Generic Literals

Florian Rabe

Jacobs University, Bremen, Germany

Abstract. MMT is a logical framework that combines the language-independence of, e.g, OPENMATH with the formal precision of type theories and logics. 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 be extended with arbitrary choices of literals. Theoretically, literals are introduced by importing a model that defines extensional interpretations for some types and function symbols. Practically, MMT is coupled with a programming language, in which these models can be defined.
We have implemented all results in the MMT system. In particular, literals and computation on them are integrated with the type checker.
We demonstrate all aspects using the dependently-typed λ-calculus as a logic and Scala as a programming language.

1 Introduction and Related Work

Literals versus Constants We call a constant **free** if it can have a different interpretation in every model, and **interpreted** if it must have the same fixed interpretation in every model. Then we could define literals as the interpreted constants of atomic types.

This works well for small finite types. For example, for the type of booleans, we would declare two interpreted constants `true` and `false`.

But for larger types, this is impractical. For example, declaring one interpreted constant for every 32-bit integer would be unreasonable. For infinite types, this would be a theoretical problem, too. While we can use theories with infinitely many constants, this greatly increases the complexity of the system, especially if we also need infinitely many axioms about these constants.

Literals versus Syntactic Values We define **syntactic values** as those expressions that are fully simplified. These expressions do not have to be atomic, e.g., if $a : A$ and $b : B$ are values, then so is $(a, b) : A \times B$. Then we can say that the literals are the atomic expressions that are also syntactic values.

Therefore, we can add literals to a language via a type-indexed family L_A of sets and the rule schema

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

For example, we would put $L_{\texttt{nat}} = \mathbb{N}$. Note that to enhance clarity, we will use the quote-**notation** "l" for literals throughout this paper, even though the concrete syntax of a practical language would of course not distinguish, e.g., $1 : \texttt{nat}$ and $1 \in \mathbb{N}$.

Literals versus Semantic Values We define **semantic values** as those objects that occur in the image of the interpretation function $[\![-]\!]^M$, i.e., models M interpret expressions as semantic values. Then we can say that literals are special expressions that have a fixed semantics in every model: Every model must interpret "l" (literally!) as l. Thus, the sets L_A occur as a fixed subset of both the syntactic values and the semantic values.

Literals are usually coupled with a set of **interpreted function** constants. For example, if we use $L_{\mathtt{nat}} = \mathbb{N}$, we expect an interpreted constant $+ : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$, which satisfies "1" + "1" $\doteq$ "2". In higher-order languages, we can think of these as literals of function type, but in general a distinction between literals and interpreted functions may be appropriate.

Related Work Programming languages and computer algebra systems primarily use interpreted constants, which makes adding literals relatively easy. Programming languages routinely provide a fixed family L_A of literals in the above sense. Typically the types A include at least booleans, bounded integers, floating point numbers, characters, and strings. Some languages like C or ML provide enumeration or inductive data types, which can be used to introduce new finite sets of literals. Computer algebra systems use essentially the same approach except for preferring unlimited precision integer and rational numbers.

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. Examples are string literals in C (converted to character arrays) or XML literals in Scala (converted to objects). Mathematica [Mat12] is interesting for offering a large variety of pseudo-literals, e.g., for images, which users input as files.

Logic-based systems like proof assistants primarily use free constants, which are constrained by axioms. This makes it more complex to add literals. The typical technique is to use inductive data types to specify the intended type of literals and then use pseudo-literals for them. For example, 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 elements of the number sets.

This approach has the disadvantage that the treatment of pseudo-literals must be built into the parser and printer by the system developer, whereas many literal-worthy types such as integer and rational numbers are defined much later by the user. In fact, it is rare that a logic features a built-in type for any literal-worthy type other than possibly booleans. One of the few examples is the primitive type of natural numbers in Martin-Löf type theory [ML74], which makes it easy to add natural number literals in implementations like Agda [Nor05]. Similarly, the TPTP family of interchange logics offers integer, rational, and real numbers as primitive types [SSCB12].

