Reflecting Declarative Rule Implementations

MSc Thesis

Aivaras Jakubauskas

September 18, 2013

Abstract

Combination of computation and deduction is a still persistent problem in computer-aided mathematics. Attaching an axiomatic framework to a computational algebra system or alternatively enriching a deductive system with computational capacities are two main approaches to this problem. Our approach takes the third path: we attempt to achieve computation and deduction in a declarative framework. Module System for Mathematical Theories, a foundation-independent declarative system, serves as the basis for our implementation.

We seek to automatically generate the implementations for certain rule declarations. These rules then can participate in term evaluation and assist theory validation. Instead of implementing the rules and providing the framework with them the user is allowed to merely declare them - the framework does the rest.

The underlying idea behind our proposed implementation generation is to interpret the rule declarations as rewrite rules and match their left-hand-side to a particular pattern. We apply a term substitution algorithm for generation of mathematical simplification rules (such as $x \circ e = x$). We observe that most mathematical rules are of the form $x \oplus (y \otimes z)$, that is there are two operators $\oplus$, $\otimes$ acting on variables or vectors of variables x, y, z. In our framework it comes down to substituting x, y, z on the right-hand-side of the rule declaration with the corresponding matches in the said form.

We seek to generalize our approach and provide an overview of other possible types of rules that could potentially be expressed in the declarative framework.

Contents

1 Introduction 2

2 Preliminaries 4
- 2.1 The MMT Language . . . 4
- 2.2 The MMT System . . . 5
- 2.3 Logical Framework . . . 7
- 2.4 Group Theory in MMT . . . 7

3 Rule-based Computation in MMT 8
- 3.1 Motivation . . . 8
- 3.2 Generic Form of Computation Rules . . . 11
- 3.3 LF as a Rule Definition Language . . . 12
- 3.4 Reflecting Rules into MMT . . . 13

4 Implementation 16

5 Towards a Generalized Rule Reflection Architecture 17

6 Applications 19
- 6.1 Examples . . . 20
  - 6.1.1 Algebra . . . 20
  - 6.1.2 Propositional Logic . . . 21
  - 6.1.3 Arithmetic . . . 21
  - 6.1.4 Type Theory . . . 23
- 6.2 Interactive Computation . . . 23
- 6.3 Computation in Validation . . . 24
  - 6.3.1 ADTs . . . 24
  - 6.3.2 Homotopy Type Theory . . . 24

7 Future Work 24

8 Conclusion 25

References 25

1 Introduction

Broadly speaking computer assistance in mathematical sciences manifests itself in two ways: computation and reasoning. Computational software is widely applied in industry and sciences to process large amounts of data. Computation assists scientific thought by providing powerful processing and presentation tools: it allows humans to understand the data and reason about it. Whereas machine reasoning allows to verify the data and computational algorithms, detect inconsistencies and draw previously unobserved conclusions. These two aspects of computer support complement each other.

Computational algebra systems (CAS) usually come with a programming language and allow the users to easily implement new functions. They also of provide libraries of already existing implementations, thus facilitating most computational needs. Whereas theorem provers (TP) are designed with mechanized deduction in mind. TPs provide a declarative language that allows for a wide scope of mathematical expressions. The declarations can be proved or disproved by the TP. Ideally a software dedicated to support of mathematics should allow the user to both declare new operations and prove their correctness.

However combining computation and deduction in a single framework is still an open problem. Numerous attempts have been made and several methods have been developed. Two immediate approaches at solving the problem are either adding computations to an existing deduction system or extending a computational system with a logic. Additionally, a middle paths exist in the form of rewrite rules and logic programming. The former gives computational semantics to rules by rewriting logical expressions. The latter uses procedural interpretation of logical declarations.

What joins the realms of computation and deduction is a declarative framework. Both CASs and TPs are usually coupled with a declarative system that facilitates adding new mathematical operations to the former one and new axioms and deduction rules to the latter one. A purely declarative system avoids the concerns of computation and deduction, and instead focuses on expressibility and extensibility.

Our Approach We approach the problem of integrating computation and deduction from the perspective of a purely declarative system. Module System for Mathematical Theories (MMT [Rab]) serves as the basis of our method.

MMT is an extensible foundation-independent framework with an OMDoc [Koh06] based declarative language. It is customizable for any logical foundation by providing urtheories and deduction rule implementations via plug-ins. In our case study the urtheory in the backend is **LF**.

The foundation-free architecture of MMT allows for a great range of expressibility, however it also tasks the user with implementing the logics. Our contribution to the MMT API consists in eliminating this task for a certain type of rules.

We provide a plug-in which, once attached to the MMT framework, generates implementations for a certain kind of declarations. The said declarations come in a form of rewrite rules. Their respective implementations are registered by the MMT rewrite rule collection and can be accessed by the framework. The rules are used for simplification of declared terms.

Consequently the user does not need to provide implementations for many basic rules - they are automatically generated. Another benefit arises when it comes to theory validation. Since the implementations are known to the framework upon reading the declaration of a rule, it can be applied to any consecutive terms. Thus rewrite rules can participate in further term validation.

Similar Work There are a few actively developed deductive systems that strive to join computation and deduction in one framework.

Mizar [TB85] aids the user in defining mathematical theories by providing keywords. Keywords like "symmetric" automate assigning certain properties to declarations. For example an equality relation can be declared and adorned with a keyword "symmetric" which would let the framework assume certain features of equality that are not explicitly declared. In a way Mizar keyword support can be viewed as an implementation generation.

Coq[The11]is an interactive theorem prover based on calculus of inductive constructions[coq]. Coq supports *tactics* – rules that given a proof goal reduce it to subgoals. Tactics correspond to deduction rules in formal logics (such as natural decution rules), except that they work backwards – given a proof goal they produce the premise of a rule. The user may also write one's own tactics and have them loaded into library.

Dedukti [SP] is a universal proof checker based on $\lambda\Pi$-calculus module formalism. It allows the user to declare modules - self-contained logical theories. Modules encapsulate declarations, definitions, rewrite rules and may import annotations. Rewrite rules may induce computation on declared terms.

HOL[GM93] and it's derivative **Isabelle**[NPW02] are interactive theorem provers based on higher- order logics as their meta-logic. Similarly to Coq, they implement a set of hard-coded tactics to facilitate proof derivation.

An opposite approach is to introduce a declarative framework and validation in a computational system. There are a few prominent attempts by Computational Algebra System (CAS) developers to introduce logical validation in their frameworks.

