A Universal Machine for Biform Theory Graphs

Michael Kohlhase Felix Mance Florian Rabe

Computer Science, Jacobs University Bremen

Calculemus, CICM, July 2013

Mathematical Knowledge Representation

Three aspects of mechanized representations:

- **declarative**

$$\mathsf{plus} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$

- **deductive**

$$\forall x, y \in \mathbb{R}, \mathtt{plus}(x, y) = \mathtt{plus}(y, x)$$

- **computational**

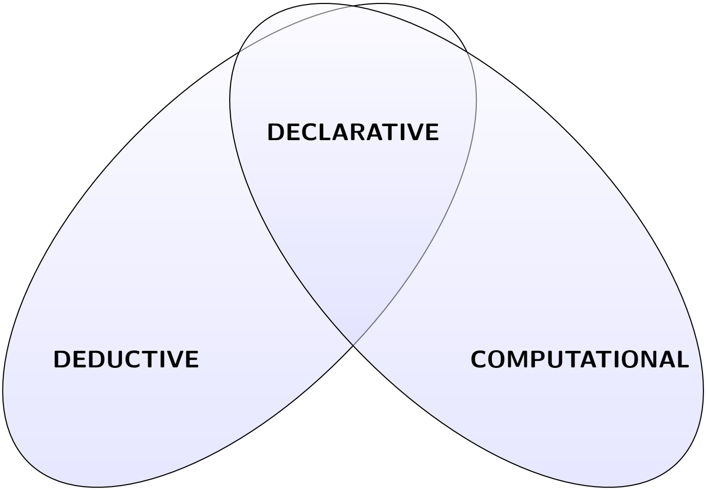
```
fun plus(x:real, y:real) = x + y
```

Motivation

- Mathematical practice involves all 3 aspects

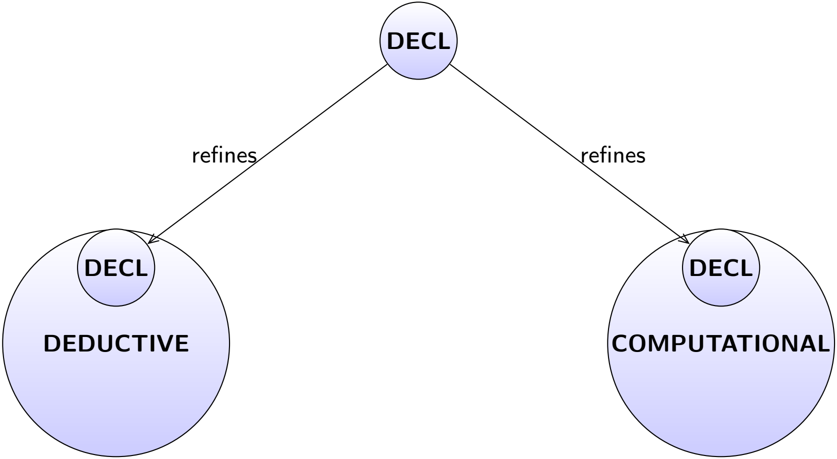
  and jumps between them seamlessly
- But: Mechanized systems tend to focus on 1-2 aspects
  - declarative representation languages
  - deduction systems
  - computer algebra systems
- Large body of research, but still no satisfactory result

Relation between Aspects



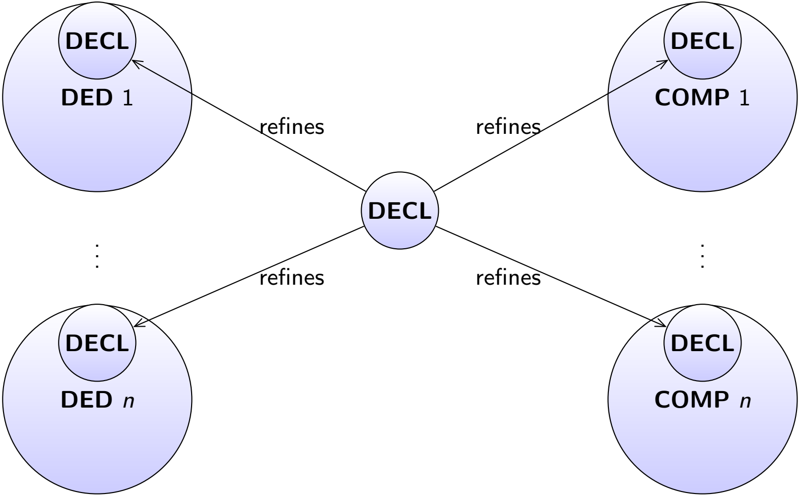
The declarative aspect is shared between the computational and the deductive representation.