Literals also present a challenge to knowledge representation languages like MathML because there is no good canonical choice which fixed set of literals to build into the language. For example, OpenMath [BCC+04] somewhat arbitrarily fixes 4 types of literals: unbounded decimal integers (OMI), decimal

or hexadecimal IEEE floating point numbers (OMF), strings (OMSTR), and bytearrays (OMB); but it lacks bounded or arbitrary-base integers as well as characters (not to mention more exotic types like URIs, dates, or colors).

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, the set of literals is usually fixed once and for all. At best, parts of the code are parametric in the choice of literals, like the constraint domains of Twelf [PS99], but only developers can add new ones.

Contribution We design and implement a general knowledge representation language that supports generic literals, i.e., users can add new literals to any type and at any time. Thus, literals become as extensible as constants, functions, axioms, and notations. We base our design on the MMT language [RK13] and tool [Rab13], which already provide a generic type theory.

MMT uses the expression constructors of OPENMATH: constants (OMS), variables (OMV), application (OMA), and binding (OMBIND). In the past, MMT also tried to adopt the 4 OPENMATH constructors for literals, but the increased complexity (4 additional cases in the syntax) and the lack of extensibility made this awkward. Moreover, it proved very difficult to integrate these literals with MMT's generic type system: It was not clear when an MMT theory would be allowed to use a certain literal and what type and meaning it would have. This is particularly difficult in a logical framework scenario, where multiple languages with different or overlapping literals are represented in the framework.

Here, we extend the MMT syntax with a single constructor for literals: "l" where l is an arbitrary element of some semantic domain. This domain can induce "$\mathbb{N}$" and "0" as well as the typing rule $\vdash$ "0" : "$\mathbb{N}$". Thus, an MMT theory can declare, e.g., `nat` : `type` = "$\mathbb{N}$", `zero` : `nat` = "0". Clearly, in a declarative formal language, we cannot easily use arbitrary set theoretical domains. 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 allow MMT theories and Scala classes to import each other. Then a typical concrete application of our system proceeds in three steps:

1. We define an axiomatic MMT theory I, e.g., with a type `nat` and a function `plus`.
2. We treat I as an abstract Scala class and extend it by a Scala class C. C implements I, e.g., using the positive integers for `nat` and addition for `plus`.
3. When we import C into an MMT theory T, `nat` and `plus` become interpreted constants within T. In particular, T may use natural number literals, and addition is added to the equational system of T.

This design also helps combine the axiomatic semantics of free constants (as typical in proof assistants) with the computational semantics of interpreted constants (as typical in computer algebra systems). The former operates on syntactic expressions and the latter on semantic values, and literals are exactly in the intersection of the two. Thus, literals can mediate the information exchange

between systems, and MMT provides an extensible representation language for defining these interfaces.

Overview We briefly present the existing MMT type theory in Sect. 2. Then we extend it with a general notion of literals in Sect. 3 and specialize to Scala-based literals in Sect. 4. We present the practical aspects of our implementation in Sect. 5. In Sect. 6, we revisit some related work and conclude.

2 Preliminaries: The MMT Language

The MMT language was originally introduced in [RK13]. The presentation here follows the type theoretical formulation of [Rab14].

Grammar The grammar of Fig. 1 is sufficient for our purposes. A **theory** Σ is a list of **constant** declarations. These are of the form $c[: A][= t]$ where c is an identifier, A is its **type**, and t its **definiens**, both of which are optional. A **context** Γ is very similar to a theory and declares typed variables.

Theory	Σ	::=	$\cdot \mid \Sigma,\ c[: t][= t]$
Context	Γ	::=	$\cdot \mid \Gamma,\ x : t$
Term	t	::=	$c \mid x \mid c(\Gamma; t^*)$

Fig. 1. MMT Grammar

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 subsume both binding and application: c is the constructor of the term, Γ declares the bound variables, and the t_i are the arguments. Both bound variables and arguments are optional, in particular $\Gamma = \cdot$ yields the case of non-binding operators (OMA) and $\Gamma \neq \cdot$ and $n = 1$ yields the case of binders (OMBIND).

