y"))
val tpB = cmatch(OMV(xn),tpA,yn,cases.map(_.getArity),OMS(Typed.ktype))
val tp = if (params.isEmpty) Some(OMS(Typed.ktype)) else Some(Pi(params,OMS(Typed.ktype)))
val df = if (params.isEmpty) Some(WType(xn,tpA,tpB)) else Some(Lambda(params,WType(xn,tpA,tpB)))
Some(Constant(parent.toTerm,tpname._1.name,tpc.alias,tp,df,tpc.rl,tpc.notC))
```

Assuming we have collected all parts of HoTT in a theory `Types`, we can finish the HoTT prototype by stating the univalence axiom. This is shown in Listing 12[14]. Here `NAT` is the type of natural numbers, and $\mathscr{U}$ forms the universes.

---

[12]same file
[13]in   file   https://gl.mathhub.info/MMT/LFX/tree/devel/scala/info/kwarc/mmt/LFX/WTypes/Features.scala
[14]in file https://gl.mathhub.info/MMT/LFX/tree/devel/source/HOTT.mmt

Listing 12: A Theory for HoTT

```
theory HOTT : Types =

    identity_fun : {i: NAT, A : 𝒰 i} A → A = [i,A]([x : A] x) # id 2

    homotopy : {i : NAT}{A : 𝒰 i, B : A → 𝒰 i} ({x:A} B x) → ({x:A} B x) → 𝒰 i
             = [i][A, B][f][g] {x:A} (f x ≐g x) # 4 ∼5

    is_equiv : {i : NAT}{A:𝒰 i,B:𝒰 i} (A → B) → 𝒰 i
             = [i][A,B][f] (∑ g : B → A . ([x:B] f (g x)) ∼(id B)) ×
                          (∑ h : B → A . ([x:A] h (f x)) ∼(id A)) # isequiv 4

    equivalence : {i : NAT}𝒰 i → 𝒰 i → 𝒰 i
             = [i][A,B] ∑f : A → B . isequiv f # 2 ≃3

    univalence : {i : NAT}{A:𝒰 i,B:𝒰 i} (A ≃B) ≃(A ≐B)
```

Our definition of HoTT followed the original one from [23]. Since then, a flurry of activity has led to many different variants, in particular regarding the treatment of equality and the formation of higher inductive types. Our prototype provides an ideal starting point to experiment with these variants.

8   Conclusion and Future Work

Successes   We have evaluated the MMT framework from a rapid prototyping perspective by defining 5 formal systems in it. Our experiences highlight two essential strengths: MMT is very flexible thus allowing for elegant representation of diverse systems — critical because prototypes are often needed exactly for novel ideas to which existing framework are not applicable. And it makes the process of implementing and testing these representations fast and convenient — critical because it enables a rapid feedback cycle between theory and practice.

These findings are supported by other case studies that we did not mention here. Major previously published case studies defined record types (including defined fields and the reflection of theories into records) [15] and the module system of PVS [14]. Unpublished results include LISP-style quasi-quotation, sequence arguments, and intersection types, more general inductive types (Colin Rothgang), and the diagram presentation language of [3] (Yasmine Sharoda).

We are also working on several exports of proof assistant libraries relative to representations of Isabelle/Pure (with Makarius Wenzel) and Coq (with Claudio Sacerdoti Coen) in MMT. In fact, such exports were part of the original motivation of MMT: Representations of proof assistant libraries in standardized frameworks had previously suffered because existing frameworks were too inflexible to represent the respective logics in ways that naturally match the implementations in the proof assistants.

To further evaluate the approach, increasingly challenging benchmarks should be attempted. We expect this will be successful for, e.g., Coq-style type classes and unification hints, Idris-style meta-programming, Beluga-style reasoning about contexts, or Abella-style two-level logic and ∇ operator.

Limitations   There are also challenges that, while possible, are currently somewhat inconvenient. While we are confident that these do not constitute limitations of the overall approach, they show limitations in the current system. Most importantly, MMT needs support for proof search in order to prototype formal systems with undecidable typing relations, where type reconstruction must synthesize a proof term. Examples are predicate subtypes and quotient types, undefinedness, and Mizar-style soft type sy-

stems. While proof search is already possible in MMT, it currently suffers from the lack of an efficient prover.

We prioritized writing declarative rules in logical frameworks in order to minimize the amount of user-written Scala code that has to be trusted. In particular, we used a plugin to operationalize declarative specifications by generating rewrite rules in Section 4. But we could still do better by dynamically generating more complex rules from appropriately tagged declarations. For example, we could generate typing rules to handle unification hints or soft typing. It is unclear if this is already possible or requires adaptations of the MMT kernel.

Finally, MMT has prioritized making the *prototyping* fast rather than the *prototype*: it focuses on making prototypes usable (flexible notations, error messages, IDE, etc.) and just fast enough for interactive case studies. While MMT rules are fast because they are written in a mainstream programming language, MMT loses speed by explicitly dispatching to these rules (rather than using the dispatch mechanism of the underlying programming language). Other systems like ELPI [6] or Dedukti [2] have instead optimized for the speed of the prototype. While the speed of the prototypes was not a bottleneck in our case studies, it could become problematic if MMT-based prototypes are evaluated on large inputs, e.g., for interactive theorem proving (as in ELPI) or batch proof checking (as in Dedukti). It will be interesting to investigate if and how the advantages of these systems can be combined, e.g., if MMT can delegate to ELPI for proof search.

References

[1] 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.

[2] 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.

[3] J. Carette and R. O'Connor. Theory Presentation Combinators. In J. Jeuring, J. Campbell, J. Carette, G. Dos Reis, P. Sojka, M. Wenzel, and V. Sorge, editors, *Intelligent Computer Mathematics*, volume 7362, pages 202–215. Springer, 2012.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] A. Felty and A. Momigliano. Hybrid - a definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning*, 48(1):43–105, 2012.

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

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

[11] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[12] 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.

[13] M. Iancu. *Towards Flexiformal Mathematics*. PhD thesis, Jacobs University Bremen, 2017.

[14] 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.

[15] Dennis Müller, Florian Rabe, and Michael Kohlhase. Theories as types. Springer Verlag, 2018. URL: http://kwarc.info/kohlhase/papers/ijcar18-records.pdf.

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

[17] 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.

[18] 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.

[19] 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.

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

[21] F. Rabe. A Modular Type Reconstruction Algorithm. *ACM Transactions on Computational Logic*, 2018. to appear; see https://kwarc.info/people/frabe/Research/rabe_recon_17.pdf.

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

[23] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[24] 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.