Wolfram **Mathematica**[Mat12] provides a large collection of hard-coded op-

erations. Computable Document Format (CDF [wol]) comes as a representation format with an interface to Mathematica, which allows for creation of computing documents. A move towards representing logics in CDF was made [CDF]: an instance of propositional logic (PL) is presented in CDF. Even though it can be considered a first significant step of the system towards the extension with capacities of inference systems, it must be noted that PL is a decidable logic, thus a computationally manageable logic. It is noteworthy that a move beyond PL presents a significant challenge for computational systems.

MathScheme [CF11] project - a research project in early development phase, aims to develop a hybrid system capable of both deduction and computation. This project stands out in it's effort to develop a completely new approach in mechanised mathematics to achieve the said goals.

Computational Algebra Systems excel at computational support yet often lack any verification of correctness. The main problem is rooted in the fact that CAS rely on a fixed implementation language. Even though new implementations and operations can be easily introduced, a CAS usually lacks the flexibility of a declarative framework and is limited in the scope of logics it can implement.

We further elaborate on the set-up of MMT framework in section 2. The motivation and description of rules and their generation in the system is covered in section 3. An elaboration of the implementation of our rule generator is provided in section 4. In section 5 we generalize our approach to rewrite rules and suggest a judgement-based rule hierarchy. Case studies of our implementation generator are described in section 6, where we describe the libraries which use generated rewrite rules for computations and validations.

2 Preliminaries

MMT framework (see 2.1) constitutes to the basis of our implementation of computational rules. However, MMT by itself is not enough - it is only capable of representation of logics. In section 2.3 we introduce the logical foundation for our logics we will be working with. In section 2.1 we further elaborate how theories are represented within MMT with LF logic framework in the background. We present an explicit example of a theory declaration in section 2.4.

2.1 The MMT Language

Module System for Mathematical Theories (MMT [Rab]) is a foundation and logic independent framework coupled with an OMDoc based MMT *language*.

Module System for Mathematics is foundation-independent modular framework. In this paper we refer to MMT API simply as MMT and to MMT declar-

ative language as MMT *language*. Theories declared in text files that MMT API can process are written in *mmt* suffixed files.

Theories and their constituents are represented using the following grammar.

theories	$G := \cdot \mid G, T^{[M]} = \{\Sigma\}$
constant declarations	$\Sigma := \cdot \mid \Sigma, c : A$
terms	$A := s \mid x \mid\ App(A, A^*) \mid\ Bind(A, (x : A)^*, A^+)$
context	$\Gamma = \cdot \mid \Gamma, x : A$

Figure 1: Fragment of MMT Grammar

A declared theory T may have a meta-theory M.

Constant c of type A is declared as $c : A$.

Type A is constructed by terms, which can be constants s, variables x, term applications $App(A, A^*)$ where first argument is applied to the rest, or more general term binders $Bind(A, \Gamma, A)$, where first argument represents the binder, Γ has the context and the last argument is bound.

Theory context may be encoded as a list of variable declarations $x : A$, which is abbreviated by Γ.

Typing Judgement	$\vdash_\Gamma E : E'$
Equality Judgement	$\vdash_\Gamma E = E'$

Figure 2: LF Judgements

2.2 The MMT System

MMT System takes theory declarations as input and can validate them and, given corresponding plug-ins, perform computations on them. The following figure 3 presents the main components involved in the said processes.

Input may be provided in various languages (XML, text etc.) or as MMT language expressions via console.

Validator type checking and type reconstruction. Bi-directional type checking with constraint delay to solve for missing subterms, e.g. types with bound variables and implicit arguments. Namely, every declaration is examined down to basic expressions, which existence can be looked-up in the library. Processed theories are stored in the Library. Validator uses rules to infer and type-check, equate terms. The function of computation on terms is left for Simplifier.

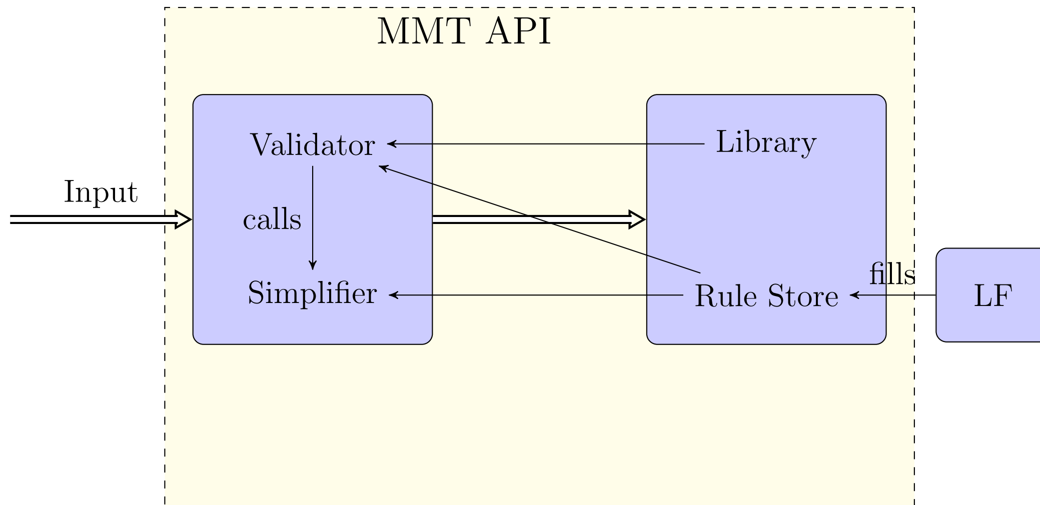

Figure 3: MMT API Architecture

Simplifier is designed to evaluate or simplify MMT terms. It is a rule-based engine that applies rules exhaustively until no further simplification is possible. No rules are hard-coded in the API, instead they are provided by plug-ins. The respective plug-in has to be called when processing a file that needs rule implementations, i.e. for **LF**-based theories an **LF** plug-in must be called.

Library is the run-time theory container. Stored theories maintain modular hierarchy - imports are referred to within a theory. Storing rules in 1

Rule Store is a module that contains all registered rules. For example LF plug-in provides rules that will be registered if the MMT API is notified of the plug-in. In the Rule Store rules are sorted by their function: type inference, equality check etc. However the rule store does not distinguish rules by what are they supposed to be applied to - the validator fetches all rules of certain type and then matches the rule input pattern.

LF ([RS09]) is a Logical Framework plug-in. Most of MMT theories are written with LF as a foundation. MMT supports LF via plug-in that implements type inference, equality, simplification and other rules. Note, that these rules are hard-coded.

