A Practical OpenMath Machine

Bachelor Thesis in Computer Science

Felix Gabriel Mance
supervised by Prof. Dr. Michael Kohlhase and Dr. Florian Rabe
Jacobs University Bremen
f.mance@jacobs-university.de

May 27, 2013

Abstract

A long-term goal in mechanized mathematics is to create a system that combines the declarative, deductive and computational aspects of Mathematical Knowledge Representation.

Our approach towards this is to first create two subsystems: one that combines the declarative and deductive aspects and one that combines the declarative and computational aspects. The two subsystems *share* MMT as the declarative component, which allows us to later on connect the deductive and computational components. Furthermore, MMT can specify *arbitrary* logics and programming languages, which makes our system fully extensible. This is a major advantage over other Proof Assistants and Computer Algebra systems, which fix the logic and the programming language, respectively.

In this thesis, we focus on the subsystem that integrates the declarative and computational components. We create a framework which uses MMT to declare the specifications and implementations of mathematical concepts and a universal machine as a computational engine that collects all the implementations and uses them to evaluate mathematical expressions. As a case study, we select a representative set of concepts from the OpenMath Content Dictionaries and we write implementations for them in the Scala programming language.

1 Introduction

There exist three main *aspects* of Mathematical Knowledge Representation. The **declarative** aspect is concerned with symbols, types, theories and relations between theories, such as theory morphisms and module systems. The **deductive** aspect consists of axioms, rules of inference, theorems and proofs and the **computational** aspect deals with numerical and symbolic computation.

Existing systems integrate at most two of these aspects and they fall into three categories:

- the **declarative systems** focus on the declarative aspect and are minimally concerned with deduction or computation. Some declarative, logic independent systems are OpenMath [BCC+04], OMDoc [Koh06] and Mmt [Rab13]. They are extensible, which means that users can add new symbols or theories to them.

- the **declarative and deductive systems** combine the declarative and deductive aspects and are commonly known as **proof assistants (PA)**. They must provide a formal language or logic in which to write the axioms and the rules of inference. They are only partially extensible, because the logic is usually fixed (in order to optimize the derivations of the proofs), but users can add new definitions and theorems. Examples of PAs are the Mizar system[TB85] and Coq[BC04]. There also exist proof assistants which declare multiple logics and are called **logical frameworks**. Examples are LF[HHP93] and Isabelle[Wen09]. LF does not optimize derivations in any of its logics, while Isabelle optimizes in only few of them.

- the **declarative and computational systems** integrate the declarative and computational aspects and are commonly known as **Computer Algebra** Systems **(CA)**. They usually provide a programming language in which to write the implementations of mathematical operations. Similar to the PAs, they are only partially extensible since the programming language is fixed, but users can add new operations and implementations. Examples of CAs are Mathematica[Mat], Maple[Map] and Axiom[Axi].

The PAs and the CAs have different purposes and areas of application. PAs are generally used in formalization of mathematics and software verification, while CAs are used in engineering, mathematics, physics or statistics. A CA typically performs numerical or symbolic computation, where the user enters an expression and the CA (usually) returns a simplified version of it. CAs are optimized for high-level mathematics and are very efficient in carrying out computations. In contrast to PAs, CAs are neither fully precise nor reliable, and are not able to express complex type systems and logics[HT98]. PAs are used for defining and proving; users can declare mathematical theories, define properties and do logical reasoning on them[Geu09]. In contrast to CAs, PAs are weaker at numerical or symbolic computation, but guarantee their correctness[HT98].

We can see that the PAs and the CAs complement each other's strengths and, therefore, combining them has been a major research topic.

Goal An outstanding goal of mechanized mathematics is to design a system that is able to compute efficiently and precisely, do proofs and verify theorems. Such a system would be a **declarative, deductive and computational (DDC)** system.

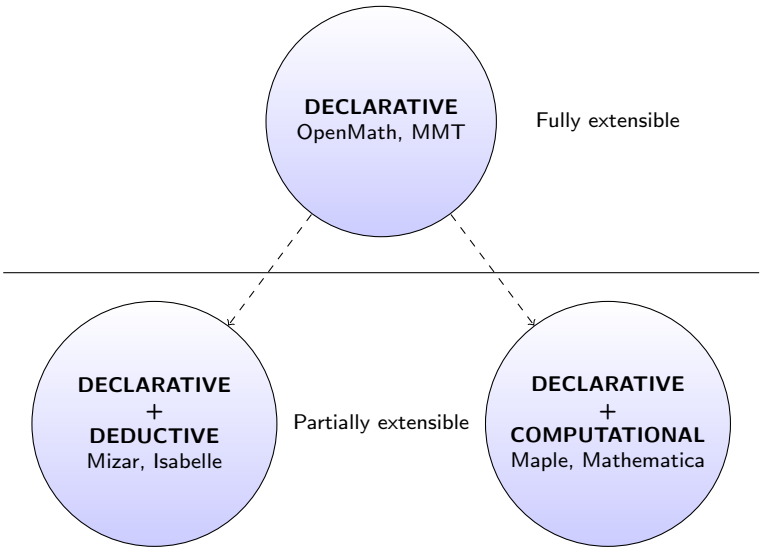


Figure 1: Some of the currently available systems.

The most difficult part in creating a DDC system, however, is the integration of deduction and computation. There exists a large body of research that aims to do that, using different approaches. One of them is the *combinational* approach, where a PA and a CA are combined or connected to obtain a DDC system. For example, in [DM05], Maple computations are imported into Coq and in [HT98], HOL and Maple are connected using a *software bus*. An obvious drawback of the combinational approach is the fact that the resulting system inherits all the weaknesses of its PA and CA, including their limited extensibility. Another approach towards a DDC system is to design a specification language powerful enough to support specifications of both a logic (for the deductive part) and a programming language (for the computational part). A system that employs this is FoCaLiZe[foc], where the environment is based on a functional language with object-oriented features which allows the user to write formal specifications and proofs. This system also fixes the programming language (OCaml) and the PA (Coq).

Our Approach We contribute to the development of a DDC system by initially creating two subsystems. The first subsystem integrates the declarative and deductive realms and the second subsystem integrates the declarative and computational realms. The **key idea** here is to have the *same declarative component* for both subsystems, so that the two can be connected at a later stage and become a complete DDC system. Moreover, the declarative system that we use, called Mmt, allows us to specify *arbitrary* logics and programming languages, which makes our DDC system **fully extensible**. This is a major advantage over other PAs and CAs which, as previously mentioned, are only

partially extensible, since they fix the logic and the programming language.

In this thesis, we present the practical realization of one of the subsystems, namely the one which integrates the declarative and computational realms. We call this system Ddc-Lite.

Let us now briefly discuss how Ddc-Lite works. We use a language L to represent the data that we are computing on (commonly known as *specification language*) and a programming language P to implement the mathematical operations. We implement L and P as Mmt *meta-theories* $\mathsf{L}_{\mathsf{MMT}}$ and $\mathsf{P}_{\mathsf{MMT}}$, respectively. These meta-theories introduce the primitive concepts of L and P into Mmt. L and P primitives are declared as Mmt *constants* in $\mathsf{L}_{\mathsf{MMT}}$ and $\mathsf{P}_{\mathsf{MMT}}$, respectively.

