Generic Literals

Florian Rabe

Jacobs University, Bremen, Germany

Abstract. MMT is a formal framework that combines the flexibility of knowledge representation languages like OPENMATH with the formal rigor of logical frameworks like LF. It systematically abstracts from theoretical and practical aspects of individual formal languages and tries to develop as many solutions as possible generically.
In this work, we allow MMT theories to declare user-defined literals, which makes literals as user-extensible as operators, axioms, and notations. This is particularly important for framework languages, which must be able to represent any choice of literals. Theoretically, our literals are introduced by importing a model that defines the denotations of some types and function symbols. Practically, MMT is coupled with a programming language, in which these models are defined.
Our results are implemented in the MMT system. In particular, literals and computation on them are integrated with the parser and type checker.

1 Introduction and Related Work

Even though literals (e.g., for booleans, integers, or strings) are a common feature of formal systems, there appears to be no general definition of what they are. Most languages simply use a fixed set of primitive types with built-in literals, which appear explicitly in the grammar, the semantics, and the implementation. To our knowledge, there is no system that is systematically parametric in the choice of literals – while users can declare new constants, functions, axioms, and notations, etc., the set of literals is usually fixed.

A fixed set of literals is often a reasonable choice. But it has some weird effects. For example, OPENMATH [BCC+04] is meant to subsume the languages of all other formal systems. But it fixes a set of literals and thus cannot represent any system that uses different literals. Another example, the proof assistant HOL Light [Har96] relegates as much as possible to its library, including, e.g., the definition of the type of natural numbers. But support for natural number literals must be built into the base system, anticipating the library's type definition.

The same observations apply to interpreted constants, which usually go along with literals. For example, languages with natural number literals "0", "1", ... : `nat` usually also provide built-in constants such as $+ : \texttt{nat} \to \texttt{nat} \to \texttt{nat}$. When applied to literals, their values can be directly computed, e.g., "1" + "1" $\rightsquigarrow$ "2". (To clarify the presentation, we will enclose all literals in quotes throughout this paper.) In higher-order languages, we can think of these as literals of function type, but the general case requires a distinction between literals and interpreted functions.

State of the Art We speak of **primitive literals** if a language fixes sets L_A for types A and adds the rule schema

$$\frac{}{\vdash \text{"}l\text{"} : A} \text{ for } l \in L_A$$

Note that L_A may be infinite, e.g., we could put $L_{\texttt{nat}} = 0, 1, \ldots$. Sets of (usually finitely many) interpreted constants are supplied accordingly.

This is typical for programming languages and computer algebra systems. Often the types A include booleans, bounded integers, floating point numbers, characters, and strings. Computer algebra systems prefer unlimited precision integers and rational numbers. Mathematica [Wol12] offers many primitive literals, e.g., for images, which users input as files.

Logics, on the other hand, use primitive literals very sparingly: firstly, because the interplay between reasoning and computation (when interpreted functions are applied to non-literals) is very difficult; secondly, because unverified computation endangers logical consistency. Martin-Löf type theory [ML74] uses a primitive type for natural numbers. Nuprl [CAB+86] uses a primitive type of integers. Recently, the TPTP family of interchange logics added support for primitive literals for integer, rational, and real numbers [SSCB12].

We speak of **literals as constants** if every literal is declared as a constant. This makes literals extensible but is only feasible for very small sets of literals. The most common example are the boolean constants `true` and `false`.

We speak of **enumeration literals** if languages allow defining new finite types by listing fresh identifiers for their elements. This is supported by C or any language with inductive data types. It is less impractical than literals as constants but still impractical in general.

We speak of **pseudo-literals** (no negative judgment intended) if literals appear in the concrete syntax but are converted into other representations in the abstract syntax. Then interpreted constants are not needed at all because they can be defined on the abstract representations. However, they are still not extensible in practice because users are usually not able to modify the parser and printer that perform the conversion.

Examples of pseudo-literals are strings in C (converted to character arrays) or XML literals in Scala (converted to objects). They are also used in various proof assistants. HOL Light [Har96] represents natural numbers as (essentially) lists of bits, and the parser and printer add pseudo-literals to the concrete syntax. Somewhat similarly, Mizar [TB85] uses 16-bit integer literals as natural numbers.

Literals present a particular **challenge for knowledge representation languages** like OpenMath because there is no good canonical choice which literals to support. For example, OpenMath [BCC+04] somewhat arbitrarily uses primitive literals for unbounded decimal integers (OMI), decimal or hexadecimal IEEE floating point numbers (OMF), strings (OMSTR), and bytearrays (OMB), and literals as constants for booleans (CD logic1). It lacks, e.g., literals for bounded or arbitrary-base integers as well as characters (not to mention more exotic types like URIs, dates, or colors).

