The Scala-REPL + MMT as a Lightweight Mathematical User Interface

Mihnea Iancu, Felix Mance, and Florian Rabe

Jacobs University, Bremen, Germany

Abstract. Scala is a general purpose programming language that includes a read-eval-print loop (REPL). MMT is a general representation language for formal mathematical knowledge implemented in Scala. Independent recent developments permit combining them into an extremely simple user interface.

Firstly, Scala introduced string interpolation – a convenient syntax that permits escaping back and forth between strings and arbitrary Scala expressions (while preserving type safety). Secondly, MMT introduced a notation-based text syntax and a rule-based evaluation engine for its mathematical objects (which are based on OpenMath).

Combining these, users can enter and work with MMT objects in the Scala-REPL with so little overhead that it essentially behaves like a dedicated MMT-REPL – except for also providing the full power of Scala. Implicit conversions (e.g., between integers represented in MMT and Scala integers) further blur the distinction between meta- and object language.

MMT is highly extensible: Users can add new type systems and logics as well as new theories and notations and evaluation rules. Thus, we obtain a REPL-style interface for any language represented in MMT with essentially no effort.

1 Representing Languages in MMT

MMT [RK13] is a generic, formal module system for mathematical knowledge and is a basis for foundation-independent knowledge representation.

The MMT language is designed to be applicable to a large collection of declarative formal base languages and all MMT notions are fully abstract in the choice of the base language. Therefore, MMT focuses on foundation-independence, scalability and modularity.

Every MMT declaration is identified by a canonical, globally unique URI.

MMT symbol declarations subsume most semantically relevant statements in declarative mathematical languages including function and predicate symbols, type and universe symbols, and — using the Curry-Howard correspondence — axioms, theorems, and inference rules. Their syntax and semantics is determined by the foundation, in which MMT is parametric.

The MMT API [Rab13] is a Scala-based [OSV07] open source implementation of the MMT language and of a number of knowledge management services for it.

Declaring Symbols A **theory** declaration $T = \{Sym^*\}$ introduces a theory with name T containing a list of symbol declarations. A **symbol** declaration $c : \omega = \omega' \# \nu$ introduces a symbol named c with **type** ω, **definiens** ω' and **notation** ν (all of which are optional).

Terms ω over a theory T are formed from symbols $\mathsf{OMS}(T?c)$ declared in T, bound variables $\mathsf{OMV}(x)$, applications $\mathsf{OMA}(\omega, \omega_1, \ldots, \omega_n)$ of a function ω to a sequence of arguments, bindings $\mathsf{OMBIND}(\omega, X, \omega')$ using a binder ω, a bound variable context X, and a scope ω' as well as integers $\mathsf{OMI}(i)$ and floats $\mathsf{OMF}(f)$ where i and f are integers and floats, respectively. This is a fragment of the OpenMath language [BCC+04].

Remark 1. For readability we will write $T?c$ instead of $\mathsf{OMS}(T?c)$ in the following. Furthermore, we will write c instead of $T?c$ when the theory T is clear from the context.

Example 1. Figure 1 shows an Mmt theory *Lists*, based on the logical framework *LF* [HHP93]. *Lists* declares natural numbers and lists as well as additional operations on them (plus for naturals and append for lists). Using the Mmt notation language, described below in paragraph 1, we also declare the usual notations: infix + , :: and ::: for plus, cons, and append, respectively.

theory *Lists* **meta** *LF*

tp	:	`type`
tm	:	$tp \to$ `type` $\#\ tm\ \mathcal{A}_1$
nat	:	tp
zero	:	$tm\ nat\ \#\ o$
succ	:	$tm\ nat \to tm\ nat\ \#\ s\ \mathcal{A}_1$
plus	:	$tm\ nat \to tm\ nat \to tm\ nat\ \#\ \mathcal{A}_1 + \mathcal{A}_2$
list	:	$tp \to tm\ nat \to tp\ \#\ list\ \mathcal{A}_1\ \mathcal{A}_2$
nil	:	$\{A\}\ tm\ list\ A\ zero$
cons	:	$\{A, N\}\ tm\ A \to tm\ list\ A\ N \to tm\ list\ A\ (s\ N)$
		$\#\ \mathcal{A}_3 :: \mathcal{A}_4$
concatenate	:	$\{A, M, N\}\ tm\ list\ A\ M \to tm\ list\ A\ M \to tm\ list\ A\ (M + N)$
		$\#\ \mathcal{A}_4 ::: \mathcal{A}_5$

Fig. 1. Theory declaration in Mmt

Remark 2. In theory *Lists* from figure 1, all symbols and notations declared in its meta-theory *LF* are available.