2.3 Logical Framework

MMT provides us with a representation framework. However, we must ground our theories in a logical framework. Logical framework LF [RS09] is the default logical foundations used by MMT, although MMT language allows for any other foundation.

LF expressions declare new types, variables and constants. They also produce dependent function types, λ-abstractions and applications. Contexts in LF correspond to MMT contexts.

Symbol	Abstract Expression	Notation
Π	$Bind(\Pi, \Gamma, E)$	$\Pi_\Gamma E$
λ	$Bind(\lambda, x, E)$	$\lambda x.E$
@	$App(@, E, E')$	$E@E'$
$type$	$App(type, E)$	$E : type$

Figure 4: LF Representation in MMT

[1] The support for handling expressions and judgements is implemented via an MMT plug-in (see previous section 2.2). Provides 10 typing equality rules and 1 simplification rule. 3 type inf type check 6 congruence 1 simplification EdN:1

2.4 Group Theory in MMT

To clarify how theories are declared in MMT we demonstrate a Group declaration. A line-by-line overview of a theory declaration:

,

```
1 namespace http://cds.omdoc.org/rules
2
3 theory Group : http://cds.omdoc.org/urtheories?LF =
4   g : type
5   e : g
6   op : g → g → g # 1 ∘ 2 prec 5
7   inv : g → g # 1 -¹ prec 4
8   groupEq : g → g → type # 1 == 2 prec 0 role Eq
```

Figure 5: Group Theory Declaration

`namespace` declaration allows us to specify the theory name space and avoid ambiguities

[1]EdNote: LF is implemented in MMT framework.

all theories start with a keyword `theory` followed by a name, in this case `Group`

meta-theory for `Group` is

group elements are of type `g`

`e` is of type `g` - neutral element

`op` - group operation that takes an element of type `g`, another element of type `g` and returns a group element of type `g`. The notation for operations and constants can be specified as done further in the `op` declaration. Notations allow for a better human-readable presentation of theories. Operation precedence may also be specified as done here.

`inv` - inverse group element operation. Again, it includes a notation and precedence specification.

`groupEq` is a declaration for group element `g` equality. It has an additional `role` specification field, which may be referred to during theory validation.

3 Rule-based Computation in MMT

We provide a framework that takes declarative definitions of simplification functions and generates their implementations during runtime. Namely, the functions are defined in MMT files as entities in theories. We extract the computational semantics of given functions by matching their respective declarative statements to a specific pattern. This limits the variety of rules we can implement but we argue that our technique provides a significant amount of useful rules.

In section 4 we provide a general overview of our framework implementation. We demonstrate a library of rules we developed in section 6, where we apply them for both computation and validation.

3.1 Motivation

In order to understand how to make implementation generation possible for a generic declaration of a rule we need to conceptualize what rules are in the context of MMT and what exactly should the framework do with the declaration of a rule in order to produce an implementation.

Inductive functions happen to be ample in various fields of mathematics and often represent the basic inductive rules. Our observation that many algebra and arithmetic rules may be represented as inductive functions leads us to a formulation of a general pattern of such functions.

Example 1 (Algebra).
Let G be a group with an operation $\circ : G \to G$ and a neutral element e. Then for any $x \in G$:

- $x \circ e = x$ and $e \circ x = x$
- $(x^{-1})^{-1} = x$
- $x \circ x^{-1} = e$ and $x^{-1} \circ x = e$

The Group axioms may not be inductive, yet they can easily be seen as partial functions for input adhering to a certain pattern. For instance neutral element absorption happens when an element e is present on either side of the outer-most $\circ$ operator: $e \circ x$ or $x \circ e$ It is clear that neutral element absorbtion axiom is valid for any group element x. It also means that it should be valid for any expression of a group element, that is x can be $a \circ b$ for $a, b \in G$ and it would still be understood that $(a \circ b) \circ e = a \circ b$.

Example 2 (Arithmetics).
Arithmetics of Natural numbers as defined by Peano: 0 is a natural number, a successor function $s : \mathbb{N} \to \mathbb{N}$, addition $+ : (\mathbb{N}, \mathbb{N}) \to \mathbb{N}$ and multiplication $\times : (\mathbb{N}, \mathbb{N}) \to \mathbb{N}$

- $n + 0 = a$ and $0 + m = a$
- $s(n) + m = s(n + m)$ and $n + s(m) = s(n + m)$
- $n \cdot 0 = 0$ and $0 \cdot m = 0$
- $s(n) \cdot m = n \cdot m + m$ and $n \cdot s(m) = n + n \cdot m$

A few of the rules for Natural numbers are in fact similar to the Group axioms. It is inherently so due to the fact that groups can be mapped to naturals. However, in addition to that we have inductive rules for natural number addition and multiplication. Again they can be seen as functions that require a match on input. For example the addition rule $s(n) + m = s(n + m)$ is applicable to $s(s(0)) + 0$ but not applicable for $0 + s(s(0))$.

Example 3 (Logic).
Boolean algebra rules:

- $x \wedge \mathrm{F} = \mathrm{F}$ and $\mathrm{F} \wedge x = \mathrm{F}$
- $x \wedge \mathrm{T} = x$ and $\mathrm{T} \wedge x = x$
- $x \wedge (x \vee y) = x$

- $\neg\neg x = x$

Example 4 (Abstract Data Types)*.* Lists:

- $\text{head}(\text{cons } e\ l) = e$
- $\text{tail}(\text{cons } e\ l) = l$

More complicated rules may be constructed using simple rules that can also be expressed as inductive functions.

Example 5 (Complex Rules in Arithmetics)*.*

- $-n \cdot (-m) = n \cdot m$ can be derived from
  - $-n \cdot m = -(n \cdot m)$
  - $n \cdot (-m) = -(n \cdot m)$
  - $-(-n) = n$
- $(n+m) \cdot (n-m) = n \cdot n - m \cdot m$ can be derived from successive applications of the following:
  - $(n+m) \cdot k = n \cdot k + m \cdot k$
  - $n \cdot (m-k) = n \cdot m - n \cdot k$
  - term collection [1]

The prevalence of inductive rules together with their use in constructing more complicated rules lets us believe in the potential utility for automation of their implementation.

General discussion about representing mathematical rules in computer algebra systems leads to an observation that unlike in the eyes of a mathematician, the machines see rules as directed rewritings of terms. Rules, that a mathematician could see both as LHS = RHS and as RHS = LHS, in a machine language must be encoded as two different rules: LHS $\to$ RHS and RHS $\to$ LHS. The same conclusion holds in our implementation of rules in MMT as well.

