A Universal Machine for Biform Theory Graphs

Michael Kohlhase, Felix Mance, and Florian Rabe

Computer Science, Jacobs University Bremen
initial.lastname@jacobs-university.de

Abstract. Broadly speaking, there are two kinds of semantics-aware assistant systems for mathematics: proof assistants express the semantic in logic and emphasize deduction, and computer algebra systems express the semantics in programming languages and emphasize computation. Combining the complementary strengths of both approaches while mending their complementary weaknesses has been an important goal of the mechanized mathematics community for some time.
We pick up on the idea of biform theories and interpret it in the MMT/OMDoc framework which introduced the foundations-as-theories approach, and can thus represent both logics and programming languages as theories. This yields a formal, modular framework of biform theory graphs which mixes specifications and implementations sharing the module system and typing information.
We present automated knowledge management work flows that interface to existing specification/programming tools and enable an OpenMath Machine, that operationalizes biform theories, evaluating expressions by exhaustively applying the implementations of the respective operators. We evaluate the new biform framework by adding implementations to the OpenMath standard content dictionaries.

1 Introduction

It is well-known that mathematical practices – conjecturing, formalization, proving, etc. – combine (among others) axiomatic reasoning with computation. Nevertheless, assistant systems for the semantics-aware automation of mathematics can be roughly divided into two groups: those that use *logical languages* to express the semantics and focus on *deduction* (commonly called **proof assistants**), and those that use *programming languages* to express the semantics and focus on *computation* (commonly called **computer algebra systems**). Combining their strengths is an important objective in mechanized mathematics.

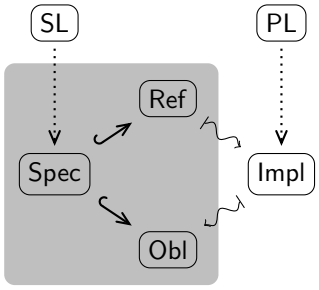
Our work is motivated by two central observations. Firstly, combination approaches usually take a deduction or computation system and try to embed the respective other mode into its operations, e.g., [HT98; DM05].

Secondly, most of these systems are usually based on the *homogeneous* method, which fixes one foundation (computational or deductive) with all primitive notions (e.g., types, axioms, or programming primitives) and uses only conservative extensions (e.g., definitions, theorems, or procedures) to model domain objects.

In this paper, we want to employ the *heterogeneous* method. It focuses on defining theories that may introduce new primitive notions, and considers truth relative to a theory. It optimizes reusability by stating every result in the weakest possible theory and using *theory morphisms* to move results between theories in a truth-preserving way. This is often called the *little theories* approach [FGT92]. In computational systems this is mirrored by using programming languages that relegate much of the functionality to an extensible library infrastructure.

Motivated by the success of the little theories approach, [Far07] proposes the concept of *biform theories*, an extension of axiomatic theories with *transformers*: named algorithms that implement axiomatically specified function symbols. We follow the intuition of heterogeneous biform theories, but interpret them in our MMT system [RK13], which extends the heterogeneous method with techniques from logical frameworks and the foundation-as-theories paradigm.

To see the potential of this approach, we contrast it with two common homogeneous approaches to formal software engineering using a (deductive) specification and a (computational) implementation: *i*) in *program verification*, the correctness of an existing program π is verified by proving safety properties p about π given a domain description D, *ii*) in *program synthesis*, a program π is extracted from a constructive proof of existence for p over D.



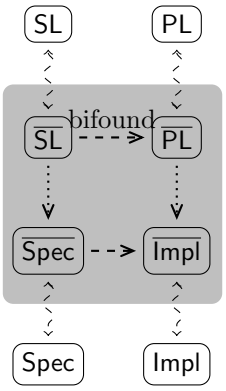
In both cases, the proofs are carried out in a theory Spec of the specification logic SL, and the implementation Impl is written in a programming language PL. This yields the situation on the right where dotted arrows denote the written-in relation. In program synthesis, Spec is extended (hooked arrows) to a refined specification Ref, which is transformed into a program (snaked arrow). In program verification, the program is transformed into an SL proof obligation that is then verified. Both employ non-trivial generation steps between PL and SL which cross the borders between the deduction system (the gray area) and the computational system.

Similarly, we find dual approaches that emphasize PL over SL. SML-style module systems and object-orientation can be seen as languages that transform parts of SL (namely the type system but not the entailment system) into PL. An example is the transformation of SL=UML diagrams into PL=Java stubs, which are then refined to a Java program. Advanced approaches can transform the whole specification into PL by enriching the programming language as in [KST97] or the programming environment as in [Ahr+05].