Specifically, we use the symbols for `type`, `arrow` (with notation $\mathcal{A}_1 \to \mathcal{A}_2$), and the `Pi` binder (with notation $\{\ \mathcal{V}_1\ \}\ \mathcal{S}_2$).

Adding Notations Notations act as the parsing and printing rules that transform between abstract syntax and text-based concrete syntax. A notation is a sequence of notation elements which can be either delimiters (i.e. strings), argument markers ($\mathcal{A}_n$), variables $\mathcal{V}_n$ and scopes $\mathcal{S}_n$ where n is a number representing

an argument position in the abstract syntax. Variables and scopes are in principle used for binders (e.g. the `Pi` binder in *LF*), and the majority of symbol notations only need delimiters and arguments. For example, the notation for infix addition is given as $\mathcal{A}_1 + \mathcal{A}_2$. It implies that conjunction is binary and constructs the application object.

It is typical to omit arguments if their value can be inferred from the remaining arguments. We call this *implicit* arguments. Therefore, we introduce a simple convention: if a component number n is absent in a notation but a higher number is present, then the missing component is assumed to be an implicit argument $\mathcal{A}_n$. For instance, we use the notation $\mathcal{A}_3 :: \mathcal{A}_4$ for *cons* above meaning that the first two arguments (the contained type A and the size N) are implicit.

We omit here the details regarding the parsing algorithm and only point out that our notation language is more complex and also covers sequence arguments. Furthermore, notations technically also include an integer precedence, which is used to resolve ambiguities when multiple notations are applicable.

Adding Evaluation Rules MMT allows users to declare evaluation rules for each symbol [KMR13]. Intuitively, the evaluation rule of a symbol c acts as the implementation of the computational semantics of c.

We give evaluation rules for *plus* and *append*, declared above in theory *Lists*. For example, the rule for *plus* is declared as an MMT assignment which maps *plus* to a λ-expression (λ together with its notation "=>" are declared in the *ScalaOM* meta-theory). The λ-expression maps the arguments x, y of *plus* to a Scala code snippet written between MMT escape characters (shown here as quotes).

A module within the MMT-API, called the Universal OpenMath Machine (UOM), translates the theory and the view to a Scala object and trait, respectively. Each MMT constant (e.g. *plus*) is translated to a Scala object which acts as the Scala constructor and pattern matcher for that constant. As a result, instead of writing e.g. $OMA(OMS(\text{"}plus\text{"}), a, b)$ we can directly write $plus(a, b)$ to construct an MMT term in Scala. Furthermore, the UOM translates each λ-expression $vars$ => $snippet$ to a function whose arguments are $vars$ and whose body is $snippet$. These functions are stored by the UOM in a *rule store* S. When given an expression E, the UOM exhaustively applies the rules in S on E.

Example 2 (Continuing example 1). We give evaluation rules for *plus* and *append*, declared above in theory *Lists*. The rules are declared as MMT assignments which map *plus* and *append* to Scala code snippets.

Given the evaluation rules from figure 2, we can use the UOM to evaluate $\mathsf{OMA}(plus, \mathsf{OMA}(succ, zero), \mathsf{OMA}(succ, \mathsf{OMA}(succ, zero)))$ and reduce it to $\mathsf{OMA}(succ, \mathsf{OMA}(succ, \mathsf{OMA}(succ, zero)))$ (i.e. in decimal notation $1 + 2$ to 3).

2 String Interpolation in Scala

Scala [OSV07] is a programming language that integrates the functional and object-oriented paradigms. It compiles to Java bytecode and can be seamlessly

```
view Impl : Lists → ScalaOM =
  plus     = (x : Term, y : Term) => "scala
             y match {
               case zero => x
               case succ(z) => succ(plus(x, z))
               case _ => plus(x, y)
             }
  "

  append = (a : Term, m : Term, n : Term, ls1 : Term, ls2 : Term) "scala
             ls1 match {
               case nil => ls2
               case cons(a, n, h, t) => cons(a, n, h, append(a, m, n, t, ls2))
               case _ => append(a, m, n, ls1, ls2)
             }
  "
```

Fig. 2. Evaluation rules in MMT

integrated with existing Java libraries. Scala also includes a read-eval-print loop for interactive evaluation of expressions.

