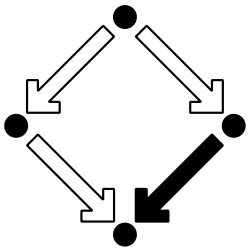
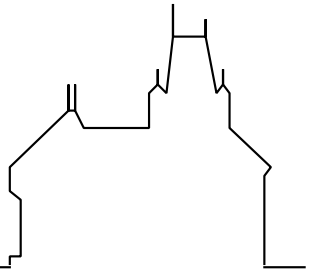


RISC-Linz
Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria, Europe



Calculemus/MKM 2007
Work in Progress

Manuel KAUERS, Manfred KERBER,
Robert MINER, Wolfgang WINDSTEIGER (Eds.)

Hagenberg, Austria
June 27–30, 2007

RISC-Linz Report Series No. 07-06

Editors: RISC-Linz Faculty
B. Buchberger, R. Hemmecke, T. Jebelean, M. Kauers, T. Kutsia, G. Landsmann, F. Lichtenberger, P. Paule, H. Rolletschek, J. Schicho, C. Schneider, W. Schreiner, W. Windsteiger, F. Winkler.

Supported by:

Preface

This collection contains the work-in-progress papers presented at two conferences, Calculemus 2007 and MKM 2007. Calculemus 2007 was the 14th in a series of conferences dedicated to the integration of computer algebra systems (CAS) and automated deduction systems (ADS). MKM 2007 was the Sixth International Conference on Mathematical Knowledge Management, an emerging interdisciplinary field of research in the intersection of mathematics, computer science, library science, and scientific publishing. Both conferences aimed to provide mechanized mathematical assistants. Regular papers of the conferences were published as Lecture Notes in Artificial Intelligence volume 4573.

Although the two conferences have separate communities and separate foci, there is a significant overlap in the interests in building mechanized mathematical assistants. For this reason it was decided to collocate the two events in 2007 for the first time, at RISC in Hagenberg, Austria. The number and quality of the submissions show that this was a good decision. While the proceedings are shared, the submission process was separate. The responsibility for acceptance/rejection rests completely with the two separate Program Committees.

By this collocation we made a contribution against the fragmentation of communities which work on different aspects of different independent branches, traditional branches (e.g., computer algebra and theorem proving), as well as newly emerging ones (on user interfaces, knowledge management, theory exploration, etc.). This will also facilitate the development of integrated mechanized mathematical assistants that will be routinely used by mathematicians, computer scientists, and engineers in their every-day business.

Table of Contents

Property inference for Maple: an application of abstract interpretation

Jacques Carette and Stephen Forrest

Computing and Software Department, McMaster University
Hamilton, Ontario, Canada
{carette,forressa}@mcmaster.ca

Abstract. We continue our investigations of what exactly is in the code base of a large, general purpose, dynamically-typed computer algebra system (Maple). In this paper, we apply and adapt formal techniques from program analysis to automatically infer various core properties of Maple code as well as of Maple values. Our main tools for this task are abstract interpretation, systems of constraints, and a very modular design for the inference engine. As per previous work, our main test case is the *entire* Maple library, from which we provide some sample results.

1 Introduction

We first set out to understand what really is in a very large computer algebra library [1]. The results were mixed: we could "infer" types (or more generally, contracts) for parts of the Maple library, and even for parts of the library which used non-standard features, but the coverage was nevertheless disappointing. The analysis contained in [1] explains why: there are eventually simply too many non-standard features present in a large code base for any kind of *ad hoc* approach to succeed.

We were aiming to infer very complex properties from very complex code. Since we cannot change the code complexity, it was natural to instead see if we could infer simple properties, especially those which were generally independent of the more advanced features of Maple [7]. The present paper explains our results: by using a very systematic design for a code analysis framework, we are able to infer simple properties of interesting pieces of code. Some of these properties are classical [9], while others are Maple-specific. In most cases, these properties can be seen as *enablers* for various code transformations, as well as enablers for full-blown type inference. Some of these properties were influenced by other work on manipulating Maple ([8, 2]) where knowledge of those properties would have increased the precision of the results.