Following our foundations-as-theories paradigm, our interest is in representing both PL and SL as MMT theories $\overline{\mathsf{SL}}$ and $\overline{\mathsf{PL}}$. In MMT, we can represent the dotted lines explicitly using the *meta-theory* relation, see the diagram below. Here the gray area is a formal diagram in the category of MMT theories. The foundational theories $\overline{\mathsf{SL}}$ and $\overline{\mathsf{PL}}$ declare the primitive notions of the languages they represent; and relatively simple parsing/generation mappings

(dashed snaked lines) transform specifications SL and implementations Impl into corresponding MMT theories $\overline{\mathsf{Spec}}$ and $\overline{\mathsf{Impl}}$.

Thus, all the interesting transformations occur inside the MMT system (the gray area) making the process transparent and accessible to machine support. In particular, as we will discuss below, the relation of a program implementing a specification can be represented using MMT theory morphisms (dashed lines) from the specification to the implementation.



Our interest at this point is not (yet) the formal deduction about the correctness of programs. Instead, we focus on enabling such *biform theory graphs* in MMT and on integrating the computational knowledge with the existing MMT knowledge representation framework. In particular, we do not provide a formal definition of the meaning of the computational knowledge other than linking symbols to algorithms via theory morphisms.

The strongest applications arise when SL and PL share language features. Often they share the language for the formation of types and expressions, which SL enriches with deductive primitives and PL and with computational primitives. MMT can leverage this: For example, the types of Spec symbols give rise to automatically generated method stubs in Impl, and examples give rise to automatically generated test cases in Impl. Note that this sharing does not have to be literal sharing – it is sufficient to give an MMT theory morphism $\mathsf{bifound} : \overline{\mathsf{SL}} \to \overline{\mathsf{PL}}$ that mediates the sharing. In particular, bifound may translate a rich type system of SL to a simpler on in PL.

As a computational backend, we develop what we call the *universal machine*. It extends MMT with a component that collects the individual implementation snippets occurring in a biform MMT theory graph and integrates them into a rule-based rewriting engine. In keeping with the spirit of MMT, our approach attempts to be *generic* and *extensible*: Any theory of any logic can be coupled with any implementation in any programming language. The universal machine keeps track of these and performs computations by applying the available rules.

We evaluate our infrastructure in an extensive case study which we will use as a running example throughout the paper. We represent a collection of OpenMath content dictionaries in MMT (i.e., SL = OpenMath) and provide implementations for the symbols declared in them using the programming language Scala (i.e., PL = Scala). The resulting biform theory graph integrates OpenMath CDs with the Scala code snippets implementing the symbols.

2 Representing Languages in MMT

In this section we introduce the MMT language and directly apply it to modeling the pieces of our running example – introducing a biform MMT theory graph for OpenMath and Scala. Notably, this modeling (the dashed snaked arrows above) is very simple and palpably adequate.

2.1 The MMT Language and System

MMT [RK13] is a knowledge representation format focusing on modularity and logic-independence. It is accompanied by the MMT API, which implements the language focusing on scalable knowledge management and system interoperability. For our purposes, the simplified fragment of MMT given in Figure 1 suffices.

```
Module    ::= theory T : ModuleName Statement*
            | view V : ModuleName → ModuleName Statement*
Statement ::= constant c [: Term] [= Term] [#Notation]
            | include ModuleName
Term      ::= c | x | number | OMA(Term+) | OMBIND(Term; x+; Term)
Notation  ::= (number[string...] | string)*
```

Fig. 1. A Fragment of the MMT Grammar

An MMT *theory* theory $T : M\ \Sigma$ defines a theory T with meta-theory M consisting of the statements in Σ. The *meta-theory* relation between theories is crucial to obtain logic-independence: The meta-theory gives the language, in which the theory is written. For example, the meta-theory of a specification is the specification logic, and the meta-theory of a program is the programming language – and the logic and the programming language are represented as MMT theories themselves (possibly with further meta-theories). Thus, MMT achieves a uniform representation of logics and programming languages as well as their theories and programs.

MMT theories form a category, and an MMT view $V : T_1 \to T_2\ \Sigma$ defines a theory morphism V from T_1 to T_2 consisting of the statements in Σ. In such a view, Σ may use the constants declared in T_2 and must declare one definition for every definition-less constant declared in T_1. Views uniformly capture the relations "T_2 interprets/implements/models T_1". For example, if T_1 represents a specification and T_2 a programming language, then views $T_1 \to T_2$ represent implementations of T_1 in terms of T_2 (via the definitions in Σ).

Theories and views are subject to the MMT module system. Here we will restrict attention to the simplest possible case of unnamed inclusions between modules: If a module T contains a statement include S, then all declarations of S are included into T.