We *specify* a concept c as an Mmt constant $\mathsf{c}_{\mathsf{MMT}}$ in a theory $\mathsf{t}_{\mathsf{MMT}}$ whose meta-theory is $\mathsf{L}_{\mathsf{MMT}}$. We *implement* c as an Mmt assignment in a *morphism* $\mathsf{T}_{\mathsf{MMT}} : \mathsf{t}_{\mathsf{MMT}} \to \mathsf{P}_{\mathsf{MMT}}$, where we assign to $\mathsf{c}_{\mathsf{MMT}}$ a code snippet $\mathsf{c}_{\mathsf{impl}}$ written in P. Intuitively, the code snippet acts as the implementation of the computational semantics of $\mathsf{c}_{\mathsf{MMT}}$[IMR13]. A component of Ddc-Lite, called the Universal OpenMath Machine, collects all the implementations and uses them to evaluate L-expressions.

The design and implementation of Ddc-Lite are explained in more detail in Sections 3 and 4 respectively, where, as a case study, we use OpenMath as the specification language and Scala as the programming language.

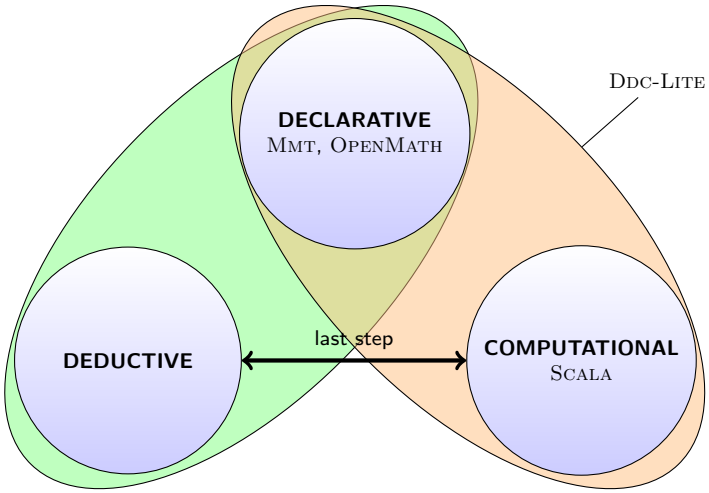


Figure 2: Our approach towards building a DDC system

Acknowledgements. Part of the material of this thesis has been developed in two papers, [KMR13] and [IMR13], jointly written by me, my supervisors and Mihnea Iancu. With this opportunity, I would like to thank them for their valuable help and support. I would also like to thank Vladimir Zamdzhiev,

who developed an initial version of the Universal OpenMath Machine as part his bachelor thesis[Zam11].

2 Preliminaries

2.1 MMT and OMDoc

MMT [Rab13] is a modular and logic independent representation language for mathematical knowledge. MMT *theories* may contain symbol declarations for constants, functions, predicates, inference rules, axioms and theorems.

Each MMT constant can have a *notation*, which makes it easier for users to write expressions using that constant. For example, constant minus has notation $1-2$, which means that it takes two arguments, separated by " $-$ ". As a result, we can write $x - y$ instead of $\mathsf{minus}(x, y)$. The arity of a constant can be determined from its notation (here, minus has arity 2).

MMT theories can have a *meta-theory* which declares the primitive notions of the language in which the theory is written. For example, the meta-theory of a program is the specification of the programming language.

MMT theories are related via theory morphisms, called MMT *views*. A view $\mathsf{A} : \mathsf{a} \to \mathsf{P}$ must declare a definition $\mathsf{c}_{\mathsf{impl}}$ for every undefined constant $\mathsf{c} \in \mathsf{a}$, specified in terms of language P. The intuition is that $\mathsf{c}_{\mathsf{impl}}$ implements c in P.

MMT *terms* are OPENMATH objects formed from the constants declared in the currently active theory[KMR13].

The MMT-API is the implementation of MMT and MMT-based knowledge management systems, and is written in SCALA [Rab13].

OMDOC [Koh06] is a semantic markup language able to represent mathematical expressions, theories (corresponding to OPENMATH CDs), definitions, proofs, natural language and code. Moreover, OMDOC is logic-independent, that is, any logic can be specified in it. MMT is the formal core of OMDOC in the sense that MMT gives a formal specification to the fragment of OMDOC concerned with theory development [RK13].

2.2 OpenMath

OPENMATH [BCC+04] is a markup language which precisely describes the semantics of mathematical expressions. Written in a standardized representation (XML or binary encoding), OPENMATH objects can be exchanged between computer programs, stored in databases, displayed in a browser, verified as being sound or published online. An OPENMATH *expression* is constructed from an abstract data type called OPENMATH object and the *concepts* inside the expression, called *symbols*, are introduced by **Content Dictionaries (CD)**.

OPENMATH objects have several types, but for our DDC-LITE system, we only need the following:

- $\mathsf{OMI}(n)$ – integer n;
- $\mathsf{OMF}(f)$ – floating point number f;
- $\mathsf{OMS}(c)$ – constant c;
- $\mathsf{OMSTR}(s)$ – string s;
- $\mathsf{OMB}(b)$ – byte array b;
- $\mathsf{OMA}(c, t_1, \ldots, t_n)$ – application of c (called the *head*) on arguments $t_1, \ldots, t_n$;
- $\mathsf{OMBIND}(c, V, t)$ – bind variables in list V (called the *context*) by c in t (e.g. $\mathsf{OMBIND}(\lambda, \mathsf{OMV}(x), \mathsf{times}(x, \mathsf{OMS}(y)))$ – x is bound by λ in $x \cdot y$);
- $\mathsf{OMV}(x)$ – reference to a variable introduced by OMBIND;

For example, $x \cdot 2$ corresponds to $\mathsf{OMA}(\mathsf{times}, \mathsf{OMV}(x), \mathsf{OMI}(2))$ (in prefix-notation), and is encoded in XML as follows:

```
<OMOBJ>
  <OMA>
    <OMS cd="arith1" name="times"/>
    <OMV name="x"/>
    <OMI>2</OMI>
  </OMA>
</OMOBJ>
```

Figure 3: Encoding of $a \times 2$ in OpenMath-XML, where arith1 refers to the OpenMath CD introducing times.

2.3 Scala

Scala [Sca] is a programming language built on top of the *Java Virtual Machine* and is compatible with the Java programming language. Besides object-oriented programming, Scala supports functional programming, which makes it convenient to write implementations of mathematical functions. Further reasons for our choice of Scala are discussed in Section 3. We do not give a formal specification of Scala in this thesis, but we introduce it by examples, shown in subsection 3.3.

3 The Design of DDC–Lite

3.1 Prerequisites

Abstractly speaking, there are three essential components in a declarative-computational system:

- **data** – objects we compute on;
- **algorithms** – instructions which implement certain operations;
- **engine** – runs algorithms on data.