[2] EdN:2

[1] The last step is term collection. Unfortunately this operation, although simple for humans to understand and relatively easy to implement on machines, has known to us formal description. We skip this step as it is already implemented in our system.

[2] EdNote: needs a development

3.2 Generic Form of Computation Rules

Our approach to generating rule implementations is syntax-based. In this section we describe the patterns of rules of interest. We also develop a more general theory about such patterns which may serve to facilitate implementation of less frequently occurring rules. In section **??** we provide a coverage of numerous mathematical rules adhering to the patterns described in this section.

The most general description of a rule in MMT is that of a function that takes a term and returns a term.

Definition 6 (Simplification Rule)**.** A Simplification rule is a function $Term \rightarrow Term$

Example 7 (Simplification rules)*.*

1) $X \wedge 0 = 0$, where $X \in \{0, 1\}$

2) $x = x$

3) $\sin(2k\pi + x) = \sin(x)$ for some $0 \leq x \leq \pi \in \mathbb{R}$

It is only implicit that a Simplification Rule actually returns a more simple term. Moreover, it is not exactly clear, what "more simple" means. Naively one could say a tree of the output term is more simple if it is of lesser depth than the one of the input term. However, an immediate counter-example is β-reduction, which may as well increase the depth of the term tree. Therefore we do not wish to specify what "simplification" means here and leave the title of the rules nominal.

In order to generate the implementation of the Simplification Rule, its input term must adhere to a certain shape. We implicitly describe that shape in the refinement of Simplification Rule definition - Depth_2 Rule. Note that we specify the definition with LF in mind.

Definition 8 (Depth_2 Rule)**.**
A declaration $c : \mathcal{A}$ in a theory T with meta-theory LF is called a Depth_2 rewrite rule over equality if $\mathcal{A}$ is of the form

$$Bind(=, App(@, \oplus, x_1, .., x_k, App(@, \otimes, y_1, .., y_l), z_1, .., z_m), rhs)$$

where $x_{1..k}$, $y_{1..l}$, $z_{1..m}$ are variables from Γ.

It is called **linear** if all $x_{1..k}$, $y_{1..l}$, $z_{1..m}$ are different (unique), otherwise – **non-linear**.

Example 9 (Depth_2 rule examples)*.*
The following are Depth_2 rules:

- $x \wedge false = false$, since *false* is treated as a symbol, i.e. $@(false, [\,]) \rightarrow false$
- $(s(n))! = s(n) \cdot n!$
- $head(cons\ x : a\ y : List(a)) = a$, where x is of type a and y is a list of type $List(a)$
- $x \vee (x \wedge y) = x$ is a non-linear Depth_2 rule

Generalizing the definition of Depth_2 rule will provide us with a wider coverage of inductive rules in mathematics. However, there are not many Depth_3 or more complex inductive rules. Depth_1 rules constitute to a marginal set of mathematical rules as well.

Definition 10. Depth_n rule:

$$Bind(=, App(@, o_1, \vec{x}^1, App(@, o_2, \vec{x}^2, .., App(@, o_n, \vec{y}), ..\vec{z}^2), \vec{z}^1), rhs)$$

Example 11 (Depth_3 rule).
$A \vee \neg A \wedge B = A \vee B$, where A, B are boolean variables, is a non-linear Depth_3 rule, since its *lhs* matches Depth_3 pattern: $@(\vee, A, @(\wedge, @(\neg, A), B))$

Example 12 (Depth_1 rules).

- $x \wedge x = x$ is a non-linear Depth_1 rule
- $x \cdot y = y \cdot x$

3.3 LF as a Rule Definition Language

We express rewrite rules in LF-grounded theories as typed constants. The name of the constant constitutes to rule name, whereas it's type is the LF expression to be reflected into MMT and give it a computational meaning. We come back to the theory `Group` to give an explicit example of a rule declaration in the theory:

,

```
1 theory Group : LF =
2   o : type
3   op    : o → o → o # 1 ∘ 2 prec 10
4   e    : o
5   equal    : o → o → type # 1 = 2 prec 20 role Eq
6   neutralL : {x : o} e ∘ x = x role Simplify
```

Figure 6: Group Declaration in an *mmt* File

1 Theory `Group` declares a previously declared theory `LF` as it's meta-theory.

2 Declaration of group element type `o`.

3 Declaration of group operation `op` with notation $\circ$.

4 Declaration of a group element `e`. We mean this to be the neutral group element.

5 Declaration of group element equality. Note that we specify the rule `Eq` - our framework assumes that the operation with the role `Eq` infers equality judgement.

6 Constant `neutralL` of type $\{\mathtt{x}\}\ \mathtt{e} \circ \mathtt{x} = \mathtt{x}$. Again, the role `Simplify` of the constant is specified for the framework to recognize it as a simplification rule.

3.4 Reflecting Rules into MMT

Each rule `R` is parsed as a constant of the theory `T` it is declared in: `T?R`. The type of the constant `R : tp` is an **LF** expression. `tp` may have a set of variables `Γ` bound by a designated equality operation (`Eq`). The expression `tp` is reflected into MMT as the input-output of the rule `R`.

In this section we present the general idea of the algorithm behind rule implementation generation. We also illustrate the results of rule generation by going step-by-step through some examples. The technical details of the implementation are to be found in the section 4.

Theorem 13. *A Depth$_2$ (linear) rule $c : \mathcal{A}$ induces a term substitution σ_c and a rewrite of any term t matching the pattern of the rhs of the rule.*

Proof. Let t be some term of the form $t = App(@, \oplus, p_1, .., p_k, App(@, \times, s_1, .., s_l), t_1, .., t_m)$ then it matches the lhs pattern of a Depth$_2$ rule:

$$App(@, \oplus, x_1, .., x_k, App(@, \otimes, y_1, .., y_l), z_1, .., z_m)$$

which induces a substitution:

$$\sigma_c^t = [x_1/p_1, .., x_k/p_k, y_1/s_1, .., y_l/s_l, z_1/t_1, .., z_m/t_m]$$

Let $s_c : Term \to Term = \lambda_\Gamma T.\sigma_c^T(rhs_c)$[3] be the implementation of rule c. Thus the term t can be rewritten as: EdN:3

$$s_c(t) = rhs[x_i/p_i, y_i/s_i, z_i/t_i]$$

□

[3] EdNote: how to express that σ_c^t is being induced by the rule c and the input term T?

The following examples illustrate how implementations are generated via substitution.