Within modules, MMT uses constants to represent atomic named declarations. A constant's optional type and definiens are arbitrary terms. Due to the freedom of using special constants declared in the meta-theory, a type and a definiens are sufficient to uniformly represent diverse statements of formal languages such as function symbols, examples, axioms, inference rules. Moreover, constants have an optional notation which is used by MMT to parse and render objects. We will not go into details and instead explain notations by example, when we use them later on.

MMT terms are essentially the OpenMath objects [Bus+04] formed from the constants included into the theory under consideration. Here, it suffices

to consider the fragment of MMT terms formed from constants c, variables x, numbers literals, applications $\texttt{OMA}(f, t_1, \ldots, t_n)$ of f to the t_i, and bindings $\texttt{OMBIND}(b; x_1, \ldots, x_n; t)$ where a binder b binds the variables x_i in the scope t.

2.2 Content Dictionaries as MMT Theories

OPENMATH declares symbols in named content dictionaries that have global scope (unlike MMT theories, which they otherwise closely resemble; but see Section 2.2). Consequently, references to symbols must reference the CD and the symbol name within that CD. The official OPENMATH CDs [OMCD] are a collection of content dictionaries for basic mathematics. For example, the content dictionary arith1 declares among others the symbols plus, minus, times, and divide for arithmetic in any mathematical structure – e.g., a commutative group or a field – that supports it.

Each symbol has a type using the STS type system [Dav00]. The types describe what kinds of application (rarely: binding) objects can be formed using the symbol. For example, its type licenses the application of plus to any sequence of arguments, which should come from a commutative semigroup. Moreover, each symbol comes with a textual description of the meaning of the thus-constructed application, and sometimes axioms about it, e.g., commutativity in the case of plus.

We represent every OPENMATH CD as an MMT theory, whose meta-theory is a special MMT theory OpenMath. Moreover, every OPENMATH symbol is represented as an MMT constant. All constants are definition-less, and it remains to describe their types and notations. Mathematical properties that are given as formulas are also represented as MMT constants using a special type.

OPENMATH	MMT
CD	theory
symbol	constant
property F	constant $\texttt{OMA}(\mathsf{FMP}, F)$

Meta-Theory and Type System OpenMath must declare all those symbols that are used to form the types of OPENMATH symbols. This amounts to a formalization of the STS type system [Dav00] employed in the OPENMATH CDs. However, because the details STS are not obvious and not fully specified, we identify the strongest type system that we know how to formalize and of which STS is a weakening. Here strong/weak means that the typing relation holds rarely/often, i.e., every STS typing relation also holds in our weakened version. The types in this system are: *i)* Object *ii)* $\texttt{OMA}(\mathsf{mapsto}, \mathsf{Object}, \ldots, \mathsf{Object}, A, \mathsf{Object})$ where A is either Object or naryObject *iii)* binder. Here binder is the type of symbols that take a context C and an Object in context C and return an Object. This type system ends up being relatively simple and is essentially an arity-system.[1]

[1] In fact, we are skeptical whether any fully formal type system for all of OPENMATH can be more than an arity system.

Moreover, we add a special symbol FMP to represent mathematical properties as follows: A property asserting F is represented as a constant with definiens $\texttt{OMA}(\mathsf{FMP}, F)$.[2] Intuitively, we can think of FMP as a partial function that can only be applied to true formulas.

This results in the following MMT theory OpenMath in Figure 2. There, the notation of mapsto means that it takes first a sequence or arguments with separator $\times$ followed by the separator $\to$ and one more argument.

```
theory OpenMath
  constant mapsto # 1×... → 2
  constant Object
  constant naryObject
  constant binder
  constant FMP
```

Fig. 2. MMT Theory OpenMath

```
theory arith1 : OpenMath
  plus : naryObject → Object
      # 1+...
  minus : Object × Object → Object
      # 1 − 2
  plus : naryObject → Object
      # 1*...
  ...
```

```
theory NumbersTest : OpenMath
  include arith1
  include fns1
  include set1
  include relations1
  maptest = FMP
    {0,1,2} map (x ↦ −x*x+2*x+3) = {3,4}
```

Fig. 3. OpenMath CDs in MMT

Notations In order to write OpenMath objects conveniently – in particular, to write the examples mentioned below – we add notations to all OpenMath symbols. OpenMath does not explicitly specify notations for the symbols in the official CDs. However, we can gather many implied notations from the stylesheets provide to generate presentation MathML. Most of these can be mapped to MMT notations in a straightforward fashion. As MMT notations are one-dimensional, we make reasonable adjustments to two-dimensional MathML notations such as those for matrices and fractions.

Example 1. The left listing in Fig. 3 gives a fragment of the MMT theory representing the CD arith1. Here the notation of plus means that it takes a sequence or arguments with separator $+$, and the one of minus that it takes two arguments separated by $-$.

The right listing uses the module system to import some CDs and then give an example of a true computation as an axiom. It uses the symbols set1?set, fns1?lambda, and relation1?eq and the notations we declare for them.