Scala recently introduced **processed strings** [Ode12] as a generalization of string literals. A processed string consists of an identifier followed by a string literal within which two forms of escaping are valid: (i) `$var` for individual variables and (ii) `${ expr }` for complex expressions where `$var` is syntactic sugar for for `${ var }`.

Thus, a processed string is of the form:
$\texttt{id"text}_0\ \texttt{\$\{ expr}_0\ \texttt{\} text}_1\ \cdots\ \texttt{\$\{ expr}_n\ \texttt{\} text}_n\texttt{"}$
where `id` defines the interpolation function that processes the string.

Besides a few built-in interpolation functions, Scala permits users to declare arbitrary custom interpolation functions `id`. These must take as arguments a sequence of strings (the parts $\texttt{text}_i$) and a sequence of typed expressions (the expressions $\texttt{expr}_i$). String interpolation is type-safe: the argument types of `id` define what expressions $\texttt{expr}_i$ are legal.

3 An MMT-REPL

Interpolating MMT Expression We implement two string processors for MMT terms. This allows us to combine and nest MMT and Scala expressions and use MMT-specific services directly from the Scala shell.

(i) `mmt`, which calls the MMT parser to evaluate an interpolated string literal into an MMT object and (ii) `uom`, which uses `mmt` but additionally calls the UOM simplifier to perform computation on the resulting term.

`mmt` takes a sequence of strings $\texttt{text}_i$ and a sequence of MMT-terms $\texttt{term}_i$. It concatenates the strings $\texttt{text}_i$ and inserts a fresh free variable $\texttt{x}_i$ for every

$\texttt{term}_\texttt{i}$. The resulting string is parsed into an MMT term as usual, and afterwards each $\texttt{x}_\texttt{i}$ is substituted with $\texttt{term}_\texttt{i}$.

For example, if the Scala variables a and b hold the MMT-term $\mathsf{OMI}(3)$ and $\mathsf{OMI}(5)$, respectively, then $\texttt{mmt}"\$a + \pi + \$b"$ is interpolated to the MMT-term $\mathsf{OMA}(plus, \mathsf{OMI}(3), \pi, \mathsf{OMI}(5))$. And, given appropriate evaluation rules as described in [KMR13], $\texttt{uom}"\$a + \pi + \$b"$ yields $\mathsf{OMA}(plus, \mathsf{OMI}(8), \pi)$.

We can also escape back and forth between Scala and MMT. For example, if $substitute(t, n, s)$ is the MMT-function for substituting the variable named n in the term t with s, then

$$\texttt{mmt}"\pi + \${substitute(\texttt{mmt}"x + x", "x", a)}"$$

yields $\mathsf{OMA}(plus, \pi, \mathsf{OMI}(3), \mathsf{OMI}(3))$.

In general, this has the effect that $\texttt{mmt}"..."$ escapes from Scala into MMT, and ${...} escapes from MMT into Scala.

An MMT-REPL Based on the string processors described above we can use MMT and the UOM directly in the Scala shell and to nest and combine MMT terms and Scala expressions.

Example 3 (Continuing examples 1 and 2). Using the `uom` interpolator we can integrate MMT notations and evaluation rules inside the Scala environment. For instance, in the example below we use the notations and evaluation rules for symbols *plus* and *concatenate* to perform operations on MMT terms from the Scala REPL.

```
> uom"s o + s (s o)"
s (s (s o)).
> uom"(o :: (s o) :: nil) ::: (o :: nil)"
(o :: (s o) :: o :: nil).
```

Note that, in the listing above, we show the result using MMT notations (and not the abstract syntax) for readability but, technically, it now also contains the inferred values for the implicit types and arguments. For example, the subterm $o :: nil$ from the list above corresponds to $\mathsf{OMA}(cons, nat, zero, zero, nil)$ in abstract syntax. Furthermore, its type is inferred as $\mathsf{OMA}(list, nat, \mathsf{OMA}(succ, zero))$ (i.e. list of type *nat* and length 1)

While notations do increase usability and readability, unary natural numbers remain awkward to work with. But we also allow users to directly write numbers in MMT concrete syntax. They are automatically parsed into the OPENMATH counterparts: OMI for integers and OMF for floats.