Actually it looks like this



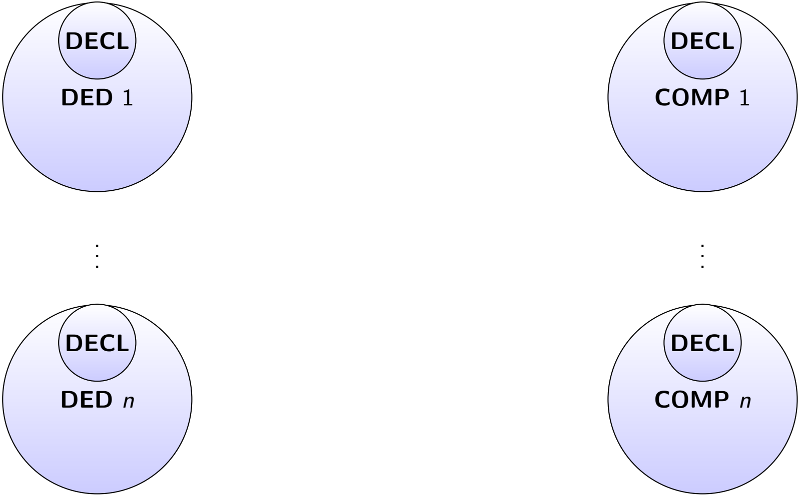
Deductive and computational systems redo the declarative part at least partially.

Actually it looks like this



Multiple deductive and computational refine the same declarative representation differently

Actually it looks like this



The various refinements are not actually spelt out.

Observations on Meta-Theories (1)

- Every representation uses a meta-theory at least implicitly
- Deductive: the logic underlying the deduction system
  e.g., Isabelle/HOL, Coq
- Computational: the language of the environment
  programming language, built-in types/values
- Declarative: the type system/logic needed for the occasion
  e.g., first-order logic for algebra

Observations on Meta-Theories (2)

- The choice of meta-theory follows different trade-offs
- Deductive/computational meta-theory should
  - have few primitives
  - permit rich structure of conservative extensions

  e.g., in set theory: $0 = \{\}$, $succ(n) = n \cup \{n\}$

  necessary to justify investment theorem prover, compiler, ...
- Declarative meta-theory should
  - be as weak as possible
  - avoid commitment

  e.g., Peano axioms

  necessary to maximize refinement options

Long Term Goal

1. Represent declarative aspects in weakly committed meta-theory
   no fixed logic, no fixed programming language
   many, flexibly custom-fitted meta-theories
2. Refine it in various deductive/computational system
   specific logics and/or programming languages
   prove theorems, implement functions
3. Shared declarative representation provides interface between systems

Note:

- Declarative language expressive enough for most questions and answers
- Refined representations mainly needed to *find* the answer

Our Contribution Here

- Start with the declarative aspect MMT
- Make MMT computation-aware new: universal machine
- Represent computational languages in MMT
  new: biform theory graphs in MMT
- No integration with computer algebra systems yet future work

So What's MMT?

- Universal framework for formal mathematical/logical content
  - declarative representations of interrelated languages
  - explicit modular meta-theories — little meta-theories
  - choose meta-theory flexible
  - move representations across meta-theories
- Close relatives
  - logical frameworks like LF, Isabelle
    but: more generic, heterogeneous
  - OMDoc/OpenMath
    but with formal semantics, more automation
- Main paper: Rabe, Kohlhase, *A Scalable Module System*, Information & Computation, 2013
- $\sim$ 10 CICM papers on individual aspects of the implementation

Central Idea: Foundation-Independence

1. We can fix and implement a logical theory e.g., set theory
2. We can fix and implement a logic
   then *define many theories in it* e.g., first-order logic
3. We can fix and implement a logical framework
   then *define many logics in it* the foundation, e.g., LF
4. We can fix and implement a meta-framework
   then *define many logical frameworks in it*
   foundation-independence: MMT

A Small Formalization Example in MMT

The logical framework LF in MMT:

```
theory Types { type }
theory LF {include Types, Π, →, λ, @ }
```

First-order Logic defined in MMT/LF:

```
theory Logic meta LF {o: type, ded : o → type }
theory FOL meta LF {
  include Logic
  u: type. imp: o → o → o, ...
}
```

Algebraic theories in MMT/LF/FOL:

```
theory Magma meta FOL { ∘: u → u → u }
...
theory Ring meta FOL {
  additive: CommutativeGroup
  multiplicative: Semigroup
  ...
}
```

MMT as a Universal Machine

- New component of MMT system
  - maintains set of computation rules
  - provides service for exhaustive rule application

  HTTP, API, Scala interpreter, OS shell

- Very general perspective:
  a rule for symbol s is a function that
  - takes any $OMA(OMS(s), arg_1, \dots, arg_n)$
  - returns some other object
- For example:
  - $OMA(OMS(\texttt{plus}), OMI(2), OMI(3), OMV(x))) \rightsquigarrow$
    $OMA(OMS(\texttt{plus}), OMI(5), OMV(x))$
  - $OMA(OMS(\texttt{integral}), f) \rightsquigarrow$
    what Mathematica says
  - $OMA(OMS(\circ), OMV(x), OMS(e)) \rightsquigarrow$
    $OMV(x)$ (in a monoid)