theory `LF`

`type`	
`Pi`	$\#\ \{\, x_1 : t_1 \,\}\, t_2$
`lambda`	$\#\ [\, x_1 : t_1 \,]\, t_2$
`apply`	$\#\ t_1\ \ t_2$
`arrow`	$\#\ t_1 \to t_2$

theory `Nat`

`prop`	$: \texttt{type}$
`nat`	$: \texttt{type}$
`succ`	$: \texttt{nat} \to \texttt{nat}$
`plus`	$: \texttt{nat} \to \texttt{nat} \to \texttt{nat} \quad \# \ t_1 + t_2$
`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)

Like OPENMATH objects, MMT terms subsume all expressions, types, kinds, universes, etc. – MMT relegates any distinction between terms to the type system. Also, both are generic in the sense that there are no predefined constants in the grammar. Therefore, when representing a language in MMT, we first have to declare a theory L with one constant for each primitive of L. Then we define L-**theories** as the MMT theories L, Σ. We call the constants of L **logical** and the ones of Σ **non-logical**.

As running examples, we will use the dependently typed λ-calculus LF [HHP93] as a language and the natural numbers as an LF-theory.

Example 1 (LF as an MMT *Theory).* LF is the theory shown on the left of Fig. 2 where we use # to indicate the notations we will use to form complex terms. For example, the abstract syntax for a λ-abstraction is $\texttt{lambda}(x : A; t)$ where $x : A$ the single variable binding, and t the single argument. The concrete syntax is $[x : A]t$. 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.

Type System The main **judgments** of MMT are given in Fig. 3.

$\Gamma \vdash_\Sigma t : A$	t has type A
$\Gamma \vdash_\Sigma t = t'$	t and t' are equal

Fig. 3. Judgments of MMT

There is no need to introduce the type system in detail here, and we refer to [Rab14]. For our purposes, it is sufficient to know that the type system is generic as well: MMT itself only provides structural rules for congruence, α-equality, and lookup. For example, the lookup rules say that if $c : A = t$ in Σ, then $\vdash_\Sigma c : A$ and $\vdash_\Sigma c = t$.

All language-specific rules must be provided separately, and we define an MMT-**language** as a pair of an MMT theory and a set of rules for the MMT judgments:

Example 2 (LF as an MMT *Language).* We extend LF to an MMT language by adding the usual rules for λ-abstraction including

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

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

where $B[t]$ denotes substitution.

We do not need any rules for the LF-theory `Nat` because it inherits all rules from LF.

Given an MMT language, the MMT system generically implements type reconstruction for it. Details are given in [Rab14], which also defines LF modulo equational theory as an MMT language.

3 Literals as Semantic Values

Here we are not interested in the semantics of MMT languages L and in models of L-theories Σ per se. But we will use them to define literals below. Therefore, we introduce some general semantic notions for MMT before we use them to define literals.

3.1 Semantics of MMT Languages

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.

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 language L 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.

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 language L consists of a semantic domain and a set of interpretation rules. The interpretation rules must extend every Σ-model M to an interpretation function $[\![-]\!]^M$ that maps Σ-terms 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, we use the following interpretation rules to define $[\![t]\!]^M \in U$ for every term t:

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

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

We omit the accordingly for terms constructed using `Pi`.

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

$$\text{if } \vdash_\Sigma t : A \quad \text{ then } \quad [\![t]\!]^M \hat{:} [\![A]\!]^M$$

$$\text{if } \vdash_\Sigma t = t' \quad \text{ then } \quad [\![t]\!]^M \hat{=} [\![t']\!]^M$$

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

3.2 Literals as Internalized Semantics

We will now internalize semantic domain, semantics, and models into the MMT language. This will give us the ability to concisely define literals.

We **internalize a semantic domain** $(U, \hat{:}, \hat{=})$ by treating every element of U as a literal. Concretely, we extend the MMT grammar from Fig. 1 with **one new production** – a constructor $"l"$ that introduces literals:

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

Moreover, we add **two new inference rules** to the MMT type system:

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

Because we extended the grammar, we also need a **new interpretation rule** so that models interpret every literal as itself:

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

Allowing any $l \in U$ as a literal may appear insanely expressive. Indeed, if we use the semantic domain from Ex. 3, the MMT terms do not even form a set anymore, let alone a recursively enumerable one. However, this level of expressivity provides a good abstraction to define the general concepts.