In this current work we follow classical static program analysis fairly closely. Thus we make crucial use of Abstract Interpretation as well as Generalized Monotone Frameworks [9]. We did have to design several instantiations of such frameworks, and prove that these were indeed proper and correct instances. We also had to extend these frameworks with more general constraint systems to be able to properly encode the constraints inherent in Maple code.

In Figure 1 we illustrate some of the facts we seek to infer from code as motivation for our task. Example 1 is the sort of procedure upon which we should like to perform successful inferences. We aim to infer that `c` is an integer or string at the procedure's termination; for this we need to encode knowledge of the behavior of the Maple function `nops` ("number of operands") and of the semantics of `*`. Example 2 illustrates the fact that Maple programs sometimes exhibit significantly more polymorphism than their authors intend. We may believe that the `r := 0` assignment requires `r` to be a numeric type, but in fact it may be a sum data structure, list, expression sequence, vector, or matrix, upon which arithmetic is performed componentwise: this "hidden polymorphism" may inhibit the range of our inferences. Example 3 illustrates "bad" code: it will always give an error, since the sequence $(x, 1, p)$ automatically flattens within `map`'s argument list to produce `map(diff,x,1,p,x)` and the `diff` command cannot accept this. (The list $[x, 1, p]$ would work correctly.) We want to detect classes of such errors statically.

Example 1

```
f1 := proc(b)
  local c;
  c := "a string";
  if b then
    c := 7 * nops(b);
  end if;
  c
end proc:
```

Example 2

```
f2 := proc(n)
  local i, r;
  r := 0;
  for i to n do
    r := i*r + f(i);
  end do;
  return r
end proc:
```

Example 3

```
f3 := proc(p, x::name)
  map(diff,(x,1,p),x)
end proc:
```

Fig. 1. Examples of Maple input

Our main contributions involve: some new abstract interpretation and monotone framework instantiations, and showing that these are effective; the use of a suitable constraint language for collecting information; a completely generic implementation (common traversal routines, common constraint gathering, etc). This genericity certainly makes our analyzer very easy to extend, and does not seem to have a deleterious effect on efficiency.

The paper is structured as follows: In section 2 we introduce Abstract Interpretation, followed by section 3 where we formally define the properties we are interested in. Section 4 outlines our approach to collecting information via constraints. In section 5, we give a sample of the results we have obtained thus far. A description of the software architecture and design is in section 6, followed by our conclusions.

2 Abstract Interpretation

Abstract Interpretation [5] is a general methodology which is particularly well suited to program analyses. While the operational semantics of a language precisely describe how a particular program will transform some input value into an

output value[1], we are frequently more interested in knowing how a program induces a transformation from one property to another. We proceed to give a quick introduction to this field; the interested reader may learn more from the many papers of P. Cousot ([4, 3] being particularly relevant). Our overview has been thoroughly enlightened by the pleasant introduction [12] by Mads Rosendahl, and David Schmidt's lecture notes [13], whose (combined) approach we generally follow in this section.

Conceptually, given two interpretations $I_1[\![p]\!]$ and $I_2[\![p]\!]$ from programs, we would like to establish a relationship R between them. Generally, I_1 is the standard meaning, and I_2 is a more abstract meaning, designed to capture a particular property.

To make this more concrete, let us begin with the standard example, the *Rule of sign*. Consider a simple expression language given by the grammar

$$e ::= n \mid e + e \mid e * e$$

We want to be able to predict, whenever possible, the sign of an expression, by using only the signs of the constants in the expression. The standard interpretation is usually given as

$$E[\![e]\!] \ : \ \mathbb{Z} \qquad\qquad E[\![e_1 + e_2]\!] = E[\![e_1]\!] + E[\![e_2]\!]$$
$$E[\![n]\!] = n \qquad\qquad E[\![e_1 * e_2]\!] = E[\![e_1]\!] * E[\![e_2]\!]$$

The abstract domain we will use will allow us to differentiate between expressions which are constantly zero, positive or negative. In fact, however, we need more: this is because if we add a positive integer to a negative integer, we cannot know the sign of the result (without actually computing the result). So we also give ourselves a value to denote that all we know is the result is a 'number'.