For our Ddc-Lite system, we instantiate:

- the **data** with a representative set of the symbols declared in the OpenMath CDs;
- the **algorithms** with Scala code snippets which implement the OpenMath symbols;
- the **engine** with a module called the Universal OpenMath Machine, which evaluates OpenMath expressions.

We chose OpenMath for several reasons. First, the OpenMath CDs contain a very rich set of mathematical concepts (arithmetical operations, list and set operations, elements of linear algebra, polynomials, transcendental functions, relations, logical operators, units, dimensions etc). This makes OpenMath-based computation useful in practice. Second, OpenMath is standardized and people already use it extensively to mark up mathematical expressions within documents. We could run the Universal OpenMath Machine on these documents and evaluate the OpenMath expressions inside.

We chose Scala as the programming language in which we write the implementations of the OpenMath objects. We could have chosen any other programming language, but since both the Mmt API and the Universal OpenMath Machine (which is a plugin of the API) are written in it, Scala's choice is natural. In addition, the object-oriented features of Scala allow straightforward translations from Mmt theories and views into Scala objects and traits (this process is explained in Subsection 3.3).

Now that we have settled upon the language of the data and the language of the algorithms, we still need a language that connects them: we use Mmt for this purpose. More precisely, we *declare* the OpenMath symbols and their Scala *implementations* in Mmt (see Subsection 3.2 for details). As mentioned in the introduction, one key benefit of using Mmt is that we will be able to connect a deductive system to Ddc-Lite, obtaining a fully declarative, deductive and computational system, which is the ultimate goal of our project.

We have now determined all components that are required for our Ddc-Lite system, so let us explain how the system works in theory. We describe in the next two subsections the **declarative** and the **computational** levels of Ddc-Lite.

3.2 The Declarative Level of DDC–Lite – Translation of OpenMath Content Dictionaries to MMT Theories

Arities Before we go on with our exposition, let us remark that OpenMath symbols can have one of the following arities:

- arity n, where $n \geq 0$ (e.g. set1.nil has arity 0, while arith1.minus has arity 2);
- arity $n*$, meaning that it takes $n \geq 0$ arguments and then a variable number of arguments (e.g. arith1.plus has arity $0*$);
- arity binder, corresponding to symbols that form objects of type OMBIND.

Meta-theories Meta-theory OpenMath declares the symbols that are used to construct the types of OpenMath objects and meta-theory Scala declares the symbols that are used to construct the types of Scala objects. The purpose of these two meta-theories is to introduce the primitive concepts of OpenMath and Scala into Mmt. Note that in the Scala theory, Term is the Scala type of Mmt terms and Context is the Scala type of Mmt contexts (a context is a list of variable declarations). Morphism Syntactic : OpenMath → Scala shows how to translate OpenMath theories into Scala programs.

```
theory OpenMath =
  Object
  FMP
  mapsto
  naryObject
  binder
```

```
theory Scala =
  Function # (1,...) => 2
  Lambda # (1,...) => 2
  List # List [1]
  Term
  Context
```

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

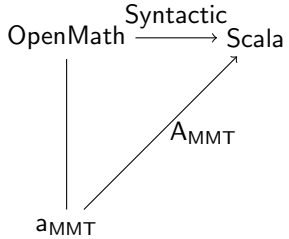
Figure 4: Meta-theories OpenMath, Scala and morphism Syntactic.

Translations to MMT Let a be an OpenMath CD. We translate a to an Mmt theory with the same name, $\mathsf{a_{MMT}}$, whose meta-theory is OpenMath. We translate each OpenMath symbol c declared in a to an Mmt constant with the same name, $\mathsf{c_{MMT}}$, in $\mathsf{a_{MMT}}$. We can also give a notation for $\mathsf{c_{MMT}}$, in order to conveniently write OpenMath expressions. Note that in $\mathsf{a_{MMT}}$ we do not write any implementation for $\mathsf{c_{MMT}}$. Furthermore, we translate mathematical properties given as formulas to a special Mmt constant, OMA(FMP, P), which asserts that P is true.

For each theory $\mathsf{a_{MMT}}$, we write one or more Mmt views $\mathsf{A_{MMT}} : \mathsf{a_{MMT}} \to$ Scala. To implement a constant $\mathsf{c_{MMT}} \in \mathsf{a_{MMT}}$, we assign it a λ-expression args $\Rightarrow$ code which maps the arguments of $\mathsf{c_{MMT}}$ to a Scala code snippet written between Mmt escape characters[IMR13]. The code snippet acts as the implementation of the computational semantics of c. The arguments args depend on the arity of $\mathsf{c_{MMT}}$ in the following way:

- for arity n, args $= (x_1 : \mathsf{Term}, \ldots, x_n : \mathsf{Term})$
- for arity $n*$, args $= (x_1 : \mathsf{Term}, \ldots, x_n : \mathsf{Term}, L : \mathsf{List[Term]})$
- for arity binder, args $= (V : \mathsf{Context}, t : \mathsf{Term})$

The following diagram illustrates the connections between the theory, the view and their meta-theories.



We are not yet using OpenMath to determine the types of OpenMath symbols, and we assume that they all have type Term. As

an immediate consequence, their arguments also have type Term. Since we are not using OpenMath, we do not need to use Syntactic. We instead use the notations in order to determine the arity of the symbols. This works only because OpenMath type system is actually an arity system. For more complex languages, we would have to use the meta-theories and the morphism.

Example Let us illustrate the steps above with an example and put a := complex1 CD. The notation CC 1 2 means that complex_cartesian takes two arguments, the real and imaginary parts, e.g. $2 + 3\mathrm{i} \rightsquigarrow$ CC 2 3. Figure 5 shows theory complex1, corresponding to the OpenMath CD with the same name. Figure 6 shows the view Complex1 which corresponds to theory complex1. The implementations are written between Mmt escape characters, shown here as quotes. Note that we only show the full implementation for object real (the other implementations are shown as " ... Scala code ...").

```
theory complex1 : OpenMath =
  complex_cartesian # CC 1 2
  complex_polar # CP 1 2
  argument # Arg 1
  real # Re 1
  imaginary # Im 1
  conjugate # Conj 1
```

Figure 5: Theory complex1

```
view Complex1 : complex1 -> ScalaOM =
  complex_cartesian = (re: Term, im: Term) " ... Scala code ..."
  complex_polar = (r: Term, phi: Term) " ... Scala code ..."
  real = (z: Term) "
    z match {
      case complex_cartesian(re, _) => re
      case complex_polar(r, phi) => arith1.times(r, transc1.cos(phi))
      case _ => real(z)
    }
  "
 imaginary = (z: Term) " ... Scala code ..."
 argument = (z: Term) " ... Scala code ..."
 conjugate = (z: Term) " ... Scala code ..."
```

Figure 6: View Complex1

3.3 The Computational Level of DDC–Lite – Translation of MMT Theories to Scala Objects

Let us review what we have so far: an Mmt theory $\mathsf{a}_{\mathsf{MMT}}$ which declares OpenMath objects as constants (with notations), and an Mmt view $\mathsf{A}_{\mathsf{MMT}}$, which

assigns a SCALA implementation to each constant declared in $\mathsf{a}_{\mathsf{MMT}}$.