Example 14 (Rule Application).

1) Let $t_1 = ((a \circ b)^{-1})^{-1}$, where a, b are group elements, $\circ$ - group operation and $^{-1}$ denotes inverse element.

   Let c be a Depth_2 rule of type $\mathcal{A} := (x^{-1})^{-1} = x$

   Then a substitution for term t_1 is induced $\sigma_c^{t_1} = [x/a \circ b]$ and once the rule c is applied to the term t_1:

   $s_c(t_1) = a \circ b$

2) Let $t_2 = -a \cdot (-b)$, where a, b are of type Int, multiplication and – have the conventional meaning for integers.

   Let $c : \mathcal{A}$ be as defined in the previous example and additionally $c_1 : x \cdot (-y) = -(x \cdot y)$, $c_2 : (-x) \cdot y = -(x \cdot y)$

   - when the rule c_1 is applied a substitution $\sigma_{c_1}^{t_2} = [x/-a, y/b]$ is induced and results in $s_{c_1}(t_2) = -((-a) \cdot b) = t_3$
   - when the rule c_2 is applied a substitution $\sigma_{c_2}^{t_3} = [x/a, y/b]$ is induced and results in $s_{c_2}(t_3) = -(-(a \cdot b)) = t_4$
   - when the rule c is applied a substitution $\sigma_c^{t_4} = [x/a, y/b]$ is induced and results in $s_c(t_4) = a \cdot b$

However **Theorem 13** proposes a naive algorithm that only works with explicit variables. The following vector data structure provides us with a counter-example.

Example 15 (Vectors with Implicit Variables).
Vector data structure (includes a theory for natural numbers[2])

,

```
theory Vec : ?LF =
  include ?Nat
  Vec : Nat
  Nil : Vec Zero
  cons : {n : Nat} Nat → Vec n → Vec (Succ n)
  head : {m : Nat} Vec → Nat
  HeadRule : {n : Nat} {m : Nat} {v : Vec m} head (cons n v) = n
```

[2]for theory `Nat` see 6.1.1

takes a natural number as constructor argument. A *zero*-length vector is an empty vector. A new vector can be constructed by adding a natural number to an existing n-length vector. Then *head* operator returns the first element of a vector matching the pattern *cons n v*.

The vector ADT demonstrates a case when our naive algorithm would fail, since the pattern for the rule would include implicit arguments. Namely, we would like to reflect the rule for *head* as a rewrite rule that matches an **LF** expression $App(\text{head}, App(\text{cons}, n, v))$. But the rhs of the rule declaration is of the form:

$$App(@, head, \underbrace{m}_{\text{implicit}}, App(@, cons, \underbrace{m}_{\text{implicit}}, n, v))$$

Note, that implicit argument m occurs.

To account for the implicit arguments we restate Theorem 13:

Theorem 16. *A Depth$_2$ (linear) rule $c : \mathcal{A}$ induces a term substitution σ_c on explicit arguments and a rewrite of any term t matching the pattern of the rhs of the rule.*

Proof. Let t be some term of the form

$t = App(@, \oplus, p_{1^i}, .., p_{k^i}, p_1, .., p_k, App(@, \times, s_{1^i}, .., s_{l^i}, s_1, .., s_l), t_{1^i}, .., t_{m^i}, t_1, .., t_m)$

then it matches the *lhs* pattern of a Depth$_2$ rule:

$$App(@, \oplus, \underbrace{x_{1^i}, .., x_{k^i}}_{\text{implicit}}, x_1, .., x_k, App(@, \otimes, \underbrace{y_{1^i}, .., y_{l^i}}_{\text{implicit}}, y_1, .., y_l), \underbrace{z_{1^i}, .., z_{m^i}}_{\text{implicit}}, z_1, .., z_m)$$

where implicit arguments $x_{1^i}, .., x_{k^i}, y_{1^i}, .., y_{l^i}, z_{1^i}, .., z_{m^i}$ are ignored.

The match induces a substitution for explicit arguments:

$$\sigma_c^t = [x_1/p_1, .., x_k/p_k, y_1/s_1, .., y_l/s_l, z_1/t_1, .., z_m/t_m]$$

Let $s_c : Term \to Term = \lambda_\Gamma T.\sigma_c^T(rhs_c)$[4] be the implementation of rule c. EdN:4

Thus the term t can be rewritten as:

$$s_c(t) = rhs[x_i/p_i, y_i/s_i, z_i/t_i]$$

□

[4] EdNote: how to express that σ_c^t is being induced by the rule c and the input term T?

4 Implementation

This section supplements the description of the MMT API given in section 2.2 with a rule processing overview. The only pre-requisite for rule implementations from MMT theories is a logical foundation. In our case it is logic foundation LF. It is the meta-theory of any new theories we write in our MMT rule testing library.

Rules come as MMT constants with a specific role - an optional string. In our case the designated role is *Simplify*. The keyword triggers **RoleHandler** to call the plug-in on the body of the constant.