Taking $\mathbf{Sign} = \{\mathsf{zero}, \mathsf{pos}, \mathsf{neg}, \mathsf{num}\}$, we can define an "abstract" version of addition and multiplication on **Sign**:

$\oplus : \mathbf{Sign} \times \mathbf{Sign} \to \mathbf{Sign}$

$\oplus$	neg	zero	pos	num
neg	neg	neg	num	num
zero	neg	zero	pos	num
pos	num	pos	pos	num
num	num	num	num	num

$\otimes : \mathbf{Sign} \times \mathbf{Sign} \to \mathbf{Sign}$

$\otimes$	neg	zero	pos	num
neg	pos	zero	neg	num
zero	zero	zero	zero	zero
pos	neg	zero	pos	num
num	num	zero	num	num

Using these operators, we can define the abstract evaluation function for expressions as:

$$A[\![e]\!] \ : \ \mathbf{Sign} \qquad\qquad E[\![e_1 + e_2]\!] = A[\![e_1]\!] \oplus E[\![e_2]\!]$$
$$A[\![n]\!] = \mathsf{sign}(n) \qquad\qquad E[\![e_1 * e_2]\!] = A[\![e_1]\!] \otimes E[\![e_2]\!]$$

where $\mathsf{sign}(\mathrm{x}) = \mathbf{if}\ x > 0\ \mathbf{then}\ \mathsf{pos}\ \ \mathbf{else\ if}\ x < 0\ \mathbf{then}\ \mathsf{neg}\ \mathbf{else}\ \mathsf{zero}$.

[1] where these values can, for imperative programs, consist of *state*

Formally, we can describe the relation between these two operations as follows (and this is typical):

$$\gamma : \mathbf{Sign} \to \mathcal{P}(\mathbb{Z}) \setminus \emptyset \qquad \alpha : \mathcal{P}(\mathbb{Z}) \setminus \emptyset \to \mathbf{Sign}$$

$$\begin{aligned} \gamma(\mathsf{neg}) &= \{x \mid x < 0\} \\ \gamma(\mathsf{zero}) &= \{0\} \\ \gamma(\mathsf{pos}) &= \{x \mid x > 0\} \\ \gamma(\mathsf{num}) &= \mathbb{Z} \end{aligned} \qquad \alpha(X) = \begin{cases} \mathsf{neg} & X \subseteq \{x \mid x < 0\} \\ \mathsf{zero} & X = \{0\} \\ \mathsf{pos} & X \subseteq \{x \mid x > 0\} \\ \mathsf{num} & \text{otherwise} \end{cases}$$

The (obvious) relation between γ and α is

$$\forall s \in \mathbf{Sign}.\alpha(\gamma(s)) = s \text{ and } \forall X \in \mathcal{P}(\mathbb{Z}) \setminus \emptyset.X \subseteq \gamma(\alpha(X)).$$

γ is called a concretization function, while α is called an abstraction function. These functions allow a much simpler definition of the operations on signs:

$$\begin{aligned} s_1 \oplus s_2 &= \alpha(\{x_1 + x_2 \mid x_1 \in \gamma(s_1) \sqcap x_2 \in \gamma(s_2)\}) \\ s_1 \otimes s_2 &= \alpha(\{x_1 * x_2 \mid x_1 \in \gamma(s_1) \sqcap x_2 \in \gamma(s_2)\}) \end{aligned}$$

From this we get the very important relationship between the two interpretations:

$$\forall e.\{E[\![e]\!]\} \subseteq \ \gamma(A[\![e]\!])$$

In other words, we can safely say that the abstract domain provides us with a *correct* approximation to the behaviour in the concrete domain. This relationship is often called a *safety* or *soundness* condition. So while a computation over an abstract domain may not give us very useful information (think of the case where the answer is num), it will never be incorrect, in the sense that the true answer will always be contained in what is returned. More generally we have the following setup:

Definition 1 *Let $\langle C, \sqsubseteq\rangle$ and $\langle A, \sqsubseteq\rangle$ be complete lattices, and let $\alpha : C \to A$, $\gamma : A \to C$ be monotonic and ω-continous functions. If $\forall c.c \sqsubseteq_C \gamma(\alpha(c))$ and $\forall a.\alpha(\gamma(a)) \sqsubseteq_A a$, we say that we have a* Galois connection. *If we actually have that $\forall a.\alpha(\gamma(a)) = a$, we say that we have a* Galois insertion.