In order to achieve computational power, the theory and the view are translated into SCALA objects by a component of DDC-LITE, called the Universal OpenMath Machine. This is done in the following way.

Translation of theory $\mathsf{a}_{\mathsf{MMT}}$ Theory $\mathsf{a}_{\mathsf{MMT}}$ is translated to the SCALA object $\mathsf{a}_{\mathsf{scala}}$ and each constant $\mathsf{c}_{\mathsf{MMT}}$ declared in $\mathsf{a}_{\mathsf{MMT}}$ is translated to the SCALA object $\mathsf{c}_{\mathsf{scala}}$ in $\mathsf{a}_{\mathsf{scala}}$. Object $\mathsf{c}_{\mathsf{scala}}$ defines apply/unapply methods and as a result, $\mathsf{c}_{\mathsf{scala}}$ acts as the constructor and pattern-matcher of c [IMR13]. For example, instead of writing e.g., $\mathsf{OMA}(\mathsf{OMS}(\text{"}\mathsf{plus}\text{"}), x_1, x_2)$, we can directly write $\mathsf{plus}(x_1, x_2)$ to construct an MMT Term in SCALA. More precisely:

- for arity n, $\mathsf{c}_{\mathsf{scala}}(x_1, \ldots, x_n) \overset{\mathsf{apply}}{\rightsquigarrow} \mathsf{OMA}(\mathsf{OMS}(\mathsf{c}), x_1, \ldots, x_n)$;
- for arity $n*$, $\mathsf{c}_{\mathsf{scala}}(x_1, \ldots, x_n, L) \overset{\mathsf{apply}}{\rightsquigarrow} \mathsf{OMA}(\mathsf{OMS}(\mathsf{c}), x_1, \ldots, x_n, L)$;
- for arity binder, $\mathsf{c}_{\mathsf{scala}}(V, t) \overset{\mathsf{apply}}{\rightsquigarrow} \mathsf{OMBIND}(\mathsf{OMS}(\mathsf{c}), V, t)$.

Translation of view $\mathsf{A}_{\mathsf{MMT}}$ View $\mathsf{A}_{\mathsf{MMT}}$ is translated to the SCALA trait $\mathsf{A}_{\mathsf{scala}}$ and the assignment $\mathsf{c}_{\mathsf{MMT}} = \mathsf{args} \Rightarrow \mathsf{code}$ declared in $\mathsf{A}_{\mathsf{MMT}}$ is translated to the SCALA function $\mathsf{a\_c}(\mathsf{args}) = \mathsf{code}$. The $\mathsf{a\_c}()$ function is applied by the Universal OpenMath Machine to those MMT terms whose head is c. For example, arith1_plus() is applied to terms of the form plus(x1, ...) (see Section 4.1 for more details). In this way, we have a bijection from object $\mathsf{c}_{\mathsf{scala}}$ to function $\mathsf{a\_c}()$, which ultimately means that $\mathsf{a\_c}()$ implements the OPENMATH symbol c.

Example We continue with the same example from the previous subsection, namely the complex1 CD. The complex1 object is shown in Figure 7 (we only show the apply/unapply methods for complex_cartesian) and the Complex1 trait is shown in Figure 8 (we only show the implementation of real). By calling the declares method at the end, the Universal OpenMath Machine collects and registers the implementation complex1_real(). This avoids reflection to obtain the list of all implementations.

```
object complex1 {
  object complex_cartesian {
    def apply(x1: Term, x2: Term) = OMA(OMID(this.path), x1 :: x2:: Nil)
    def unapply(t: Term): Option[(Term, Term)] = t match {
      case OMA(OMID(this.path), x1 :: x2:: Nil) => Some((x1, x2))
      case _ => None
    }
  }
  object complex_polar { ... }
  object argument { ... }
  object real {... }
  object imaginary {... }
  object conjugate {... }
}
```

Figure 7: SCALA object complex1

```
trait Complex1 extends complex1 {
  def complex1_complex_cartesian(re: Term, im: Term) : Term = {... }
  def complex1_complex_polar(r: Term, phi: Term) : Term = {...}
  def complex1_argument(z: Term) : Term = {... }
  def complex1_real(z: Term) : Term = {
    z match {
              case complex_cartesian(re, _) => re
              case complex_polar(r, phi) => arith1.times(r, transc1.cos(phi))
              case _ => real(z)
            }
  }
  def complex1_imaginary(z: Term) : Term = {...}
  def complex1_conjugate(z: Term) : Term = {... }

  declares(Implementation.A(complex1.real.path)(complex1_real _))
}
```

Figure 8: SCALA object Complex1

3.4 Summary

We summarize the workflow of mechanizing the OPENMATH CDs in Table 1. We assume an OPENMATH CD called a and a symbol c with arity 2 declared in a.

Declarative level MMT	theory a: OpenMath
	  c # c 1 2
		view A: a −> Scala
		  c = (Term, Term) "
		  Scala implementation of c
		  "

Computational level SCALA	object a {
	  object c {
	  val name = "c"
	  def apply(x1: Term, x2: Term) =
	  OMA(OMS(name), x1 :: x2 :: Nil)
	  }
	}
		trait A {
		  def a_c(x1: Term, x2: Term) : Term = {
		  // Scala implementation of c
		  }
		}

Table 1: Workflow summary

4 Implementation of DDC–Lite

Now that we discussed in section 3 how DDC-LITE works in theory, in this section we give an outline of how it is implemented.

The DDC-LITE system is made up of two components. The first component is a **library of implemented OpenMath symbols**. This is the **data + programs** part of DDC-LITE. The second component is the **Universal OpenMath Machine (UOM)** – the **engine** of the system. It *extracts* and *registers* the implementations from the library, and exhaustively *applies* them to OPENMATH objects.

Since users' contributions are more likely to be directed towards the library than towards the UOM, we separated the two components, allowing them to be developed independently. The UOM is a module within the MMT-API and the library is part of the oaff collection of archives.

4.1 The Universal OpenMath Machine

The main components of the UOM are:

- the **extractor**, which reads the MMT files from the library, creates SCALA files from them and writes them back to the library;
- the **simplification function**, which takes an OPENMATH object as argument and exhaustively applies the registered implementations on it;
- the **synthesizer**, which works the other way around as the extractor. More precisely, it reads the SCALA files from the library and creates OMDOC files from them. This is all it does at the moment, but the next step would be to translate the OMDOC files to MMT theories.

Let us now describe these components in more detail.