They are also a **challenge for logical frameworks** like LF [HHP93], which represent other languages as theories. Because these other languages may use different literals, a logical framework should not fix any literals but allow theories to declare flexibly which literals are used. The Twelf system [PS99] supports constraint domains for this purpose, which subsume primitive literals. However, only developers but not users can add new constraint domains.

Contribution We give a **general definition** of literal and interpreted constants. We base our definition on the MMT language [RK13], which already gives extremely general definitions of theory, declaration, and expression. Our MMT literals are **extensible** and **context-specific**: MMT theories can declare new primitive literals and new interpreted constants, and every theory sees only the ones it declares (or imports). We implement our literals as a part of the MMT system [Rab13].

We hold that this is the **only feasible design for generic languages** such as OPENMATH-style representation languages or logical frameworks.

Our key idea is to extend the MMT syntax with a single constructor for literals: "l" where l is an arbitrary element of some semantic domain. Clearly, in a declarative formal language, we cannot easily use a set theoretical domain. Therefore, in practice, we use the Scala programming language, which underlies the MMT system, as the semantic domain.

Building on the correspondence between the MMT module system and Scala inheritance [KMR13], we let MMT **theories and Scala classes import each other**. Then a typical use of our system proceeds in three steps:

1. We axiomatize an MMT theory T, e.g., with a type `nat` and a constant `plus`.
2. T behaves like an abstract Scala class. We implement it in a concrete Scala class C, e.g., using the positive integers for `nat` and addition for `plus`.
3. C behaves like an MMT theory that includes T and adds a definiens to every constant in T. The respective definiens is the literal representing the Scala value assigned by C. MMT theories including C may use the literals and the interpretations of the constants into scope. For example, that would make natural number literals available, and add addition to the MMT simplifier.

Our primary motivation is to use our extensible literals sparingly: The choice of literals will usually be made at the beginning of a development, and most users will never add new literals.

However, if used more aggressively, our approach also offers a framework in which we can **combine deduction** (which guarantees correctness) **and computation** (which is much more efficient) and investigate their interplay. Our new MMT theories can mediate between the two paradigms by combining logical theories T and computational theories C. Even without further insights into how to verify the correctness of computations, this can help make them more trustworthy: Just putting the two side by side in a common formal framework is a step up compared to some current systems.

Therefore, we do not simply design our system as a foreign function interface to MMT (even though it can be seen as such). Instead, we first develop the

semantics of combined theories for arbitrary models C, and then specialize to the case where C is an effectively given model, i.e., a set of computations.

Overview We recap the existing MMT language in Sect. 2. Then we extend it with a general notion of literals in Sect. 3 and specialize to Scala-based computational literals in Sect. 4. We present the practical aspects of our implementation in Sect. 5. In Sect. 6, we discuss further related work and conclude.

2 Preliminaries: The MMT Language

The MMT language and module system were originally introduced in [RK13]. The recap here follows the formulation of [Rab14] and focuses on a very small, non-modular fragment.

Grammar The grammar for basic MMT theories is given in Fig. 1. A **theory** Σ is a list of **constant** declarations. These are of the form $c[:\ A][=\ t][\#N]$ where c is an identifier, A is its **type**, t its **definiens**, and N its notation, all of which are optional. A **context** Γ is very similar to a theory and declares typed variables.

Theory	Σ	::=	$(c[:\ t][=\ t][\#N])^*$
Context	Γ	::=	$(x : t)^*$
Terms	t	::=	$c \mid x \mid c(\Gamma; t^*)$
Notation	N	::=	omitted

Fig. 1. MMT Grammar

Constant declarations subsume the most common declarations of formal systems including built-in symbols, universes, function/predicate/type symbols, and (by using propositions as types) axioms, theorems, and inference rules.

Type and definiens are terms, and **terms** are formed from constants c, variables x, and complex terms $c(\Gamma; t_1, \ldots, t_n)$. Complex terms bind the variables of Γ in the arguments t_i. This includes OPENMATH-style binding and application: $\Gamma = \cdot$ yields application (OMA) of c to the t_i, and $\Gamma \neq \cdot$ and $n = 1$ yields binding (OMBIND) with binder c. Like OPENMATH objects, MMT terms subsume most expressions of formal systems such as terms, types, formulas, and proofs.

Like OPENMATH, MMT's syntax is **generic** in the sense that there are no predefined constants in the grammar. Therefore, to represent any language L in MMT, we first have to declare a special theory (which we usually also call L for convenience) with one untyped constant for each built-in symbol of L. Then we define L-**theories** as the MMT theories of the form L, Σ such that all Σ-constants have a type.