2.3 Scala Classes as MMT Theories

Scala [OSV07] combines features of object-oriented and functional programming languages. At the module and statement level, it follows the object-oriented

[2] We do not use a propositions-as-types representation here because it would make it harder to translate OpenMath to other languages.

paradigm and is similar to Java. At the expression level, it supplements Java-style imperative features with simple function types and inductive types.

A *class* is given by its list of member declarations, and we will only make use of 3 kinds of members: *types*, immutable typed *values*, and *methods*, which are essentially values of functional type.

Values have an optional definiens, and a class is *concrete* if all members have one, otherwise abstract. Scala introduces special concepts that can be used instead of classes without constructor arguments: *trait* in the abstract and *object* in the concrete case. Traits permit *multiple inheritance*, i.e., every class can inherit from multiple traits. Objects are singleton classes, i.e., they are at the same time a class and the only instance of this class. An object and a trait may have the same name, in which case their members correspond to the static and the non-static members, respectively, of a single Java class.

The representation of Scala classes proceeds very similarly to that of OpenMath CDs above (see Figure 4). In particular, we use a special meta-theory Scala that declares the primitive concepts needed for our Scala expressions. Then we represent Scala classes as Mmt theories and members as constants. While OpenMath CDs always have the flavor of specifications, Scala classes can have the flavor of specifications (abstract classes/traits) or implementations (concrete classes/objects).

Scala	Mmt
trait T	theory $\overline{T}$
type member	constant of type type
value member	constant
method member	constant of functional type
object O of type T	theory morphism $\overline{T} \to$ Scala
members of O	assignment to the corresponding $\overline{T}$-constant
extension between classes	inclusion between theories

Fig. 4. Scala Classes as Mmt Theories

Meta-Theory and Type System Our meta-theory Scala could declare symbols for every primitive concept used in Scala expressions. However, most of the complexity of Scala expressions stems from the richness of the term language. While the representation of terms would be very useful for verification systems, it does not contribute much to our goals of computation and biform development. Therefore, we focus on the simpler type language. Moreover, we omit many theoretically important technicalities (e.g., singleton and existential types) that have little practical bearing. Indeed, many practically relevant types (e.g., function and collection types) are derived notions defined in the Scala library.

Therefore, we represent only the relevant fragment of Scala in Scala. Adding further features later is easy using the Mmt module system. For all inessential

(sub-)expressions, we simply make use MMT escaping: MMT expressions can seamlessly escape into arbitrary non-MMT formats.

Thus, we use the MMT theory Scala in Figure 5, which gives mainly the important type operators and their introductory forms. Where applicable, we use MMT notations that mimic Scala's concrete syntax. This has the added benefit that the resulting theory it is hardly Scala-specific and thus can be reused easily for other programming languages. It would be straightforward to add typing rules to this theory by using a logical framework as the meta-theory of Scala, but this is not essential here.

```
theory Scala
  constant type
  constant Any
  constant Function # (1,...)=> 2
  constant Lambda # (1,...)=> 2
  constant List # List[1]
  constant list # List(1,...)
  constant BigInt
  constant Double
  constant Boolean
  constant String
```

Fig. 5. The MMT Theory Scala

Representing Classes It is now straightforward to represent a Scala trait T containing only 1. type members 2. value members whose types only use symbols from Scala 3. method members whose argument and return types only use symbols from Scala as an MMT theory $\overline{T}$ with meta-theory Scala.

1. type n yields constant n: type
2. val n: A yields constant n: $\overline{A}$
3. def n(x1:A_1,..,x_r:A_r):A yields constant n: ($\overline{A_1}$,...,$\overline{A_r}$)=>$\overline{A}$

Here $\overline{A}$ is the structural translation of the Scala type A into an MMT expression, which replaces every Scala primitive with the corresponding symbol in Scala.

Similarly, we represent every object O defining (exactly) the members of T as an MMT view $\overline{O} : \overline{T} \to$ Scala. The member definitions in O give rise to assignments in $\overline{O}$ as follows:

1. type n $= t$ yields constant n $= \overline{t}$
2. val n: $A = a$ yields constant n = "a"
3. def n(x_1:A_1,...,x_r:A_r):$A = a$ yields constant n = (x_1:$\overline{A_1}$,...,x_r:$\overline{A_r}$):$\overline{A}$ = "a"

Here "E" represents the escaped representation of the literal Scala expression E. Note that we do not escape the λ-abstraction in the implementation of comp. The resulting partially escaped MMT term is partially parsed and type-checked by MMT. This has the advantage that the back-translation from MMT to Scala can reuse the same variable names that the Scala programmer had chosen.

Example 2. A Scala class for monoids and an implementation in terms of the integers are given as the top two code fragments in Figure 6, their MMT representations in the lower two.