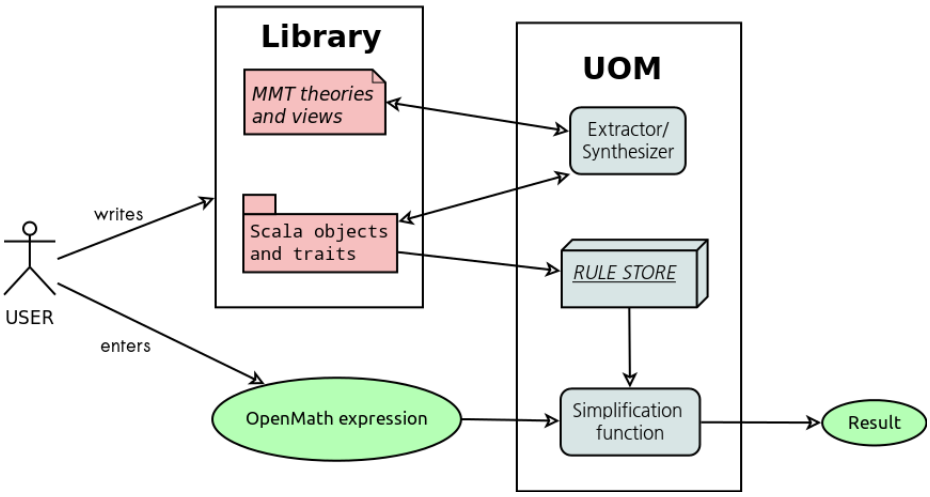


Figure 9: DDC-LITE framework

The extractor The theoretical process of obtaining SCALA objects and traits from the MMT theories and views was also described in Section 3. Let us just summarize the process here and use the same notations for theories, views, objects and traits as in Section 3. The extractor works in the following way.

For each MMT theory $\mathsf{a}_{\mathsf{MMT}}$ it creates a file with the same name. The file contains a trait $\mathsf{a}_{\mathsf{scala}}$ which declares a function $\mathsf{a\_c}$ for each constant $\mathsf{c}_{\mathsf{MMT}}$ in $\mathsf{a}_{\mathsf{MMT}}$ (e.g. for $\mathsf{list1.map}$, it creates the function $\mathsf{list1\_map()}$). The arity of the function is determined from the notation of the constant $\mathsf{c}_{\mathsf{MMT}}$. The extractor then creates an object $\mathsf{a}_{\mathsf{scala}}$ which declares object $\mathsf{c}_{\mathsf{scala}}$ for each constant $\mathsf{c}_{\mathsf{MMT}} \in \mathsf{a}_{\mathsf{MMT}}$, together with $\mathsf{apply/unapply}$ methods.

For each MMT view $\mathsf{A}_{\mathsf{MMT}}$, the extractor creates a file with the same name, which contains a trait $\mathsf{A}_{\mathsf{scala}}$ that extends trait $\mathsf{a}_{\mathsf{scala}}$ with the definition of the $\mathsf{a\_c}$ function. In the end, an object $\mathsf{A}_{\mathsf{scala}}$ that extends trait $\mathsf{A}_{\mathsf{scala}}$ is declared.

Additionally, the extractor creates methods for *registering* the implementations. The UOM maintains these registered implementations in a *store* S.

The simplification function We now describe how the simplification function, called $\mathsf{simplify}$, works. The function takes an OPENMATH object t (all objects are MMT Terms in the MMT-API) as argument and exhaustively applies the registered implementations to it, returning result $\mathsf{t'}$. Let us see how this is done.

Recall that the UOM keeps a *store* S of registered implementations. When the $\mathsf{simplify}$ function is applied to t, it traverses it and looks at its structure:

- if t is an $\mathsf{OMA}(\mathsf{OMS}(\mathsf{c}), \mathsf{args})$, the function recursively simplifies the arguments args to $\mathsf{args'}$ and then looks if there is any implementation of c in

S. If it finds implementation a_c, it returns simplify(a_c(args')). If it does not find any implementation, it returns OMA(OMS(c), args').

- if t is an OMS(c), then again, simplify checks for any implementation a_c of c in S, returning simplify(a_c()) if it finds one and t otherwise.
- in all other cases, simplify returns back t.

Since simplify recursively simplifies t, the result t' and the terms inside, if any, cannot be simplified anymore, using the implementations from S.

The synthesizer We use the same notations as in section 3. As mentioned, the synthesizer does the reverse job of the extractor. For each SCALA file containing the trait A_{scala}, it creates an OMDOC view A_{OMDOC}. For each function $a\_c(args)$ defined in A_{scala}, it extracts the code inside, creates an OMDOC constant of the form $c_{OMDOC} = OMBIND(args, code)$ and adds it to A_{OMDOC}.

The next step would be to extract MMT theories and views from the OMDOC files, such that constant $c_{OMDOC} \in A_{OMDOC}$ is translated to constant $c_{MMT} \in a_{MMT}$ with implementation $c_{MMT} =$ args "code" in view A_{MMT}. As a result, we get back the original MMT theories and views from the SCALA files.

The synthesizer is useful when users want to write the implementations in the SCALA files rather than in the MMT files, because most editors do not have syntax highlighting for SCALA code embedded into MMT.

4.2 The Library Infrastructure

Let us now describe how the library is organized. We outline the directory structure here and we enumerate the OPENMATH CDs that we chose to implement in Section 5.

The library is located in the openmath repository, which contains the MMT and the SCALA files used by the Universal OpenMath Machine. The top-level files and directories are:

- source/ – MMT theories and views;
- scala/ – corresponding SCALA files, generated by the UOM;
- compiled/, content/, narration/, notation/, relational/ – OMDOC files produced by the UOM when building the archive as well as MMT's indexes for the archive.
- bin/ – class files generated by compiling the scala directory;
- build.msl – script which runs the UOM extractor on the source/ directory and produces SCALA and OMDOC files;
- integrate.msl – script which runs the UOM synthesizer on the SCALA files and produces OMDOC files;
- interactive.msl – script which can be used to interactively test the UOM.

The source/ contains several directories and files:

- openmath.mmt – the OpenMath meta-theory;
- scala.mmt – the Scala meta-theory;
- bifoundations.mmt – the Syntactic view;
- cds/ – MMT theories, one for each OPENMATH CD;
- uom/ – MMT views, one for each theory in cds/;
- test/ – unit test cases.

The scala/ files are arranged by namespace in accordance with Java conventions:

- http..www.openmath.org/cd/ – the namespace of the OPENMATH CDs, one SCALA trait file for each MMT theory;
- http..oaff.kwarc.info/openmath/ – the namespace of the implementations, contains two directories:
  - test/ – test cases, translated to SCALA;
  - uom/ – SCALA files with objects corresponding to the MMT views;

Additionally, one file NAMESPACE.scala is generated for each directory. These files contain code that allows the UOM to iterate and register all implementations in the respective directory and run the test cases.

Any editor or IDE can be used to change the scala/ directory. For example, the root folder is at the same time an ECLIPSE SCALA project that can be imported into an ECLIPSE workspace.

5 A Library for DDC–Lite

In this section we enumerate the OPENMATH Content Dictionaries and the symbols declared in them that we chose to implement in the openmath library.