We usually think of L as a logical framework that is used to define other languages as L-theories. As running examples, we will use the dependently typed λ-calculus LF [HHP93] as a logical framework and the natural numbers as an LF-theory:

Example 1 (An MMT Theory for LF). LF is the theory shown on the left of Fig. 2. For example, the abstract syntax for a λ-abstraction is `lambda`$(x : A; t)$ where $x : A$ the single bound variable, and t the single argument. The notation declares the concrete syntax to be $[x : A]t$. Similarly, in an application

$\texttt{apply}(\cdot; f, t)$, no variables are bound, and f and t are the arguments. The concrete syntax is $f\ t$.

The theory `Nat` on the right of Fig. 2 extends LF with some example declarations for the natural numbers. We can also represent axioms in the usual LF-style, but do not do that here because we do not need them later on.

theory `LF`		theory `Nat`	
`type`		`prop`	$: \texttt{type}$
`Pi`	$\#\ \{\, x_1 : t_1 \,\}\, t_2$	`nat`	$: \texttt{type}$
`lambda`	$\#\ [\, x_1 : t_1 \,]\, t_2$	`succ`	$: \texttt{nat} \to \texttt{nat}$
`apply`	$\#\ t_1\ \ t_2$	`plus`	$: \texttt{nat} \to \texttt{nat} \to \texttt{nat} \quad \#\ t_1 + t_2$
`arrow`	$\#\ t_1 \to t_2$	`equal`	$: \texttt{nat} \to \texttt{nat} \to \texttt{prop} \ \#\ t_1 \doteq t_2$

Fig. 2. LF in Mmt (left) and Natural Numbers in LF (right)

Inference System The main **judgments** of Mmt are given in Fig. 3. There is no need to introduce the type system in detail here, and we refer to [Rab14] instead. For our purposes, it is sufficient to know that the type system is generic as well: Mmt itself only provides the structural rules for declarations, congruence, α-equality, and constants.

$\Gamma \vdash_\Sigma \_ : A$	A may be a type
$\Gamma \vdash_\Sigma t : A$	t has type A
$\Gamma \vdash_\Sigma t \equiv t'$	t and t' are equal

Fig. 3. Judgments of Mmt

All other rules are provided separately when representing a logical framework:

Example 2 (The Logical Framework LF). For LF, we add the usual rules for λ-abstraction including

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

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

where $B[t]$ denotes substitution. We do not need any rules for the LF-theory `Nat` because it inherits all rules from LF.

Once a logical framework L is represented in this way, we never have to add rules again – we represent further languages declaratively as L-theories.

Finally, consider an L-theory $\Sigma = \ldots, c_i : A_i[= t_i], \ldots$ and let $\Sigma^i = \ldots, c_{i-1} : A_{i-1}[= t_{i-1}]$. Σ is **valid** if $\vdash_{L,\Sigma^i} \_ : A_i$ [and $\vdash_{L,\Sigma^i} t_i : A_i$] for all i.

Implementation The Mmt system [Rab13] implements the above concepts. This includes generic implementations of parsing (according to the notations) and type reconstruction (according to the supplied rules). It adds a module system for theories, of which we will make modest use in some examples.

3 Literals as Semantic Values

We first introduce some general semantic notions for MMT before we define literals. The main intuition is that models are collections of values and functions, which we want to reflect into the syntax.

3.1 Models of MMT Theories

A **semantic domain** is a triple $(U, \hat{:}, \hat{=})$ where U is any collection of objects, and $\hat{:}$ and $\hat{=}$ are binary relations on U such that $\hat{=}$ is an equivalence and congruent with respect to $\hat{:}$.

Example 3 (Set Theory). To obtain a semantic domain for set theory, we put U to be the collection of all classes. Then $\hat{:}$ is the $\in$ relation, and $\hat{=}$ is extensional equality of classes.

Let us now fix a logical framework L represented in MMT and a semantic domain $(U, \hat{:}, \hat{=})$. Given an L-theory Σ, a Σ-**model** M is a function that maps every Σ-constant c to an element c^M of U. **Assignments** α interpret the free variables of a context accordingly.

Example 4 (Standard Natural Numbers). Given the LF-theory for the natural numbers from Ex. 1, we define the standard model `StdNat` in set theory in the obvious way: $\texttt{nat}^{\texttt{StdNat}} = \mathbb{N}$, $\texttt{prop}^{\texttt{StdNat}} = \{0, 1\}$, and $\texttt{succ}^{\texttt{StdNat}}$, $\texttt{plus}^{\texttt{StdNat}}$, $\texttt{equal}^{\texttt{StdNat}}$ are defined as usual.

A **semantics** for an MMT theory L consists of a semantic domain and a set of interpretation rules. The interpretation rules must extend every Σ-model M to a (usually partial) interpretation function $[\![-]\!]^M$ that maps closed Σ-terms to elements of U.