Representing the Module Systems The correspondence between MMT theory inclusions and Scala class extensions is not exact due to what we call the import name clash in [RK13]: It arises when modules M_1 and M_2 both declare a symbol c and M imports both M_1 and M_2. OPENMATH and MMT use qualified names for scoped declarations (e.g., $M_1?c$ and $M_2?c$) so that the duplicate use of c is

```
trait Monoid {
  type U
  val unit: U
  def comp(x1: U, x2: U): U
}
```

```
object Integers extends Monoid {
  type U = BigInt
  val unit = 0
  def comp(x1: U, x2: U) = x1 + x2
}
```

```
theory Monoid : Scala
  constant U : type
  constant unit : U
  constant comp : (U,U) => U
```

```
view Integers : Monoid -> Scala
  constant U = BigInt
  constant unit = "0"
  constant comp = (x1:U, x2:U) => "x1 + x2"
```

Fig. 6. Scala and MMT representations of Monoids and Integers

inconsequential. But Scala – typical for programming languages – identifies the two constants if they have the same type.

There are a few ways to work around this problem, and the least awkward of them is to qualify all field names when exporting MMT theories to Scala. Therefore, the first declaration in the trait Monoid is actually type Monoid_U and similar for all other declarations. Vice versa, when importing Scala classes, we assume that all names are qualified in this way.

It remains future work to align larger fragments of the module systems, which would also include named imports and sharing.

3 Biform Theory Development in MMT

We can now combine the representations of OpenMath and Scala in MMT into a biform theory graph. In fact, we will obtain this combination as an example of a general principle of combining a logic and a programming language.

Bifoundations Consider a logic represented as an MMT theory L and a programming language represented (possibly partially as in our case with Scala) as an MMT theory P. Moreover, consider an MMT theory morphism $s : L \to P$. Intuitively, s describes the meaning of L-specifications in terms of P.

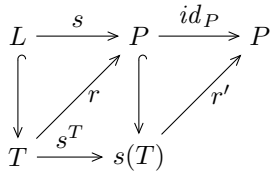
Definition 1. A *bifoundation* is a triple $(L, P, s : L \to P)$.

Now consider a logical theory T represented as an MMT theory with meta-theory L. This yields the diagram in the category of MMT theories, which is given on the right. Then, inspired by [Rab12], we introduce the following definition of what it means to implement T in P:

$$\begin{array}{ccc} L & \xrightarrow{s} & P \\ \downarrow & & \\ T & & \end{array}$$

Definition 2. A *realization* of T over a bifoundation (L, P, s) is a morphism $r : T \to P$ such that the resulting triangle commutes.

In MMT, the pushout of inclusions always exists. Moreover, we can make coherent canonical choices for these pushouts that are also straightforward to compute. Therefore, s induces a pushout functor from L-theories to P-theories, and we write $s(T)$ for the pushout of T along s. Every constant declaration in $s(T)$ (except those included from P) arises by translating a constant declaration in T (except those included from L), and s^T maps these T-constants to the corresponding $s(T)$-constant of the same name.



This situation is described in the commutative diagram above. Now, using the universal property of the pushout, we see that realizations r are in a canonical bijection with morphisms $r' : s(T) \to P$ (which are completely independent of L). Moreover, in reasonable practical situations, all morphisms r' are in the image of the representation function that represents P-programs as MMT morphisms. Therefore, we can identify the P-programs implementing T with the MMT morphisms $s(T) \to P$.

A Bifoundation for OPENMATH *CDs and Scala* We obtain a bifoundation by giving an MMT morphism s : OpenMath $\to$ Scala. This morphism hinges upon the choice for the Scala type that interprets the universal type Object. There are two canonical choices for this type, and the resulting morphisms are given in Figure 7. Firstly, we can choose the universal Scala type Any. This leads to a semantic bifoundation where we interpret every OPENMATH object by its Scala counterpart, i.e., integers as integers, lists as lists, etc. Secondly, we can choose a syntactic bifoundation where every object is interpreted as itself. This requires using a conservative extension ScalaOM of Scala that defines inductive types Term of OPENMATH objects and Context of OPENMATH contexts. Such an extension is readily available because it is part of the MMT API.

```
view Semantic: OpenMath -> ScalaOM
  constant Object = Any
  constant mapsto = Function
  naryObject = List[Any]
  binder = (Context,Term) => Any
  FMP = (x:Any) => "assert(x == true)"
```

```
view Syntactic: OpenMath -> ScalaOM
  constant Object = Term
  constant mapsto = Function
  naryObject = List[Term]
  binder = (Context,Term) => Term
  FMP = (x:Term) =>
          "assert(x == OMS(logic1.true))"
```

Fig. 7. Two Bifoundations From Scala to OPENMATH

In both cases, n-ary arguments are easily interpreted in terms of lists and functions as functions. The case for binders is subtle: In both cases, we must interpret binders as Scala functions that take a syntactic object in context. Therefore, even the semantic foundation requires ScalaOM as the codomain.