Note that at this point, there are no typing rules that relate literals to other non-literal terms. Thus, our literals are almost inconsequential. That changes with the next definition: We **internalize a semantics** for a language L by stating 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:

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

$$\frac{\vdash_\Sigma f = "F" \qquad \vdash_\Sigma t = "T"}{\vdash_\Sigma f\ t = "F(T)"}$$

We do not have a corresponding rule for the binders `lambda` and `Pi`. But the above is sufficient to prove Thm. 2 below for the practically important case of typed first-order functions.

Given an internalized semantics, we call t a **value** if $\vdash_\Sigma t = "l"$ for some l.

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

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

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

Fig. 4. Vectors in LF with Natural Number Literals

Note that $(**)$ and Thm. 1 are inverse properties.

Finally, we can **internalize models**. Let us fix a semantic domain $(U, \hat{:}, \hat{=})$ and an internalized semantics for a language L. For an L-theory Σ and a Σ-model M, the Mmt theory "M" contains for every Σ-constant $c : A[= t]$ the constant $c : A = \text{"}c^M\text{"}$. The following theorem shows that we can indeed treat models as special theories:

Theorem 2. *Consider a language L with a sound semantics and 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 $\vdash_{\text{"}M\text{"}} \text{"}c^M\text{"} : A$ for every Σ-constant $c : A$.

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

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

3.3 Defining Specific Literals

We will now assume an internalized semantics and use it as a general environment in which we can define specific sets of literals. The central idea is to give a model M to define the interpreted constants and then to work in the theory "M".

Consider a language L with an internalized semantics. We fix an L-theory Σ and a Σ-model M.

Then we call Θ an L-theory with literals from M if "M", Θ is an L-theory. We call the constants of "I" **interpreted** and the ones of Θ **free**. And we call N a Θ-**model** if it is a Σ, Θ-model that agrees with M on Σ.

Thus, the theory Σ acts as the interface between Θ (which uses the literals) and "M" (which provides them).

The Mmt module system already provides the concrete syntax to write the theory "M", Θ by including "M" into Θ:

Example 7. We work with the internalized semantics of LF from Ex. 6. Let `Vec` be the LF-theory of Fig. 4. It includes "`StdNat`" from Ex. 4 (and thus also

includes `Nat` from Ex. 1). The left part introduces $\texttt{vec}\,n$ 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 constants are valid only because the internalization of the semantics implies that $\vdash_{\texttt{Vec}} \texttt{vec}\,2 = \texttt{vec}\,(\texttt{succ}(\texttt{succ}(0)))$ and $\vdash_{\texttt{Vec}} \texttt{vec}\,4 = \texttt{vec}\,(2+2)$ Such an integration of interpreted constants into a dependent type system is non-trivial even for a fixed built-in set of literals.

It remains to define theories like "`StdNat`" in a practical way. 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 of such definitions.

Then we obtain a semantic domain $(U, \hat{:}, \hat{=})$ as follows. U consists of the symbol `type`, all Scala types over G, and all side effect-free 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 Σ makes use of dependent types. Most importantly, this applies to axioms. Apart from axioms, however, most operations on literals that we want to implement in practice are simply-typed, often 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 A^{sq} arises as the quotient of A^s by r.

This permits defining 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 $p \hat{:} \texttt{PER}$ if $p : \texttt{PER}$; and $p \hat{=} p'$ if p and p' evaluate to the same value.

And we put $(p, u) \hat{:} p$ if $p.\texttt{valid}(u)$; and $(p, u) \hat{=} (p, v)$ 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 literals for 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. Thus, the Scala type system can check the soundness of a model only partially.

5 Implementing Literals in the MMT System

5.1 Internalizing a Computational Semantics

Our reason for using Scala is that it underlies our implementation of MMT. Therefore, we can extend our implementation with Scala-based literals seamlessly. We can internalize the semantics of Scala from Ex. 12 by making the 5 changes described below, which adapt not only the abstract syntax but also the type checker.

1) We add a feature to the MMT build tool: It exports all LF-theories T as abstract Scala classes C. C is defined such that the concrete Scala classes M inheriting from C are exactly the Scala-models of T. C and M correspond to the two classes of Ex. 10.