After surveying all the CDs in the CD groups repository, we identified a large set of symbols that are described in them. We implement most of the official CDs, but also several experimental CDs, because they either complement the official ones (for example, the experimental linalg4 contains relevant functions for matrices and vectors) or they contain fundamental mathematical datatpyes (such as polynomials, which are defined under the experimental polygrp CD group). During the selection process, we discarded objects that do not fit computation well, such as CD directives, the meta CD group or the functional operators (such as range, function inverse) since they may be difficult to represent and compute with.

In the next two subsections, we give an outline of all symbols currently implemented in the library. Each of the items listed in the tables are linked to a corresponding entry with the same name within the OPENMATH CD.

OPENMATH objects fall into two categories: basic and compound. The basic objects are the ones that are directly represented by OPENMATH literals and the compound objects are built recursively using OMA, OMS and OMBIND.

Also, note that all numerical types (such as integers, floats, rational numbers) have access to all arithmetical functions listed in sub-section 5.2.

5.1 Basic Objects

The OpenMath basic objects are integers, floats, strings and bytearrays. Below, we list their constructors together with the functions defined for them.

- **integers** have constructor OMI(n). Functions on integers are declared in the integer1 and combinat1 CDs and they are:

Object	**Description**
OMA (factorof, a, b)	true if a divides b
OMA (factorial, n)	$n! = 1 \cdot 2 \cdot \ldots \cdot n$
OMA (quotient, a, b)	a div b – integer division
OMA (remainder, a, b)	a mod b
OMA (binomial, n, k	$\binom{n}{k}$, n-choose-k
OMA (multinomial, $n, k_1, k_2, \ldots, k_p$)	$\binom{n}{k_1, k_2, \ldots, k_p} = \frac{n!}{k_1 k_2 \cdots k_p}, k_1 + k_2 + \cdots + k_p = n$
OMA (Stirling1, n, k)	Stirling numbers of the first kind
OMA (Stirling2, n, k)	Stirling numbers of the second kind
OMA (Fibonacci, n)	the n-th Fibonacci number, $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$
OMA (Bell, n)	number of partitions of a set with n elements
OMA (permutation, $a_1, \ldots, a_n$)	permutation, no arguments means identity permutation

  All arithmetical functions apply to integers.

- **floats** have constructor OMF(x), where x is an IEEE double-precision floating point number. All arithmetical functions apply to floats.

- **strings** have constructor OMSTR(s) where s is a Unicode Character string. Unfortunately, none of the standard OpenMath CDs define functions on strings. Instead, we define some ourselves:

Object	**Description**
OMA (strlen, s)	length of s
OMA (strcat, s, p)	concatenate s and p
OMA (substr, s, p)	returns true if s is a substring of p

- **bytearrays** have constructor OMB(b), where b is a sequence of bytes.

5.2 Compound Objects

Compound objects contain fundamental datatypes such as lists, sets, booleans, polynomials, vectors and matrices. As mentioned, they are built recursively using the OMA, OMS and OMBIND.

After exploring the OpenMath CDs, we have come up with the following list of constructed objects.

- **booleans** are defined in the logic1 CD: They are OMS (true) and OMS (false) and the following functions apply to them.

Function	**Description**
OMA (equivalent, A, B)	equivalence, $A \Leftrightarrow B$
OMA (implies, A, B)	implication, $A \Rightarrow B$
OMA (not, A)	negation, $\neg A$
OMA (and, $A_1, \ldots, A_n$)	n-ary conjunction, $A_1 \wedge \ldots \wedge A_n$
OMA (or, $A_1, \ldots, A_n$)	n-ary disjunction, $A_1 \vee \ldots \vee A_n$
OMA (xor, $A_1, \ldots, A_n$)	n-ary exclusive disjunction, $A_1 \oplus \ldots \oplus A_n$

- **inductive natural numbers** are defined in the indNat CD inductively, using OMS (zero) as the induction base and OMA (succ, n) as $n+1$, the successor of n. All arithmetical functions apply to them.

- **rational numbers** are constructed as OMA (rational, a, b). We assume a and b to be integers. All arithmetical functions apply to them.

- **complex numbers** are constructed as OMA (complex_cartesian, a, b) which stands for $a + \mathrm{i}b$, or as OMA (complex_polar, r, θ), which stands for $r\mathrm{e}^{\mathrm{i}\theta}$. There are several functions for them. Note that we have shown the implementation of real in section 3.

Function	**Description**
OMA (real, z)	$\mathrm{Re}(z)$ – the real part of $z = a + \mathrm{i}b$ is a
OMA (imaginary, z)	$\mathrm{Im}(z)$ – the imaginary part of $z = a + \mathrm{i}b$ is b
OMA (argument, z)	$\mathrm{Arg}(z)$ – the argument of $z = r\mathrm{e}^{\mathrm{i}\theta}$ is θ
OMA (conjugate, z)	the conjugate of $z = a + \mathrm{i}b$ is $\bar{z} = a - \mathrm{i}b$

  All arithmetical functions apply.

- **lists**. Since we cannot compute with infinite lists, our implementation assumes all lists are finite. List are constructed as OMA (list, $a_1, \ldots, a_n$), representing the list $[a_1, a_2, \ldots, a_n]$ and have the following functions defined on them:

Function	Description
OMA (map, f, L)	representing the list $[f(x)\|x \in L]$
OMA (suchthat, S, p)	representing the list $[x \in S\|p(x) = \text{true}]$
OMA (list_selector, i, L)	represents $L[i]$
OMA (first, L)	represents $L[1]$, the head of L
OMA (rest, L)	represents $L[2:]$, the tail of L
OMA (cons, x, L)	represents $x :: L$
OMS (nil)	the empty list
OMA (reverse, L)	reverses L
OMA (size, L)	returns the size of L
OMA (in, x, L)	means that $x \in L$

- **sets** As with lists, we assume all sets are finite. They are constructed as OMA (set, $a_1, \ldots, a_n$), representing the set $\{a_1, a_2, \ldots, a_n\}$ and they have the following functions defined on them:

Function	Description
OMA (cartesian_product, $A_1, A_2, \ldots, A_n$)	cartesian product
OMS (emptyset)	the empty set
OMA (map, f, S)	represents the set $\{f(x)\|x \in S\}$
OMA (size, S)	returns the size of S
OMA (suchthat, S, p)	represents the set $\{x \in S\|p(x) = \text{true}\}$
OMA (intersect, $A_1, A_2, \ldots, A_n$)	set intersection $A_1 \cap A_2 \cap \cdots \cap A_n$
OMA (union, $A_1, A_2, \ldots, A_n$)	set union $A_1 \cup A_2 \cup \cdots \cup A_n$
OMA (setdiff, A, B)	set difference $A \setminus B$
OMA (subset, A, B)	means that $A \subseteq B$
OMA (prsubset, A, B)	means that $A \subsetneq B$
OMA (notsubset, A, B)	means that $A \not\subseteq B$
OMA (notprsubset, A, B)	means that $A \not\subsetneq B$
OMA (in, x, S)	means that $x \in S$
OMA (notin, x, S)	means that $x \notin S$
OMA (lift_binary, op, A, B)	returns $\{x \text{ op } y\|x \in A, y \in B\}$