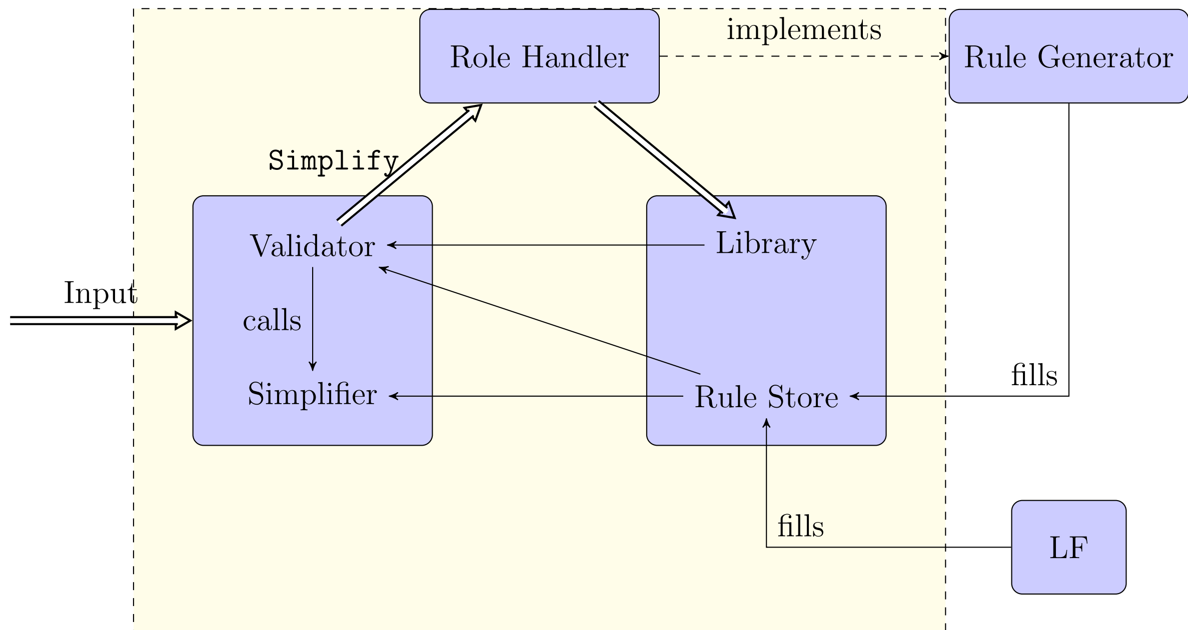

Figure 7: MMT API with Rule Generator

Rule Generator is our implementation of declarative rule reflection to MMT. It implements the rule matching algorithm described in Theory **??**.

General architecture of an MMT archive with rule implementation:

base theory - a theory that that defines basic entities rules could manipulate. For example inductive definition of natural numbers, a definition of an algebraic group etc.

rule declarations - may be contained either in the base theory or in any other theory which imports the former.

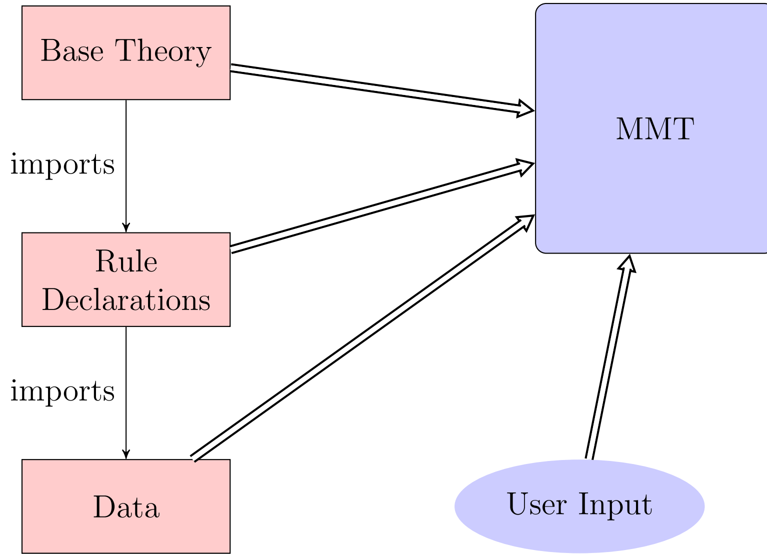

Figure 8: MMT Input Hierarchy

data - declarations which can be simplified (processed) by the previously given rules. The theory which contains data must import rule declarations for them to be in scope.

Once the theory containing rule files is read, MMT API theory validator is triggered. The validator detects designated rules and then they are handled according to the algorithm in **??** of previous section. Once a rule is reflected back to MMT as an MMT term computation, it is registered with MMT Rule Store.

The theories coming into MMT are modular, which allows multiple users to write rules for them.

5 Towards a Generalized Rule Reflection

Basing our understanding of rules on type theory (**??** we claim that rules are series of pre-requisite judgements with the last judgement being the desired conclusion. We also specify, what kind of rules exactly we intend to generate implementations for.

Recall **??** that MMT language allows declarations of constants of the form $c : A$, $c = A$ and $c : A = A'$. These declarations represent to logic judgements: typing judgement and equality judgement respectively. Our intuition of what rules should be in MMT arises from the notion of judgements in MMT. We observe that from the two kinds of judgements (typing and equality) 5 term search problems arise.

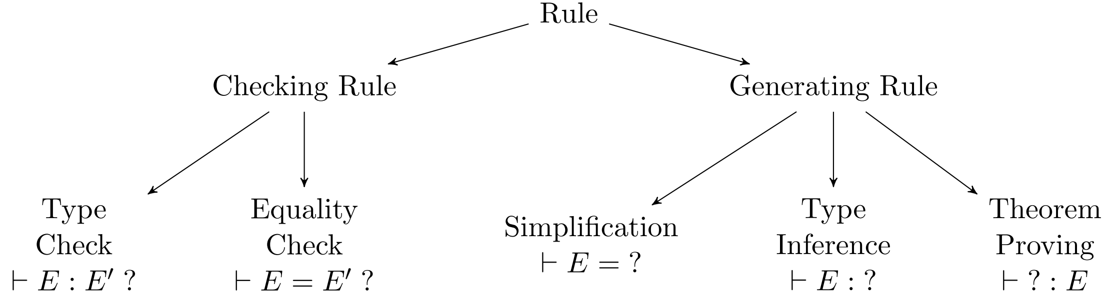

Figure 9: Rule Hierarchy induced by Judgements

- $\vdash E = E'$ – equality judgement (expression E is defined by the expression E')
- $\vdash E : E'$ – typing judgement (expression E is of type E')

If one of the terms on either side of the typing or equality symbol are absent, we denote it by "?". Such term "holes" represent goals. Whereas if "?" is appended to the judgement, it implies that the given statement needs to be proved.

- $\vdash E = E'$? – equality check. The task of this problem is check if the expression E on the right-hand side is equal (whatever equality could mean) to the expression E' on the left-hand side. Such task resonates with the unification problem.
- $\vdash E : E'$? – type check. Checking whether the expression E is of the type E' is a common task in computer science and it's applications range from assurance of program soundness to code maintenance.
- $\vdash E =$? – simplification. This problem represents a wide range of tasks ranging from expression evaluation to symbolic simplification. However, note that the solution need not be a more simple expression, it might might an alternate formulation. For instance $a^2 – b^2 = (a - b)(a + b)$ or $a \circ b = b \circ a$ are treated as simplifications.
- $\vdash E :$? – type inference. Given an expression E it's type needs to be inferred. The task of machine inference is common in programming and machine-reasoning.
- $\vdash$? $: E'$ – theorem proving. Determining the term E of the type E' essentially can be seen as finding a proof E to a proposition E'. This observation arises in Curry-Howard isomorphism – propositions can be treated as types and programs as proofs.

rule	$R := \Pi_{\Gamma_g}.J_1 \to ... \to J_n \to A$
judgements	$J := \Pi_{\Gamma_l}.A_1 \to ... \to A_m \to A$
atomic judgement	$A := t_1 = t_2 \mid t_1 : t_2$

Figure 10: Judgements

General rules-as-judgements paradigm:

A rule R may require several guards (hypotheses) J_1,...,J_n to be valid before the conclusion A can be made. Similarly every guard J may be composed of several atomic judgements A_1,...,A_m. An atomic judgement may be either an equality judgement $t_1 = t_2$, which entails that the term t_1 is equivalent to the term t_2, or a typing judgement $t_1 : t_2$, where term t_1 must be of type t_2.