Finally, we map mathematical properties to certain Scala function calls, e.g., assertions. In the semantic case, we assert the formula to be true. In the syntactic case, we assert it to be equal to the symbol true from the OpenMath CD logic1. Here, OMS is part of the Mmt API.

Of course, in practice, only the simplest of FMPs actually hold in the sense that a simple Scala computation could prove them. However, our interpretation of FMPs is still valuable: It naturally translates examples given in the OpenMath CDs to Scala test cases that can be run systematically and automatically. Moreover, in the syntactic case, we have the additional option to collect the asserted formulas and to maintain them as input for verification tools.

4 Mechanizing Biform Theory Graphs

We are particularly interested in the syntactic bifoundation given above. It corresponds to the well-understood notion of a syntactic model of a logic. Thus, it has the advantage of completeness in the sense that the algorithms given in T-realizations can be used to describe deductive statements about T. In this section, we make this more precise and generalize it to arbitrary logics.

Abstract Rewrite Rules First we introduce an abstract definition of rule that serves as the interface between the computational and the deductive realm. We need one auxiliary definition:

Definition 3. An **arity** is an element of $\{n, n* : n \in N\} \cup \{\mathsf{binder}\}$.

This definition of arity is a slight simplification of the one we give in [KR12] and use in Mmt, which permits sequences anywhere in the argument list and gives binders different arities as well.

Now let us fix an Mmt theory graph and write $\mathcal{C}$ for the set of constants declared in these theories. We write $\mathcal{T}$ for the set of closed Mmt terms using only constants from $\mathcal{C}$, and $\mathcal{T}(x_1, \ldots, x_n)$ for the set of terms that may additionally use the variables $x_1, \ldots, x_n$. Then we define:

Definition 4. A **rule** r for a constant c with arity $n \in \mathbb{N}$ is a mapping $\mathcal{T}^n \to \mathcal{T}$. Such a rule is **applicable** to any $t \in \mathcal{T}$ of the form $\texttt{OMA}(c, t_1, \ldots, t_n)$. In that case, its intended meaning is the formula $t = r(t_1, \ldots, t_n)$.

A **rule** for a constant c with arity $n*$ is a mapping $\mathcal{T}^n \times (\bigcup_{i=0}^{\infty} \mathcal{T}^i) \to \mathcal{T}$. Such a rule is **applicable** to any $t \in \mathcal{T}$ of the form $\texttt{OMA}(c, t_1, \ldots, t_k)$ for $k \geq n$. In that case, its intended meaning is the formula $t = r(t_1, \ldots, t_k)$.

A **rule** for a constant c with arity binder is a mapping $\{(G, t) | G = x_1, \ldots, x_n \wedge t \in \mathcal{T}(G)\} \to \mathcal{T}$. Such a rule is **applicable** to any $t \in \mathcal{T}$ of the form $\texttt{OMBIND}(c; G; t')$. In that case, its intended meaning is the formula $t = r(G, t')$.

A **rule base** R is a set of rules for some constants in $\mathcal{C}$. We write $R(c, a)$ for the set of rules in R for the constant c with arity a.

Our rules are different from typical rewrite rules [TN99] of the form $t_1 \rightsquigarrow t_2$ in two ways. Firstly, the left hand side is more limited: A rule for c is applicable exactly to the terms t_1 whose head is c. This corresponds to the intuition of a rule implementing the constant c. It also makes it easy to find the applicable rules within a large rule base. Secondly, the right hand side is not limited at all: Instead of a term t_2, we use an arbitrary function that returns t_2. This corresponds to our open-world assumption: Constants are implemented by arbitrary programs (written in any programming language) provided by arbitrary sources.

In the special case without binding, our rules are essentially the same as those used in [Far07], where the word *transformer* is used for the function $r(-)$.