We do not spell out the general shape of these interpretation rules, and instead only give a concrete example for the standard set theoretical semantics of LF. In fact, if we use LF as a logical framework in which other languages are specified, we only need to define the semantics of LF anyway:

Example 5 (Set Theoretical Semantics of LF). We use the domain from Ex. 3. Given a model M of an LF-theory Σ and an assignment α for the context Γ, we use the following interpretation rules to define $[\![t]\!]^M_\alpha \in U$ for terms t in context Γ:

$$[\![type]\!]^M_\alpha = \mathcal{SET} \qquad [\![A \to B]\!]^M_\alpha = ([\![B]\!]^M_\alpha)^{[\![A]\!]^M_\alpha}$$

$$[\![[x : A]t]\!]^M_\alpha = [\![A]\!]^M_\alpha \ni u \mapsto [\![t]\!]^M_{\alpha, x \mapsto u} \qquad [\![f\ t]\!]^M_\alpha = [\![f]\!]^M_\alpha([\![t]\!]^M_\alpha)$$

We omit the according rule for terms constructed using `Pi`.

Finally, a model M is **sound** if it preserves typing and equality, i.e., if

$$\text{if } \Gamma \vdash_\Sigma t : A \quad \text{ then } \quad [\![t]\!]^M_\alpha \mathrel{\hat{:}} [\![A]\!]^M_\alpha$$

$$\text{if } \Gamma \vdash_\Sigma t \equiv t' \qquad \text{then} \qquad [\![t]\!]^M_\alpha \mathrel{\hat{=}} [\![t']\!]^M_\alpha$$

In particular, soundness requires models to interpret Σ-constants $c : A[= t]$ as values $c^M \mathrel{\hat{:}} [\![A]\!]^M$ [such that $c^M \mathrel{\hat{=}} [\![t]\!]^M$] $(*)$. We call the semantics as a whole **sound** if the inverse holds, too, i.e., if every model satisfying $(*)$ is sound. This is the case for the semantics of Ex. 5.

3.2 Internalizing Models

We fix a logical framework L and a sound semantics for L that uses a semantic domain $(U, \hat{:}, \hat{=})$. We will now internalize the semantic domain, the semantics, and individual models into Mmt. The main idea is to treat every element of $l \in U$ as a literal that may occur as a term $"l"$. Of course, this may yield an uncountable set of terms, or even a proper class if we use the semantic domain from Ex. 3. However, this level of abstraction is well-suited to define the general concepts even if practical systems must be much more restricted.

To **internalize the semantic domain**, we extend the Mmt grammar from Fig. 1 with **one new production** $"l"$:

$$t ::= c \mid x \mid c(\Gamma; t^*) \qquad \mid "l" \text{ for } l \in U$$

Moreover, we add **two new inference rules** to Mmt:

$$\frac{}{\vdash_\Sigma "l" : "l'"} \text{ for } l \mathrel{\hat{:}} l' \qquad \frac{}{\vdash_\Sigma "l" \equiv "l'"} \text{ for } l \mathrel{\hat{=}} l' \qquad (*)$$

Because we extended the grammar, we also need to extend the semantics with a **new interpretation rule** – models interpret every literal as itself:

$$[\!["l"]\!]^M = l \qquad (**)$$

We call t a **value** if $\vdash_\Sigma t \equiv "l"$ for some l. Then we have the following inverse of $(**)$:

Theorem 1. *If t is a value and M is a sound Σ-model, then $\vdash_\Sigma "[\![t]\!]^M" \equiv t$.*

Proof. Assume $\vdash_\Sigma t \equiv "l"$. The key step is to use soundness, $(**)$, and $(*)$ to obtain $\vdash_\Sigma "l" \equiv "[\![t]\!]^M"$.

At this point, there are no typing rules that relate literals to non-literal terms. Thus, our literals are logically inconsequential. That changes with the next definition: We **internalize the semantics** by adding its interpretation rules as Mmt typing rules. Again, it is sufficient for our purposes to give only an example for LF:

Example 6. We use the following typing rules to internalize the interpretation rules from Ex. 5:

$$\frac{}{\vdash_\Sigma \texttt{type} \equiv "\mathcal{SET}"} \qquad \frac{\vdash_\Sigma a \equiv "A" \qquad \vdash_\Sigma b \equiv "B"}{\vdash_\Sigma a \to b \equiv "B^A"}$$

$$\frac{\vdash_\Sigma f \equiv "F" \qquad \vdash_\Sigma t \equiv "T"}{\vdash_\Sigma f\ t \equiv "F(T)"} \qquad \frac{\vdash_\Sigma a \equiv "A" \qquad \vdash_\Sigma t["u"] \equiv "T_u" \text{ for all } u \in A}{\vdash_\Sigma [x : A]t \equiv "A \ni u \mapsto T_u"}$$

```
include "StdNat"
c    : a
a    : type
vec  : nat → type
nil  : vec "0"
cons : {n : nat} a → vec n → vec (succ n)
concat : {m : nat}{n : nat}
         vec m → vec n → vec (m + n)
test0  : vec "2" = cons "1" c (cons "0" c nil)
test1  : vec "4" = concat "2" "2" test0 test0
```