2) We add the include declarations that we have already used in Ex. 7. If an MMT theory includes "M", MMT locates the Scala class M and adds its definitions to the type system.

3) We add a new constructor `OMLIT` for MMT terms: It takes a PER p on a Scala type and a value $u : p.\texttt{univ}$. Thus, $\texttt{OMLIT}(p, u)$ implements the literal "u".

We do not add corresponding constructors "l" for arbitrary Scala expressions l. Therefore, the only way to reference a Scala expression is 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 expression "c^M".

This restriction has two desirable effects: (i) Scala expressions can appear in MMT terms only if they have been explicitly imported. (ii) Models M implementing theories T remain transparent: All we know about M is what we can infer from extensional manipulation of the interpretations c^M.

4) The MMT type checker accepts a literal $\texttt{OMLIT}(p, u)$ only if (i) $p = c^M$ for some "M" imported into the current theory, and (ii) $p.\texttt{valid}(u)$ is true. In that case, the inferred type is c. The MMT equality checker applies normalization in the obvious way.

5) The MMT simplifier implements the rule

$$\frac{l = c^M(l_1, \ldots, l_n) \qquad \text{"}M\text{" imported into } \Sigma}{\vdash_\Sigma c\text{"}l_1\text{"} \ldots \text{"}l_n\text{"} = \text{"}l\text{"}}$$

Because the simplifier is fully integrated with the type system, this makes the type system computation-aware.

We stress that the models M can be written and included by users at run time just like normal MMT theories. MMT dynamically adds the interpreted constants to its type system.

5.2 Lexing Rules

Literals are only practical if we also adapt our lexer appropriately. This can be a major hurdle in systems that are designed for a fixed set of lexing rules. However, MMT already uses an extensible lexer anyway, where the lexing rules are chosen

depending on 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 parametric lexing rules for the most important cases that can be reused easily. In particular, these include quoted and digit-based literals.

```
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 c (cons c nil)
test1  : vec 4 = concat test0 test0
```

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

Our integration of literals into the type system is not perfect at this point: A difficulty arises when interpreted constants are mixed with free constants and variables. Consider the following example:

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

$$\texttt{head} : \{n : \texttt{nat}\}\texttt{vec}\,(\texttt{succ}\,n) \to a \qquad test2 : a = head\,test0$$

Type-checking the declaration $test2$ 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.

The generic MMT type checker [Rab14] is already highly extensible by various kinds of rules. Therefore, it is easy to overcome this limitation by adding rules. We allow models to couple an implementation c^M with a (possibly partial) implementation of the inverse function $c^{M\,-1}$. If provided, MMT uses $c^{M\,-1}$ to add the following rule to the type checker:

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

If we implement $\texttt{succ}^{\texttt{StdNat}\,-1}$ as the predecessor function in Ex. 15, this rule suffices to solve the above meta variable as $X = \texttt{succ}^{\texttt{StdNat}\,-1}(2) = 1$.

6 Conclusion

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 users can declare new 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 as well as the associated lexing and computation rules are subject to the MMT module system in the same way as constants, axioms, and notations. Thus, different languages represented in MMT can use different literals.

We believe that the applications of our work go beyond our present focus on literals. Firstly, our design is a promising candidate for combining logical reasoning with computer algebra: Here the literals comprise the intersection of the logical expressions and the computational values. Secondly, if we generalize our design to structured literals (literals that may contain other terms as subterms), it may be possible to splice arbitrary external language features into formal systems.

Related Work The theoretical aspect of our work shares a basic idea with biform theories [FvM03]. In a theory $\text{"}M\text{"}, \Theta$, we can think of Θ as the axiomatic/intensional and of $\text{"}M\text{"}$ 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 $\text{"}t\text{"}$ for terms t, which can be seen as quoted terms. Structured literals would correspond to quasi-quotation.

References

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.

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.

Mat12. Mathematica, 2012.

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

Nor05. U. Norell. The Agda WiKi, 2005. http://wiki.portal.chalmers.se/agda.

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. *Lecture Notes in Computer Science*, 1632:202–206, 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 Generic Type Theory. see http://kwarc.info/frabe/Research/rabe_mmttypetheory_14.pdf, 2014.

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, 1985.