Feeding the Universal Machine

- MMT takes rules from anywhere
  - hand-written — in any programming language
  - normalization rules of type checker — e.g., β-reduction for LF
  - generated from declarative specification — e.g., algebra
  - exported from deductive system — e.g., Isabelle code generation
  - wrapper for external computational system — e.g., Mathematica
- MMT
  - maintains sources of rules
  - determines applicable rules

Our Case Study

1. Written a set of declarative specifications in MMT
   - meta-theory: OpenMath
   - specifications: OpenMath standard CDs

     arith, linalg, lists, sets, logic, relations, ...
2. Translated to a computational system
   - meta-theory: Scala
   - refinements: implementations of the CDs

     example: arith1 for integers, arith1 for vectors, ...
3. Each refinement yields a bunch of rules

- Why OpenMath: simplest possible meta-theory almost empty
- Why Scala: rules can be loaded by MMT

  same programming language

Theory-Implementation Codevelopment in MMT

- Automated translation

  MMT theory hierarchy $\longleftrightarrow$ Scala class hierarchy

  bijective, preserves module system
- Theories developed in MMT, implementations developed in a Scala IDE MMT project is also elicpse project

---

MMT theory based on OpenMath:

theory *om.arith*1 **meta** *OpenMath* $=$
  $plus : Obj \times Obj \to Obj$

MMT theory based on Scala (generated):

theory *sc.arith*1 **meta** *Scala* $=$
  $plus : (Term, Term) \Rightarrow Term$

Scala class (generated)

abstract class *arith*1 $\{$
  def
$plus(x : Term, y : Term) : Term$
$\}$

Term: type of OpenMath objects in MMT system

Theory-Implementation Codevelopment in MMT (2)

- Scala snippets embedded into MMT source files

  partially parsed by MMT
- Scala snippets may
  - refer to previously defined functions
  - use intuitive constructors+pattern matchers

    automatically generated by MMT
- Scala snippets edited/compiled using Scala IDE
- Edited code and compiled binaries loaded back into MMT

---

view *integers* **from** *sc.arith1* **to** *Scala*

$plus = (x : Term, y : Term) \Rightarrow$ "*scala*

```
(x, y) match {
  case (OMI(a), OMI(b)) => OMI(a + b)
  case (a, arith1.unary_minus(b)) =>
    arith1.minus(a,b)
  case _ => OMA(plus, x, y)
}
```

"

Our Case Study as a Theory Graph

- s: translation MMT/OpenMath $\longrightarrow$ Scala
  - theories (i.e., CDs) become abstract classes
  - theory inclusion becomes class extension
  - theory morphisms between CDs become functors
- s^T: induced translation of OpenMath objects to Scala expressions
- *integers*: implementation of arith1 for numbers

$$
\begin{tikzcd}
\text{OpenMath} \arrow[r, "s"] \arrow[d, hook] & \text{Scala} \arrow[d, hook] \\
\text{arith1 CD} \arrow[r, "s^T"] & \text{abstract class arith1} \arrow[u, bend right=60, "\text{implementing class } \textit{integers}"']
\end{tikzcd}
$$

General Case: Biform Theory Graphs

- L: Declarative specification language
  e.g., first-order logic
- T: Specification
  e.g., rings, integers
- P: Realization language
  - programming language
    or
  - primitive concepts of computer algebra system
- s: refinement L to P
  possibly partial, e.g., drop axioms

- $s(T)$ translated version of T in simple cases: pushout
- s^T: induced encoding of L-expressions in P

$$\begin{tikzcd}
L \arrow[r, "s"] \arrow[d, hook] & P \arrow[d, hook] \\
T \arrow[r, "s^T"] & s(T)
\end{tikzcd}$$

Note: same picture applies if P is deduction system

Putting Things together in MMT

1. Develop declarative theory graph in MMT

   e.g., algebra in MMT/FOL

2. Translate theories to a more refined meta-theory

   algebra in MMT/Scala

   - for operations: just pushout
   - for axioms: generate unit tests

3. Generate (abstract) Scala classes from MMT/Scala theories

   trivial step

4. Implement abstract classes in Scala IDE
5. Merge edited code back into MMT source
6. Load compiled rules into universal machine

2, 3, 5, 6 automated by MMT system
user focuses on 1, 4

Conclusion and Future Work

- Good understanding of MMT as interface framework
- Develop more translation+code generation pipelines
  current targets: Python+Sage, OpenAxiom, ...
- Uniformly generated classes provide interface between target systems
- Dually: export CAS code base as MMT theories
  easy for Sage using Python code introspection
- Relate MMT-generated classes to existing CAS classes
- Code generation leverages known relations
  automatically generate converter functions