Fig. 4. Vectors in LF with Natural Number Literals

Finally, we can **internalize models**. For an L-theory Σ and a Σ-model M, the MMT theory "M" contains for every Σ-constant $c : A[= t]$ the constant $c : A = "c^M"$. Intuitively, in "M", every constant is equal to its interpretation, and via the internalized semantics, so is every interpretable term. The following theorem shows that we can indeed treat models as special theories:

Theorem 2. *Consider a valid L-theory $\Sigma = c_1 : A_1, \ldots, c_n : A_n$. If all A_i are values, then a model M is sound iff "M" is a valid L-theory.*

Proof. Because we assume that Σ is valid, the validity of "M" is equivalent to $\vdash_{"M"} "c^M" : A$ for every Σ-constant $c : A$.

Assume M is sound. Then $c^M \hat{:} [\![A]\!]^M$ and thus $\vdash_{"M"} "c^M" : "[\![A]\!]^M"$. Then validity follows using Thm. 1.

Conversely, assume "M" is valid. We show that M is sound by induction on the declarations in Σ. For $c : A = "c^M"$ in "M", the induction hypothesis implies $[\!["c^M"]\!]^M \hat{:} [\![A]\!]^M$. Then $(**)$ yields $c^M \hat{:} [\![A]\!]^M$.

3.3 Literals through Internalized Models

We can now show how MMT theories can declare specific sets of literals. First we give an L-theory Σ, which declares all types for which we want to supply literals and all function symbols for which we want to fix an interpretation. Then we give a Σ-model M, which supplies the literals and interpretations. Finally any theory of the form "M", Θ includes these literals and interpretations. In particular, every theory may use different literals.

Example 7. We work with the internalized semantics of LF from Ex. 6. Let `Vec` be the LF-theory of Fig. 4. It uses the MMT module system to include the theory "`StdNat`" from Ex. 4 (and thus also includes `Nat` from Ex. 1). The left part introduces `vec` n as the type of vectors of length n over a fixed type a (with a dummy value $c : a$). The right part introduces an operation for concatenation (whose axioms we omit) and declares some example constants that use natural number literals.

Note that the example declarations are well-typed in LF only because the internalization of the semantics implies that $\vdash_{\texttt{Vec}} \texttt{vec}"2" \equiv \texttt{vec}(\texttt{succ}(\texttt{succ}("0")))$ and $\vdash_{\texttt{Vec}} \texttt{vec}"4" \equiv \texttt{vec}("2" + "2")$. Such an integration of interpreted constants into a dependent type theory is non-trivial even for a fixed set of literals.

It remains to define theories like "`StdNat`" in practice. Obviously, this is only possible if we can internalize the semantics in a computationally effective way. We look at that in the next section.

4 Literals as Computational Values

4.1 Models as Implementations

While set theory is interesting theoretically, in practice we need to use a programming language or computer algebra system to define models. To do that, we can reuse all concepts from Sect. 3 – we only have to use a different semantic domain. As a concrete example, we will use the simply-typed functional programming language Scala [OSV07], but we could use any other computational language accordingly:

Example 8 (Scala as a Semantic Domain). Scala permits toplevel declarations of new types (classes) and values (objects). Therefore, we assume a fixed set G. We further assume all classes are immutable (which is the only case we need anyway).

Then we obtain a semantic domain $(U, \hat{:}, \hat{=})$ as follows. U consists of the symbol `type`, all Scala types over G, and all typed Scala expressions over G. $\hat{:}$ relates all types to `type` and all expressions to their type. $\hat{=}$ relates `type` to itself, two types if they expand to the same normalized type, and two expressions if they evaluate to the same value.

Scala is not dependently typed. Therefore, we use a straightforward type erasure translation to interpret LF in Scala – it interprets LF functions as Scala functions but removes all arguments from dependent types:

Example 9 (Semantics of LF in Scala). Using the semantic domain from Ex. 8, we define the following interpretation rules:

$$[\![type]\!]^M = \texttt{type} \qquad [\![A \to B]\!]^M = [\![A]\!]^M \Rightarrow [\![B]\!]^M \qquad [\![\{x : A\}B]\!]^M = [\![A]\!]^M \Rightarrow [\![B]\!]^M$$

$$\text{for terms}: \quad [\![[x : A]t]\!]^M = (x : [\![A]\!]^M) \Rightarrow [\![t]\!]^{M, x \mapsto x} \qquad [\![f\ t]\!]^M = [\![f]\!]^M([\![t]\!]^M)$$

$$\text{for types}: \quad [\![[x : A]t]\!]^M = [\![t]\!]^M \qquad [\![f\ t]\!]^M = [\![f]\!]^M$$

where $\Rightarrow$ is Scala's syntax for both function types and λ-abstraction.