It is now routine to obtain a rewrite system from a rule base:

Definition 5. Given a rule base R, R-rewriting is the reflexive-transitive closure of the relation $\rightsquigarrow \subseteq \mathcal{T} \times \mathcal{T}$ given by:

$$\frac{r \in R(c,0)}{c \rightsquigarrow r()} \qquad \frac{t_i \rightsquigarrow t_i' \text{ for } i=0,\ldots,n}{\texttt{OMA}(t_0,\ldots,t_n) \rightsquigarrow \texttt{OMA}(t_0',\ldots,t_n')} \qquad \frac{r \in R(c,n) \text{ or } r \in R(c,i*) \text{ for } i \leq n}{\texttt{OMA}(c,t_1,\ldots,t_n) \rightsquigarrow r(t_1,\ldots,t_n)}$$

$$\frac{t_i \rightsquigarrow t_i' \text{ for } i=1,2}{\texttt{OMBIND}(t_1;G;t_2) \rightsquigarrow \texttt{OMBIND}(t_1;G;t_2)} \qquad \frac{r \in R(c,\mathsf{binder})}{\texttt{OMBIND}(c;G;t) \rightsquigarrow r(G,t)}$$

R-rewriting is not guaranteed to be confluent or terminating. This is unavoidable due to our abstract definition of rules where not only the set of constants and rules are unrestricted but even the choice of programming language. However, this is usually no problem in practice if each rule has evaluative flavor, i.e., if it transforms a more complex term into a simpler one.

Realizations as Rewriting Rules Consider a realization r of T over the bifoundation $(\mathsf{OpenMath}, \mathsf{ScalaOM}, \mathsf{Syntactic})$, and let ρ be the corresponding Scala object. Then for every constant c with type $\texttt{OMA}(\mathsf{mapsto}, \mathsf{Object}, \ldots, \mathsf{Object})$ declared in T, we obtain a rule r_c by putting $r_c(t_1, \ldots, t_n)$ to be the result of evaluating the Scala expression $\rho.c(t_1, \ldots, t_n)$[3]. We obtain rules for constants with other types accordingly. More generally, we define:

Definition 6. Given a theory T, an **arity assignment** maps every T-constant to an arity.

Given an arity assignment, a realization $T \to \mathsf{ScalaOM}$ is called **syntactic** if the type of every T-constant with arity a is mapped to the following Scala type: `(Term,...,Term) => Term` if $a = n$; `(Term,...,Term, List[Term]) => Term` if $a = n*$; and `(Context,Term) => Term` if $a = \mathsf{binder}$.

Definition 7. *Given a syntactic realization $r : T \to \mathsf{ScalaOM}$, we write* $\mathrm{Rules}(r)$ *for the rule base containing for each constant c declared in T the rule induced by the Scala function $r(c)$.*

[3] Technically, in practice, we need to catch exceptions and set a time-out to make r_c a total function, but that is straightforward.

A general way of obtaining arity assignments for all theories T with a fixed meta-theory L is to give an MMT morphism $e : L \to \mathsf{OpenMath}$. e can be understood as a *type-erasure translation* that forgets all type information and merges all types into one universal type. Then the arities of the T-constants are determined by the OPENMATH types in the pushout $e(T)$. Therefore, we can often give bifoundations for which all realizations are guaranteed to be syntactic, the bifoundation $(\mathsf{OpenMath}, \mathsf{Scala}, \mathsf{Syntactic})$ being the trivial example.

Def. 7 applies only to realizations in terms of Scala. However, it is straightforward to extend it to arbitrary programming languages. Of course, MMT – being written in Scala – can directly execute Scala-based realizations whereas for any other codomain it needs a plugin that supplies an interpreter.

The Universal Machine We use the name *universal machine* for the new MMT component that maintains a rule base containing the rules induced by the syntactic realizations in MMT's knowledge base. Here "universal" refers to the open-world perspective that permits the extension with new logics and theories as well as programming languages and implementations. The universal machine implements the rewrite system from Def. 5 and exposes it via a web server, via the MMT API, and via an interactive interpreter.

5 Building a Biform Library in MMT

We evaluate the new MMT concepts by building a biform MMT theory graph based on the bifoundation $(\mathsf{OpenMath}, \mathsf{Scala}, \mathsf{Syntactic})$, which provides Scala implementations for the official OPENMATH CDs. This development is available as an MMT project and described in more detail at `https://tntbase.mathweb.org/repos/oaff/openmath`.

MMT projects [Hor+11] already support different dimensions of knowledge, such as source, content, and presentation, as well as build processes that transform developments between dimensions. We add one new dimension for generated programs and workflows for generating it.

Firstly, we write MMT theories representing the OPENMATH CDs such as the one given on the left of Fig. 3. Specifically, we represent the arith, complex, fns, integer, interval, linalg, list, logic, minmax, nums, relation, rounding, set, setname, and units CDs along with appropriate notations.