Example 4. linalg2?vector refers to an MMT declaration that implements the OPENMATH symbol for vectors. It also has the notation $\langle\ \mathcal{SA}_1\ \rangle$ where $\langle$ and $\rangle$ are delimiters and $\mathcal{SA}$ is a sequence argument representing an arbitrary number of arguments from the abstract syntax (comma separated). Therefore, we can use the notation to construct vectors of integers and the UOM to perform computations on them (the evaluation rules for addition of OPENMATH integers, floats and vectors are already implemented [KMR13]).

```
> mmt"⟨1,2,3⟩"
OMA(linalg2?vector, OMI(1), OMI(2), OMI(3))
> uom"⟨1,2,3⟩ + ⟨2,3,4⟩"
OMA(linalg2?vector, OMI(3), OMI(5), OMI(7))
```

Implicit Conversions Furthermore, we use Scala implicit conversions to automatically convert Scala terms to corresponding MMT terms (e.g from Scala integers to MMT/OPENMATH integers).

Example 5. After implementing the following implicit conversion from Scala integers to MMT/OPENMATH integers:

```
implicit def int2OM(i: Int) = OMI(i)
```

we can interpolate Scala integers into MMT-specific string literals.

```
> var x = 3
> mmt"$x + ${4 - 2}"
OMA(arith1?plus, OMI(3), OMI(2))
> uom"$x + ${4 - 2}"
OMI(5)
```

Moreover, we can use implicit conversions in the opposite direction to be able to use the result of computations performed by MMT and the UOM in Scala.

Example 6. After implementing the implicit conversion from MMT/OPENMATH to Scala integers we can use directly use them inside Scala expressions. In the example below, $*$ and $-$ are Scala operators, $+$ is the notation of the MMT symbol *plus* and the final result is a Scala integer.

```
> 7 * uom"$x + ${4 - 2}"
35
```

Implicit conversion also works for more complex notions as long as there is a Scala counterpart. Moreover, since the UOM automatically constructs Scala objects for each MMT symbol declaration, we can easily refer to and construct MMT objects from within Scala.

Example 7. For instance, we can define implicit conversions between MMT and Scala vectors. In the listing below, Scala automatically converts the MMT term `vect` into a Scala vector and then during the map operation, each MMT integer into the corresponding Scala integer finally yielding a Scala vector as a result.

```
> var vect = uom"vector 1 2 3"
OMA(linalg2?vector, OMI(1), OMI(2), OMI(3))
> vect.map(x => 1 + x)}
Vector(2, 3, 4)
```

Note that, even though we only gave simple, self-contained examples here, any Scala or Java library can be used to process or render the results of computations done by MMT and the UOM. The output can then be fed back into MMT for further computation. Moreover, with implicit conversions this integration can be done seamlessly and with almost no overhead.

4 Conclusion

The combination of the MMT notation language and Scala string interpolation yields an input language for mathematical objects that permits arbitrary escaping between Scala and MMT. This turns the Scala REPL into an MMT-REPL that gives users access to notations and computation rules defined in MMT, the syntax manipulation functions defined in the MMT API, and any custom function defined in arbitrary Java/Scala packages.

Clearly, this Scala/MMT-REPL does not give us a powerful computer algebra system. But, it is interesting to consider what is missing. Indeed, we can easily imagine building a practical CAS on top of it simply by adding symbols, notations, and evaluation rules.

This has the appeal that users can write mathematical algorithms using a strongly typed and widely used general purpose programming language. Moreover, it is easy to integrate with existing systems. Existing CASs can be used by simply adding special symbols for them and adding computation rules that simplify, for example, $\mathsf{OMA}(maple, t)$ into the result of calling Maple on t.

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.

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 D. Aspinall, J. Carette, C. Lange, and W. Windsteiger, editors, *Intelligent Computer Mathematics*. Springer, 2013. to appear.

Ode12. Martin Odersky. SIP 11: String interpolation and formatting. http://docs.scala-lang.org/sips/pending/string-interpolation.html, January 15, 2012.

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

Rab13. F. Rabe. The MMT API: A Generic MKM System. In D. Aspinall, J. Carette, C. Lange, and W. Windsteiger, editors, *Intelligent Computer Mathematics*. Springer, 2013. to appear.

RK13. F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, pages 1–95, 2013. to appear; see http://kwarc.info/frabe/Research/RK_mmt_10.pdf.