Of course, the type erasure translation loses some precision if an LF-theory makes use of dependent types. This is harmless, however, because most interpreted functions that we want to implement in practice are simply-typed, usually even first-order.

Giving models relative to this semantics means to implement MMT theories in Scala. Moreover, a model of a theory T has the same structure as a Scala object that implement an abstract class T. Therefore, we can directly use Scala syntax to write the models:

Example 10 (Integers in Scala). We give the abstract Scala class obtained by applying the interpretation rules from Ex. 9 to the theory `Nat` from Ex. 1, and a model of it:

```
abstract class Nat {
  type nat
  type prop
  def succ(x:nat): nat
  def plus(x:nat,y:nat): nat
  def equal(x:nat,y:nat): Boolean
}
object StdInt extends Nat {
  type nat = BigInt
  type prop = Boolean
  def succ(x:nat) = x+1
  def plus(x:nat,y:nat) = x+y
  def equal(x:nat,y:nat) = x == y
}
```

Here, we modeled the type `nat` as Scala's unbounded integers `BigInt`. We get back to that in Ex. 12.

Example 11 (OpenMath Literals). It is now straightforward to recover the 4 types of literals used by OpenMath as special cases. `StdInt` already implements OMI, along with some interpreted constants. Floating point numbers, strings, and byte arrays are equally simple.

4.2 Types as Partial Equivalence Relations

We can generalize the semantic domain from Ex. 8 substantially if we use partial equivalence relations (PERs) instead of types. A PER consists of a Scala type A and a symmetric and transitive binary relation r on A.

It is well-known that a PER r on A defines a quotient of a subtype of A. To see that, let A^s be the subtype of A containing all elements that are in relation to any other element. Then the restriction of r to A^s is reflexive and thus an equivalence. The postulated quotient arises as the quotient of A^s by r.

This results in a more expressive semantic domain in which we can take subtypes and quotients to build the exact type we need for our literals:

Example 12 (Scala PER Domain). We define PERs using the Scala class:

```
abstract class PER {
  type univ
  def valid(u: univ): Boolean
  def normal(u: univ): univ
}
```

`valid` defines the subtype, and $\texttt{normal}(x) = \texttt{normal}(y)$ defines the relation r.

Then we obtain a semantic domain $(U, \hat{:}, \hat{=})$ as follows. U contains `PER`, all expressions $p : \texttt{PER}$, and all pairs (p, u) for $p : \texttt{PER}$ and $p : i.\texttt{univ}$.

Then we put $p \mathbin{\hat{:}} \texttt{PER}$ if $p : \texttt{PER}$; and $p \mathbin{\hat{=}} p'$ if p and p' evaluate to the same value.

And we put $(p, u) \mathbin{\hat{:}} p$ if $p.\texttt{valid}(u)$; and $(p, u) \mathbin{\hat{=}} (p, v)$ if $p.\texttt{normal}(u) = p.\texttt{normal}(v)$.

We can now refine Ex. 10 by interpreting the type `nat` as a subtype of `BigInt`:

Example 13 (Natural and Rational Numbers). We use the semantic domain from Ex. 12 to define literals for natural numbers:

```
object StdNat extends PER {
  type univ = BigInt
  def valid(u: univ) = u >= 0
  def normal(u: univ) = u
}
```

Similarly, we can define rational numbers e/d using pairs (e, d) of integers:

```
object StdRat extends PER {
  type univ = (BigInt,BigInt)
  def valid(u: univ) = u._2 != 0
  def normal(u: univ) = {
    val (e,d) = u
    val g = (e gcd d) * d.signum
    return (e/g, d/g)
  }
}
```

Here validity ensures that the denominator is non-zero, and normalization cancels by the greatest common divider.

It is straightforward to adapt the semantics from Ex. 9 to this new semantic domain. The only subtlety are function types: Given two PERs p on A and q on B, it is easy to define the needed PER $p \Rightarrow q$ on $A \Rightarrow B$ in theory. However, validity and normalization in $p \Rightarrow q$ are not in general computable anymore, and the soundness of models that use functions on PERs is undecidable. This is acceptable because users anyway have to verify the correctness of their implementations manually.

5 Implementing Literals in the MMT System

5.1 Internalizing a Computational Semantics

Now we internalize the semantics of LF in Scala using the semantic domain of Ex. 12. Our reason for using Scala is that it underlies our implementation of

MMT. Therefore, we can extend it with Scala-based literals seamlessly. We make the following changes to the MMT implementation:

1) We add a feature to the MMT build tool: It exports all LF-theories T as abstract Scala classes T^*. Now the Scala-models of T are the Scala objects M that implement T^* (as in Ex. 10).

2) Conversely, we allow include declarations that include Scala objects M (as in Ex. 7). If an MMT theory includes "M", MMT locates the Scala object M and dynamically adds its definitions to the type system.