For example a simplification rule arising from group neutral element axiom $a \circ e = a$ has no guards ($n = 0$) and is composed of only one atomic judgement $t_1 = t_2$, where $t_1 = a \circ e$, $t_2 = a$.

A more complicated example comes up when examining typing rules. A typical

$$\frac{F\#A => B \quad x\#A}{F@x\#B}$$

Figure 11: Function Application Type Inference in STT

type inference judgement for STT (figure ??) comes as a rule with two hypotheses $J_1 = F\#A => B$ and $J_2 = x\#A$ which are typing judgements themselves. This rule also has a global context, namely we know that A and B are types, x and F are terms, thus $\Gamma_g = \{x : tm, F : tm, A : tp, B : tp\}$.

6 Applications

In this section we cover several test libraries we have written. The libraries are MMT archives containing theory declarations. Theories are either base theories that set up ground for rules, the rule declarations or the testing declarations that can be validated by the MMT API once the rule implementations are in effect. This library combines the composable and extensible architecture of the archive, and the computational properties of rules.

We demonstrate a typical setup for base theories and their respective rule declarations in subsection 6.1. It contains a set of algebraic, logical and arithmetic rules with their respective base theories.

The use of those theories is demonstrated in subsection 6.2. MMT allows for an interactive term computation via console. We use the console interface

to demonstrate explicitly the computations possible with the run-time generated rules from subsection 6.1.

Subsection 6.3 contains more complicated use-cases of generated rules. These cases occur when simplification rules represent functions that act in type checking. Namely, sometimes the type subterms also need to be simplified in order to prove or refute the types.

6.1 Examples

In this section we introduce a small library we developed using the MMT rule infrastructure described in section 4. The library contains many rules provided as examples in section 3.1. The rules for Group theory, Boolean algebra rules accompanying a propositional logic theory and a set of arithmetic rules are just a few cases where our approach for rule generation works

We also develop slightly more complicated examples in section 6.1.4 - a case study of type theory with type products. Although the latter contains just a couple of simplification rules, they serve to demonstrate the flexibility and range of application of our approach.

6.1.1 Algebra

Our running case, the Group theory example 5, contains the following rules:

Name	LF Declaration	Semantics
neutralL	$\{x : g\}\ e \circ x == x$	$\forall x.x \circ e = x$
neutralR	$\{x : g\}\ x \circ e == x$	$\forall x.e \circ x = x$
invInv	$\{x : g\}\ x\ {-^1}\ {-^1} == x$	$\forall x.x^{-1^{-1}} = x$
invL	$\{x : g\}\ x\ {-^1} \circ x == e$	$\forall x.x^{-1} \circ x = e$
invr	$\{x : g\}\ x \circ x{-^1} == e$	$\forall x.x \circ x^{-1} = e$
invOp	$\{x : g, y : g\}\ (x \circ y){-^1} == y\ {-^1} \circ x{-^1}$	$\forall x.\forall y.(x \circ y)^{-1} = y^{-1} \circ x^{-1}$

Figure 12: Group Simplification Rules

These rules are not really useful yet, however, note that the basic group properties defined by these rules are valid for a wide variety of other algebraic constructs. Therefore it is noteworthy that rule implementations can be generated for these simple rules.

MMT libraries are modular and can define morphisms between theories. Thus the said rule implementations can be inherited by theories that include Group theory. They could also potentially be added to a theory bound by morphism, for example rules such as neutralL could become $\forall n.0 + n = n$ for naturals, were a morphism between Group and a theory defining natural numbers established.

6.1.2 Propositional Logic

We have developed a brief Propositional Logic theory in order to extend it with boolean rules.

,

```
i : type
true : i
false : i
bool_eq : i → i → type # 1 = 2 prec 0 role Eq
and : i → i → i # 1 ∧ 2 prec 5
or : i → i → i # 1 ∨ 2 prec 4
not : i → i # ¬ 1 prec 6
```

Where `i` is type of individuals, and `true` and `false` are individuals. Boolean equivalence relation is defined by `bool eq` and its notation is `=`. Boolean operators `and`, `or` and `not` have the conventional meaning, but their respective semantics is not reflected in the declarations.

The following rules extend the computational semantics of the logical operations on individuals:

Name	Declaration	Semantics
andTrueL	$\{x : i\}\ true \wedge x = x$	
orTrueR	$\{x : i\}\ x \vee true = true$	
andFlaseR	$\{x : i\}\ x \wedge false = false$	
orFalseL	$\{x : i\}\ false \vee x = x$	
absorbOr	$\{x : i, y : i\}\ x \vee (x \wedge y) = x$	
complementAndL	$\{x : i\}\ \neg x \wedge x = false$	
complementOrR	$\{x : i\}\ x \vee \neg x = true$	

Note that we chose to present only half of the rules here, the symmetric counterpart of each rule is not presented.

6.1.3 Arithmetic

Our goal is to show that our proposed method for implementation generation is sufficiently flexible to support a wide variety of known computational rules. For example we can easily describe integers and operations on integers.

We declare an LF type `Nat`. `Zero` is of type `Nat`, or as the corresponding Peano axiom states: number zero is a natural number. `succ` is the successor function for our type `Nat`. We also declare the binder for equality of natural numbers `Nat`. [5] EdN:5

[5] EdNote: `refl`?