- **vectors** are constructed as OMA (vector, $a_1, \ldots, a_n$), representing the *row* vector $(a_1, a_2, \ldots, a_n)$ and **matrices** are constructed as OMA (matrix, $v_1, \ldots, v_n$), representing the matrix made from row vectors $v_1, \ldots, v_n$. The matrix construction is slightly different from the one OpenMath proposes, which is that a matrix is constructed from matrixrows. Since a matrixrow is the same as a vector anyway, we do not implement matrixrows and use vectors instead. Now, here are the functions for vectors and matrices:

Function	Description
OMA (vectorproduct, v_1, v_2)	$v_1 \times v_2$, where v_1, v_2 are 3-dimensional vectors
OMA (scalarproduct, v_1, v_2)	$v_1 \cdot v_2$, the dot product of v_1, v_2
OMA (outerproduct, v_1, v_2)	$v_1 \otimes v_2$, the outer product of v_1, v_2
OMA (transpose, M)	M^t, the transpose of matrix M
OMA (determinant, M)	the determinant of matrix M
OMA (vector_selector, i, V)	returns $V[i]$, where V is a vector
OMA (matrix_selector, r, c, M)	returns $M[r][c]$, where M is a matrix
OMA (eigenvalue, M, i)	the i-th eigenvalue of M, ordered by modulus
OMA (eigenvector, M, i)	the eigenvector of eigenvalue λ_i
OMA (characteristic_eqn, M)	the characteristic equation of M
OMA (size, v)	the size of vector v
OMA (rank, M)	the rank of matrix M
OMA (rowcount, M)	number of rows
OMA (columncount, M)	number of columns
OMA (identity, n)	I_n, the $n \times n$ identity matrix
OMA (zero, m, n)	O_{mn}, the $m \times n$ zero matrix
OMA (diagonal_matrix, $d_1, \ldots, d_n$)	diagonal matrix with $d_1, \ldots, d_n$ as diagonal
OMA (scalar, n, s)	the matrix $s \cdot I_n$

- **constants** are infinity, e, i, pi, gamma, NaN.
- **physical quantities** are constructed using times. For example a length of 10 kilometers is written as times(10, prefix(kilo, metre));
  - **dimensions:** there exists an OpenMath object for every dimension. They are: length, area, volume, speed, displacement, velocity, acceleration, time, mass, force, temperature, pressure, charge, current, voltage, resistance, density, energy, concentration, momentum;
  - **metric units:** metre, metre_sqrd, litre, metres_per_second, metres_per_second_sqrd; second, gramme, Newton, Joule, Watt, degree_Kelvin, Pascal, Coulomb, amp, volt, second, minute, hour, day, week, calendar_month, calendar_year;
  - **imperial and US units:** foot, yard, mile, acre, pint, miles_per_hr, miles_per_hr_sqrd, pound_mass, pound_force, bar, foot_us_survey, yard_us_survey, mile_us_survey, acre_us_survey, pint_us_dry, pint_us_liquid;
  - **SI prefixes** take a unit as argument and return a unit. They are manipulated using prefix and they are: yotta, zetta, exa, peta, tera, giga, mega, kilo, hecto, deka, deci, centi, milli, micro, nano, pico, femto, atto, zepto, yocto;

  We did not include degree_Celsius and degree_Fahrenheit, since they do not form quantities via times, in the sense that e.g. 2° C is not twice as much as 1° C.
- **polynomials** are either univariate or multivariate.

1. **Univariate polynomials** are defined using four constructors:
   - OMA (term, n, c) – monomial constructor, n is a non-negative integer and c is a coefficient;
   - OMA (poly_u_rep, $X, m_1, \ldots, m_n$) – X is the variable and $m_1, \ldots, m_n$ are the monomials. For example, $X^4 + 3X^2 - 2$ is represented as $\mathsf{poly\_u\_rep}(X, \mathsf{term}(4,1), \mathsf{term}(2,3), \mathsf{term}(0,-2))$;
   - OMA (polynomial_ring_u, R, X) – constructs a univariate polynomial ring $R[X]$;
   - OMA (polynomial_u, R, p) – constructs a univariate polynomial, R is a polynomial_ring_u and p is a poly_u_rep;
2. **Multivariate polynomials** are defined similarly to the univariate polynomials:
   - OMA (term, n, c) – n is a non-negative integer and c is a coefficient;
   - OMA (poly_r_rep,X, $m_1, \ldots, m_n$), where X is a variable and $m_1, \ldots, m_n$ are monomials. For example, $X^2Y^3 + X$ is represented as $\mathsf{poly\_r\_rep}(X, \mathsf{term}(2, \mathsf{poly\_r\_rep}(Y, \mathsf{term}(3,1))), \mathsf{term}(1,1))$;
   - OMA (polynomial_ring_r, R, $X_1, \ldots, X_n$) – constructs a multivariate polynomial ring;
   - OMA (polynomial_r, R, p) – constructs a multivariate polynomial, R is a polynomial_ring_r and p is a poly_r_rep;

There exists several constructors and functions for dealing with various kinds of polynomials:

- constructors:
  - OMA (power, p, n) – formal power, p is a polynomial and n is a non-negative integer;
  - OMA (factored, $p_1, \ldots, p_n$) – factorization constructor, p_i are irreducible polynomials;
  - OMA (squarefreed, $p_1, \ldots, p_n$) – square-free factorization constructor, p_i are square-free polynomials;
  - OMA (partially_factored, $p_1, \ldots, p_n$) – factorization constructor, nothing is assumed about p_i;
- functions:

Operator	Description
OMA (expand, f)	expands f into a polynomial
OMA (degree, p)	degree of polynomial p
OMA (degree_wrt, p, X)	degree of p with respect to variable X
OMA (leading_coefficient, p, X)	leading coefficient of p with respect to X
OMA (coefficient, p, $[X_1, \ldots, X_m]$, $[\alpha_1, \ldots, \alpha_m]$)	the coefficient of $X_1^{\alpha_1} \cdots X_m^{\alpha_m}$ in p
OMA (coefficient_ring, p)	the coefficient ring
OMA (evaluate, p, $x_1, \ldots, x_n$)	evaluate $p(x_1, \ldots, x_n)$
OMA (factor, p)	decomposes p into irreducible factors
OMA (squarefree	decomposes p into square-free factors
OMA (gcd, $p_1, \ldots, p_n$)	the greatest common divisor of the polynomials
OMA (lcm, $p_1, \ldots, p_n$)	the least common multiple of the polynomials
OMA (discriminant, p, X)	the discriminant of p with respect to X
OMA (resultant, p, q, X)	the resultant of p and q with respect to X
OMA (convert, p, R)	converts p into a polynomial over ring R

- **arithmetical functions** hold for all numerical types.