3) We implement the term constructor "l" for the special case where l is an element the Scala PER domain from Ex. 12. In particular, l can be of the form (p, u).

At this point, we do not add concrete syntax for constructing Scala expressions u to the MMT grammar (but see Sect. 5.2). Therefore, users can reference these new terms only indirectly: If c is a constant in T and we have included a model "M" of T, then we can use c to refer to the Scala value "c^M".

This restriction has two desirable effects: (i) Scala expressions can appear in MMT terms only if they have been explicitly imported. (ii) It remains transparent how a model M implements the theory T.

4) We internalize the Scala semantics by adding the following rules to the MMT type checker:

$$\frac{p = c^M \text{ for some "}M\text{" imported into } T \qquad p.\texttt{valid}(u) = \mathit{true}}{\vdash_T \text{"}(p,u)\text{"} : c}$$

$$\frac{p.\texttt{normal}(u) = p.\texttt{normal}(u')}{\vdash_T \text{"}(p,u)\text{"} \equiv \text{"}(p,u')\text{"}} \qquad \frac{l = c^M(l_1,\ldots,l_n) \qquad \text{"}M\text{" imported into } \Sigma}{\vdash_\Sigma c\text{"}l_1\text{"} \ \ldots\ \text{"}l_n\text{"} \equiv \text{"}l\text{"}}$$

This fully integrates literals and computation with MMT's type reconstruction. (If computations in M do not terminate, then type reconstruction times out.)

Notably, the models M can be written and included by users at run time just like normal MMT theories. In particular, the generated classes T^* hide details specific to the MMT code base, and users do not have to rebuild MMT after implementing M.

5.2 Lexing Rules

The changes of Sect. 5.1 only allow supplying interpreted function constants. They do not modify the parser, which is a major hurdle towards extensibility. However, MMT systematically uses rule-based lexing and parsing algorithms, whose rules are provided by the context. That makes it possible (and easy) to couple every type of literals with an appropriate lexing rule.

In our implementation, the Scala class `PER` actually has one additional field, which provides an optional lexing rule. Such a rule is a function that takes an input stream and returns either nothing or a literal that occurs at the beginning of the stream. We also provide some parametric lexing rules that can be instantiated to quickly create lexing rules for the most important cases. In particular, these include quoted and digit-based literals.

$$
\begin{array}{ll}
\texttt{include}\ \text{"}\texttt{StdNat}\text{"} & \\
c \quad : a & \texttt{concat} : \{m : \texttt{nat}\}\{n : \texttt{nat}\} \\
a \quad : \texttt{type} & \qquad \texttt{vec}\, m \to \texttt{vec}\, n \to \texttt{vec}\,(m+n) \\
\texttt{vec} \ : \texttt{nat} \to \texttt{type} & \mathit{test}0 \quad : \texttt{vec}\, 2 = \texttt{cons}\, c\,(\texttt{cons}\, c\, \texttt{nil}) \\
\texttt{nil} \ : \texttt{vec}\, 0 & \mathit{test}1 \quad : \texttt{vec}\, 4 = \texttt{concat}\, \mathit{test}0\, \mathit{test}0 \\
\texttt{cons} : \{n : \texttt{nat}\}\, a \to \texttt{vec}\, n \to \texttt{vec}\,(\texttt{succ}\, n) &
\end{array}
$$

Fig. 5. Vectors in LF using Scala-based Natural Number Literals

Example 14. Fig. 5 shows a variant of Ex. 7, which used set theory to define the natural number literals. Now we use the model from Ex. 13 extended with an appropriate lexing rule for digit-based literals.

Fig. 5 shows the concrete syntax that can be processed by Mmt. In particular the interpreted functions of "`StdNat`" are integrated with the dependent type system. Note that we can also omit some of the arguments to `cons` and `concat` because they can be inferred by Mmt.

5.3 Inversion Rules

A central difficulty of combining deduction and computation lies in terms that use both the usual function symbols and variables on the one hand and interpreted function symbols and literals on the other hand.

Example 15 (Unification). We extend Ex. 14 with the following declarations for the head of a non-empty vector:

$$\texttt{head} : \{n : \texttt{nat}\}\texttt{vec}\,(\texttt{succ}\, n) \to a \qquad \mathit{test}2 : a = \mathit{head}\, \mathit{test}0$$

Type-checking the declaration $\mathit{test}2$ leads to the unification problem $\texttt{vec}\,(\texttt{succ}\, X) = \texttt{vec}\, 2$ and thus $\texttt{succ}\, X = 2$, where X is a meta-variable representing the omitted first argument of `head`.

Without further help, the type checker is unable to solve this problem. In fact, because Mmt knows nothing about how $\texttt{succ}^{\texttt{StdNat}}$ is implemented, it cannot even tell if the problem is solvable at all.