Secondly, we write views from these CDs to ScalaOM. Then a new MMT build process generates all corresponding Scala classes. Typically, users write view stubs in MMT and then fill out the generated Scala stubs using an IDE of their choice. Afterwards MMT imports the Scala stubs and merges the user's changes into the MMT views. This is exemplified in Fig. 8. Here the left side gives a fragment of an MMT view out of arith1, which implements arithmetic on numbers. (We also give other views out of arith1, e.g., for operations on matrices.) The implementation for plus is still missing whereas the one for minus is present. The right side shows the generated Scala code with the editable parts marked by comments.

```
view NumberArith :
  arith1 -> ScalaOM =
    plus = (args: List[Term]) "
    "

    minus = (a: Term, b: Term) "
      (a,b) match {
        case (OMI(x), OMI(y)) =>
          OMI(x − y)
      }
    "
```

```
object NumberArith extends arith1 {
  def arith1_plus(args: List[Term]) : Term = {
    // start NumberArith?plus
    // end NumberArith?plus
  }
  def arith1_minus(a: Term, b: Term) : Term = {
    // start NumberArith?minus
    (a,b) match {
        case (OMI(x), OMI(y)) => OMI(x − y)
      }
    // end NumberArith?minus
  }
}
```

Fig. 8. Partial Realization in Mmt and Generated Scala Code

Finally, we write Mmt theories for extensions of the OpenMath CDs with examples as on the right in Fig. 3. We also give realizations for them, which import the realizations of the extended CDs. Here Mmt generates assertions for each FMP.

In fact, we generate a little more code than the examples show. Most importantly, we generate methods for iterating over all generated Scala objects, and all of those contain code for iterating over all their methods. Thus, a single Scala function call is sufficient to register all rules induced by all realizations with the universal machine or to run all test cases and generate a report.

6 Conclusion

We described a formal framework and a practical infrastructure for biform theory development, i.e., the integration of deductive theories and computational definitions of the functions specified in them. The integration is generic and permits arbitrary logics and programming languages; moreover, the same module system is used for specifications and implementations. Our work permits linking function symbols with unverified implementations as well as "soft verification" where axioms generate comments and test cases. It also provides a convenient starting point for verification tools.

We have instantiated our design with a biform development of the OpenMath content dictionaries in Scala. Future work will focus on the development of larger biform libraries and the use of further logics and programming languages. In particular, we want to explore how to treat richer type systems and to preserve their information in the generated Scala code.

Acknowledgements The work reported here was prompted by discussions with William Farmer and Jacques Carette in the TetraPod project. An initial version of the universal machine was developed in collaboration with Vladimir Zamdzhiev.

References

[Ahr+05] W. Ahrendt et al. "The KeY Tool". In: *Software and System Modeling* 4 (2005), pp. 32–54.

[Bus+04] S. Buswell et al. *The Open Math Standard, Version 2.0*. Tech. rep. See http://www.openmath.org/standard/om20. The Open Math Society, 2004.

[Dav00] James Davenport. "A small OpenMath type system". In: *Bulletin of the ACM Special Interest Group on Symbolic and Automated Mathematics (SIGSAM)* 34.2 (2000), pp. 16–21.

[DM05] D. Delahaye and M. Mayero. "Dealing with Algebraic Expressions over a Field in Coq using Maple". In: *Journal of Symbolic Computation* 39.5 (2005), pp. 569–592.

[Far07] William M. Farmer. "Biform Theories in Chiron". In: *Towards Mechanized Mathematical Assistants. MKM/Calculemus*. Ed. by Manuel Kauers et al. LNAI 4573. Springer Verlag, 2007, pp. 66–79. ISBN: 978-3-540-73083-5. DOI: http://dx.doi.org/10.1007/978-3-540-73086-6_6.

[FGT92] W. Farmer, J. Guttman, and F. Thayer. "Little Theories". In: *Conference on Automated Deduction*. Ed. by D. Kapur. 1992, pp. 467–581.

[Hor+11] F. Horozal et al. "Combining Source, Content, Presentation, Narration, and Relational Representation". In: *Intelligent Computer Mathematics*. Ed. by J. Davenport et al. Springer, 2011, pp. 212–227.

[HT98] J. Harrison and L. Théry. "A Skeptic's Approach to Combining HOL and Maple". In: *Journal of Automated Reasoning* 21 (1998), pp. 279–294.

[KR12] M. Kohlhase and F. Rabe. "Semantics of OpenMath and MathML3". In: *Mathematics in Computer Science* 6.3 (2012), pp. 235–260.

[KST97] S. Kahrs, D. Sannella, and A. Tarlecki. "The definition of extended ML: A gentle introduction". In: *Theoretical Computer Science* 173.2 (1997), pp. 445–484.

[OMCD] OpenMath *Content Dictionaries*. URL: http://www.openmath.org/cd/ (visited on 11/13/2010).

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

[Rab12] F. Rabe. "A Logical Framework Combining Model and Proof Theory". In: *Mathematical Structures in Computer Science* (2012). to appear; see http://kwarc.info/frabe/Research/rabe_combining_10.pdf.

[RK13] F. Rabe and M. Kohlhase. "A Scalable Module System". In: *Information and Computation* (2013). conditionally accepted; see http://arxiv.org/abs/1105.0548.

[TN99] Franz Baader aund Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.