Function	Description
OMA (lcm, $a_1, \ldots, a_n$)	the least common multiple of $a_1, \ldots, a_n$
OMA (gcd, $a_1, \ldots, a_n$)	the greatest common divisor of $a_1, \ldots, a_n$
OMA (plus, $a_1, \ldots, a_n$)	n-ary commutative plus, $a_1 + \cdots + a_n$
OMA (unary_minus, a)	unary minus, the additive inverse $-a$
OMA (minus, a, b)	binary minus, $a - b$
OMA (times, $a_1, \ldots, a_n$)	n-ary multiplication, $a_1 a_2 \cdots a_n$
OMA (divide, a, b)	binary division, $a/b, b \neq 0$
OMA (power, a, x)	power, a^x
OMA (root, x, n)	root, $\sqrt[n]{x}$
OMA (abs, a)	$\lvert a \rvert$, the absolute value, or modulus in $\mathbb{C}$
OMA (sum, I, f)	summation over interval, $\sum_{x \in I} f(x)$
OMA (product, I, f)	multiplication over interval, $\prod_{x \in I} f(x)$
OMA (times, $a_1, \ldots, a_n$)	n-ary **commutative** multiplication, `arith2` CD
OMA (inverse, a)	inverse of an element, a^{-1}

- **transcendental functions** are: log, ln, exp, sin, cos, tan, sec, csc, cot, sinh, cosh, tanh, sech, csch, coth, arcsin, arccos, arctan, arcsec, arccsc, arccot, arcsinh, arccosh, arctanh, arcsech, arccsch, arccoth;

- **miscellaneous functions:**

Function	Description
OMA (eq, a, b)	equality, $a = b$
OMA (neq, a, b)	inequality, $a \neq b$
OMA (lt, a, b)	less than, $a < b$
OMA (leq, a, b)	less than or equal to, $a \leq b$
OMA (gt, a, b)	greater than, $a > b$
OMA (geq, a, b)	greater than or equal to, $a \geq b$
OMA (approx, a, b)	approximately equal, $a \approx b$
OMA (max, S)	the maximum element in set S
OMA (min, S)	the minimum element in set S
OMA (ceiling, x)	ceiling function, $\lceil x \rceil$
OMA (floor, x)	floor function, $\lfloor x \rfloor$
OMA (trunc, x)	round x towards 0, e.g. $\mathbf{trunc}(3.5) = 3, \mathbf{trunc}(-3.5) = -3$
OMA (round, x)	round to nearest integer

6 User Interface

In this section we describe a simple user interface of DDC-LITE and how to use it to evaluate expressions. We have seen in Section 4 that users send an expression E to the Universal OpenMath Machine, which uses the registered implementations to return a result E' (usually E' is a simplified version of E).

To run the Universal OpenMath Machine, we execute the interactive.msl script in the openmath folder. This opens the SCALA shell (with all the implementations registered) where users can enter expressions to be evaluated.

An expression is written as a SCALA *processed string* which is *interpolated* into a SCALA expression.

Processed strings A processed string [Ode12] is of the form

$$\texttt{id"text}_0\ \$\{\ \texttt{expr}_0\ \}\ \texttt{text}_1\ \cdots\ \$\{\ \texttt{expr}_n\ \}\ \texttt{text}_n\ \texttt{"}$$

where `id` is an interpolation function, $\texttt{text}_i$ are strings and $ escapes into SCALA: $`var` for variables and ${`expr`} for more complex expressions. The interpolation function processes the string and also performs type-checking for the typed expressions $\texttt{expr}_i$.

Integration with the Universal OpenMath Machine Integers and floats are implicitly converted by the UOM into OMI's and OMF's, respectively[IMR13]. We also implement two additional interpolation functions:

- `mmt` – calls the MMT parser to interpolate the string into an MMT Term;
- `uom` – first uses `mmt` and then calls the UOM simplification function on the MMT Term.

For example, we can write

```
> var x = mmt"2 + 3"
x: Term = plus(OMI(2), OMI(3))
```

Integers 2 and 3 were implicitly converted into OMI(2) and OMI(3) and the + was recognized as the notation for the MMT constant plus. As discussed in 3.3, the result is equivalent to $\mathsf{OMA}(\mathsf{OMS}(\mathsf{plus}), \mathsf{OMI}(2), \mathsf{OMI}(3))$.

Now, in order to evaluate the expression $2 + 3$ we use the `uom` function. We can either escape variable x:

```
> uom"$x"
res1: Term = OMI(5)
```

or we can write the whole expression:

```
> uom"2 + 3"
res2: Term = OMI(5)
```

We can further use the result in a new computation:

```
> uom"5 * $res2"
res3: Term = OMI(25)
```

We can also escape back and forth between MMT and SCALA. For example, we can run OPENMATH computations for all elements in a SCALA list:

```
> List(1,2,3).map(x => uom"x ^ 2")
res4: Term = List(OMI(1), OMI(2), OMI(3))
```

The fact that we can combine SCALA and OPENMATH computations essentially means that we can efficiently integrate object-oriented and rule-based computation into an extremely simple user interface.

7 Conclusion and Future Work

We described a novel approach towards creating a declarative, deductive and computational system. By using MMT as a common declarative component for two subsystems, one which combines the declarative and deductive aspects and another one which combines the declarative and computational aspects, we can later on also combine the deductive and computational aspects. The fact that MMT is logic-independent allows us to specify arbitrary logics and programming languages, which makes our system fully extensible. This is a major advantage over other systems which try to combine computation and deduction.

Additional work on DDC-LITE may include providing support for programming languages other than SCALA, creating concrete applications such as the ones described in [Zam11] (unit conversion and web-based education) and also designing an improved graphical user interface (as opposed to the current command line interface). With regards to the whole DDC system, future work should be concentrated on creating the other subsystem and after this is completed, our goal will be to connect the deductive and computational components.

References

[Axi] Axiom. See http://www.axiom-developer.org/.

[BC04] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[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.

[DM05] D. Delahaye and M. Mayero. Dealing with Algebraic Expressions over a Field in C oq using M aple. *Journal of Symbolic Computation*, 39(5):569–592, 2005.

[foc] Focalize. See http://focalize.inria.fr/.

[Geu09] H. Geuvers. Proof assistants: History, ideas and future, 2009.

[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.

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

[IMR13] M. Iancu, F. Mance, and F. Rabe. The Scala-REPL + MMT as a Lightweight Mathematical User Interface. 2013.

[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.

[Koh06] Michael Kohlhase. *OMDoc – An open markup format for mathematical documents*. Number 4180 in LNAI. Springer, 2006.

[Map] *Maple*. Waterloo Maple Inc., Waterloo, Ontario, Canada. http://www.maplesoft.com/products/Maple/features/index.aspx.

[Mat] *Mathematica*. Wolfram Research, Inc., Champaign, IL, USA. http://www.wolfram.com.

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

[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.

[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.

[Sca] The scala programming language. http://lamp.epfl.ch/scala/.

[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.

[Wen09] M. Wenzel. The Isabelle/Isar Reference Manual, 2009. http://isabelle.in.tum.de/documentation.html, Dec 3, 2009.

[Zam11] V. Zamdzhiev. Universal OpenMath Machine, 2011. Bachelor's thesis, Jacobs University Bremen.