This is a general problem, for which we do not claim a complete solution. It is also related to the more general unification problems addressed by canonical structures in Coq and unification hints in Matita [ARCT09].

However, the rule-based and highly extensible type reconstruction algorithm of Mmt provides a good setting for investigating possible solutions. As a first step, we allow Scala models to couple an interpretation c^M with a (possibly partial) implementation of the inverse function $c^{M^{-1}}$. If provided, Mmt adds the following rule to the type checker:

$$\frac{(l_1, \ldots, l_n) = c^{M^{-1}}(l) \qquad \vdash_\Sigma t_i \equiv \text{"}l_i\text{"}}{\vdash_\Sigma c\, t_1 \ \ldots \ t_n \equiv \text{"}l\text{"}}$$

This is already sufficient to solve many special cases in practice. For example, if we extend Ex. 15 with an implementation of $\texttt{succ}^{\texttt{StdNat}^{-1}}$ as the predecessor function, we can solve the above meta-variable as $X = \texttt{succ}^{\texttt{StdNat}^{-1}}(2) = 1$.

6 Conclusion and Further Related Work

Based on MMT, we have developed the syntax, semantics, and implementation of a formal language for mathematical content that offers extensible literals. Thus, no literals have to be built-in, and individual languages defined in MMT can fine-tune the set of literals freely. Our implementation uses partial equivalence relations on Scala types, which is expressive enough to cover all types of literals we are aware of.

Moreover, users can add interpreted functions that provide computation on literals. This computation is integrated into the equational theory of the MMT type system, including the use of computation in dependent types. Literals, interpreted functions, and the associated lexing and computation rules are subject to the MMT module system in the same way as constants, axioms, and notations. In particular, each MMT theory sees only the literals it declares or imports.

The applications of our work may go beyond our present focus on literals. Our design is a candidate for combining logical reasoning with efficient computation in the style of computer algebra systems. This is particularly interesting if we can generalize the concepts to allow structured literals (literals that may contain other terms as subterms). This would allow supplying literals for complex types such as polynomials over an arbitrary ring. Structured literals for functions would correspond to normalization by evaluation [BS91].

Related Work The theoretical aspect of our work shares a basic idea with biform theories [FvM03]. In a theory $"M", \Theta$, we can think of Θ as the axiomatic/intensional and of $"M"$ as the algorithmic/extensional form of a theory. This corresponds to the distinction made in biform theories.

In the context of rewrite systems, a similar idea was realized in [KN13]. There, sorted first-order rewrite theories consist of an interpreted and a free part, and computation on the interpreted part is relegated to an arbitrary model.

Our literals are also intriguingly similar to quotation in the sense of [Far13]. If we use the MMT language itself as the semantic domain, we obtain literals $"t"$ for terms t, which can be seen as quoted terms. Structured literals would correspond to quasi-quotation.

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.

BCC+04. S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See http://www.openmath.org/standard/om20.

BS91. U. Berger and H. Schwichtenberg. An Inverse of the Evaluation Functional for Typed λ-Calculus. In G. Kahn, editor, *Logic in Computer Science*, pages 203–211. IEEE Computer Society Press, 1991.

CAB+86. R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.

Far13. W. Farmer. The Formalization of Syntax-Based Mathematical Algorithms Using Quotation and Evaluation. In J. Carette, D. Aspinall, C. Lange, P. Sojka, and W. Windsteiger, editors, *Intelligent Computer Mathematics*, pages 35–50. Springer, 2013.

FvM03. W. Farmer and M. von Mohrenschildt. An Overview of a Formal Framework for Managing Mathematics. *Annals of Mathematics and Artificial Intelligence*, 38(1–3):165–191, 2003.

Har96. J. Harrison. HOL Light: A Tutorial Introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 265–269. Springer, 1996.

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.

KMR13. M. Kohlhase, F. Mance, and F. Rabe. A Universal Machine for Biform Theory Graphs. In J. Carette, D. Aspinall, C. Lange, P. Sojka, and W. Windsteiger, editors, *Intelligent Computer Mathematics*, pages 82–97. Springer, 2013.

KN13. C. Kop and N. Nishida. Term Rewriting with Logical Constraints. In P. Fontaine, C. Ringeissen, and R. Schmidt, editors, *Frontiers of Combining Systems*, pages 343–358. Springer, 2013.

ML74. P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the '73 Logic Colloquium*, pages 73–118. North-Holland, 1974.

OSV07. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. artima, 2007.

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. How to Identify, Translate, and Combine Logics? *Journal of Logic and Computation*, 2014. doi:10.1093/logcom/exu079.

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

SSCB12. G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. The TPTP Typed First-Order Form with Arithmetic. In N. Bjørner and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence*, pages 406–419. Springer, 2012.

TB85. A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28. Morgan Kaufmann, 1985.

Wol12. Wolfram. Mathematica, 2012.