,

```
theory Nat : http://cds.omdoc.org/urtheories?LF =
  Nat : type
  succ : Nat → Nat
  Zero : type

  ==     : Nat → Nat → type # 1 == 2 prec 3 role Eq
  refl   : {n} n == n
```

Figure 13

Rule	Declaration	Semantics	Notes
addZeroL	$\{\texttt{x} : \texttt{Nat}\}\texttt{Zero} + \texttt{x} == \texttt{x}$	$\forall x.0 + x = x$	
addZeroR	$\{\texttt{x} : \texttt{Nat}\}\texttt{x} + \texttt{Zero} == \texttt{x}$	$\forall x.x + 0 = 0$	
addLeft	$\{\texttt{x} : \texttt{Nat}, \texttt{y} : \texttt{Nat}\}\texttt{Sx} + \texttt{y} == \texttt{S}(\texttt{x} + \texttt{y})$	$\forall x.\forall y.s(x) + y = s(x + y)$	
addRight	$\{\texttt{x} : \texttt{Nat}, \texttt{y} : \texttt{Nat}\}\texttt{x} + \texttt{Sy} == \texttt{S}(\texttt{x} + \texttt{y})$	$\forall x.\forall y.x + s(y) = s(x + y)$	
timesZeroL	$\{\texttt{x} : \texttt{Nat}\}\texttt{Zero} * \texttt{x} == \texttt{Zero}$	$\forall x.0 \cdot x = 0$	
timesZeroR	$\{\texttt{x} : \texttt{Nat}\}\texttt{x} * \texttt{Zero} == \texttt{Zero}$		
multLeft	$\{\texttt{x} : \texttt{Nat}, \texttt{y} : \texttt{Nat}\}\texttt{Sx} * \texttt{y} == \texttt{y} + \texttt{x} * \texttt{y}$		
multRight	$\{\texttt{x} : \texttt{Nat}, \texttt{y} : \texttt{Nat}\}\texttt{x} * \texttt{y} == \texttt{x} + \texttt{x} * \texttt{y}$		

Figure 14: Implementations of Basic Arithmetic Rules for Natural and Integer Numbers

More importantly, we can declare accompanying rules, implementations for which get for free. The rules for integer addition and multiplication follow the Peano axioms. The rules in 14 are enough to do basic arithmetics.

,

```
theory Int : http://cds.omdoc.org/urtheories?LF =
  include ?Nat
  One : Nat = S Zero

  plus : Nat → Nat → Nat # 1 + 2 prec 5
  minus : Nat → Nat → Nat # 1 - 2 prec 5
  times : Nat → Nat → Nat # 1 * 2 prec 6

  fac : Nat → Nat# 1 ! prec 3
```

,

```
theory Fac : http://cds.omdoc.org/urtheories?LF =
  include ?Int

  timesL
  timesR

  factorial
```

For computation of factorial of some number of type `Nat` we only need to declare the factorial rule itself. The previously declared arithmetic rules for `Nat` take care of the rest. Namely, the addition and multiplication of the numbers is resolved by the UOM though multiple application of rules found applicable. Since addition and multiplication of `Nat` is already given in the imported files, it is in scope, thus UOM can apply those rules until no rules are applicable to the term. Note that we declare two rules for factorial. One rule covers the base case, the other - the inductive case. In our framework each rule represents a single case, so partial functions are also possible.

[6]

EdN:6

6.1.4 Type Theory

6.2 Interactive Computation

In this section we present interactive computation

Loading the archive with the rules effective allows the user to compute the result of a factorial function:

The console output in figure 15 contains:

[6] EdNote: comparison to Dedukti case study

,

```
scala-mmt> uomp"S S Zero !"
uom: simplifying S S Zero!
    uom: simplifying S Zero!
        uom: simplifying Zero!
        uom: simplifying Zero*(Zero!)
    uom: simplifying S Zero*(S Zero!)
        uom: simplifying Zero*(S Zero!)
    uom: simplifying Zero+S Zero
    uom: not simplifying S Zero
res2: String = S S Zero
```

Figure 15: Interactive Computation of 2!

line `1` An MMT console prefix `scala-mmt>` followed by a simplifier (`uomp`) call on argument string `"S S Zero !"`. Note that we used a customized simplifier command `uomp` which presents the term in notation. The default command `uom` presents MMT term instead.

lines `2 - 8` The simplifier prompts the user with a message that it is simplifying some subterm of the input term. Each successive application of a simplification rule prompts a corresponding message.

line `9` No further simplification rule possible.

line `10` The resulting term is printed out in notation. The actual term is of the form succ@(succ@Zero) in LF and is represented by an MMT term App(@, succ, App(@, succ, Zero)).

6.3 Computation in Validation

Immediate application of generated rule implementations arises when we consider the already existing MMT functionality - term validation. MMT has several type inference rules stored internally via direct Scala implementation. However, introducing additional rules in runtime allows us a wider range of operations. Namely, we can validated some expressions in typing and equality judgements that we were not able to validated before.

For example if we go back to our *List* library, we find a term with a complex list expression in it's type will not validate normally.

6.3.1 ADTs

6.3.2 Homotopy Type Theory

A homotopy type theory (HoTT) gives semantics to intensional type theory by interpretation in the context of homotopy theory. An encoding of HoTT in MMT done by Florian Rabe is an example case that uses our simplification rule generator.

7 Future Work

Challenges to resolve, questions to explore:

- similar depth rule strategy application in generating type inference rules
- rule generation for declarations of arbitrary depth (going beyond depth-2 rules)
- rule generation for declarations with multiple operations at the same level, for example $\neg x \wedge \neg y = x \vee y$
- rule application priorities, i.e. choosing to apply certain rules before any other
- rule implementations and theory morphisms - can implementation be sustained for a morphism?
- example cases of rule generation with foundations other than LF

8 Conclusion

References

[CDF] Wolfram demonstrations project: Propositional logic test. http://demonstrations.wolfram.com/PropositionalLogicTest/. last accessed September 18, 2013.

[CF11] Jacques Carette and William M. Farmer. Mathscheme: Project description. In *Intelligent Computer Mathematics, volume 6824 of Lecture Notes in Computer Science*, pages 287–288. Springer-Verlag, 2011.

[coq] Coq reference manual: Calculus of inductive constructions. http://coq.inria.fr/doc/Reference-Manual006.html.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL – A Theorem Proving Environment For Higher Order Logic.* Cambridge University Press, 1993.

[Koh06] Michael Kohlhase. OMDoc – *An Open Markup Format for Mathematical Documents [Version 1.2].* Number 4180 in LNAI. Springer Verlag, August 2006. URL `http://omdoc.org/pubs/omdoc1.2.pdf`.

[Mat12] Mathematica, 2012.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* Number 2283 in LNCS. Springer, 2002.

[Rab] Florian Rabe. The mmt language and system. URL `https://svn.kwarc.info/repos/MMT`.

[RS09] F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, volume LFMTP'09 of *ACM International Conference Proceeding Series*, pages 40–48. ACM Press, 2009.

[SP] Ronan Saillard and Mines Paristech. Dedukti: a universal proof checker.

[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.

[The11] The Coq Development Team. The coq proof assistant: Reference manual. Technical report, INRIA, 2011.

[wol] Computable document format. `http://www.wolfram.com/cdf/`.