The reader is urged to read [6] for a complete mathematical treatment of lattices and Galois connections. The main property of interest is that α and γ fully determine each other. Thus it suffices to give a definition of $\gamma : A \to C$; in other words, we want to name particular subsets of C which reflect a property of interest. More precisely, given γ, we can mechanically compute α via $\alpha(c) = \sqcap\{a \mid c \sqsubseteq_C \gamma(a)\}$, where $\sqcap$ is the meet of A.

Given this, we will want to synthesize abstract operations in A to reflect those of C; in other words for a continuous lattice function $f : C \to C$ we are interested in $\tilde{f} : A \to A$ via $\tilde{f} = \alpha \circ f \circ \gamma$. Unfortunately, this is frequently too much to hope for, as this can easily be uncomputable. However, this is still the correct goal:

Definition 2 *For a Galois Connection (as above), and functions $f : C \to C$ and $g : A \to A$, g is a* sound approximation *of f if and only if*

$$\forall c.\alpha(f(c)) \sqsubseteq_A g(\alpha(c))$$

or equivalently

$$\forall a.f(\gamma(a)) \sqsubseteq_C \gamma(g(a)).$$

Then we have that (using the same language as above)

Proposition 1 *g is a sound approximation of f if and only if $g \sqsubseteq_{A\to A} \alpha \circ f \circ \gamma$.*

How do we relate this to properties of programs? To each program transition from point p_i to p_j, we can associate a *transfer function* $f_{ij} : C \to C$, and also an abstract version $\tilde{f_{ij}} : A \to A$. This defines a computation step as a transition from a pair (p_i, s) of a program point and a state, to $(p_j, f_{ij}(s))$ a new program point and a new (computed) state. In general, we are interested in *execution traces*, which are (possibly infinite) sequences of such transitions. We naturally restrict execution traces to feasible, non-trivial sequences. We always restrict ourselves to *monotone* transfer functions, i.e. such that

$$l_1 \sqsubseteq l_2 \implies f(l_1) \sqsubseteq f(l_2)$$

which essentially means that we never lose any information by approximating. This is not as simple as it sounds: features like *uneval quotes*, if treated naïvely, could introduce non-monotonic functions.

Note that compared to some analyses done via abstract interpretation, our domains will be relatively simple (see [11] for a complex analysis).

3 Properties and their domains

We are interested in inferring various (static) properties from code. While we would prefer to work only with decision procedures, this appears to be asking for too much. Since we have put ourselves in an abstract interpretation framework, it is natural to look at properties which can be *approximated* via *complete lattices*. As it turns out, these requirements are easy to satisfy in various ways.

On the other hand, some of these lattices do not satisfy the Ascending Chain Condition, which requires some care to ensure termination.

3.1 The properties

Surface type. The most obvious property of a *value* is its type. As a first approximation, we would at least like to know what *surface type* a value could have: in Maple parlance, given a value `v`, what are the possible values for `op(0,v)`? More specifically, given the set IK of all kinds of *inert forms* which correspond to Maple values, we use the complete lattice $L = \langle \mathcal{P}(IK), \subseteq \rangle$ as our framework.

Then each Maple operation induces a natural transfer function $f : L \to L$. It is straightforward to define abstraction α and concretization γ functions between the complete lattice $\langle \mathcal{P}(V), \subseteq \rangle$ of sets of Maple values (V) and L. It is nevertheless important to note that f is still an approximation: if we see a piece of code which does `a := l[1]`, even if we knew that $\alpha(l) = \texttt{LIST}$, the best we can do is $\alpha(a) \subseteq E$, where $E = \mathcal{P}(IK) \setminus \{\texttt{EXPSEQ}\}$.

Expression sequence length. This is really two inferences in one: to find whether the value is a potential expression sequence (expseq), and if so, what length it may be. From Maple's semantics, we know that they behave quite differently in many contexts than other objects, so it is important to know whether a given quantity is an expression sequence. An expression sequence is a Maple data structure which is essentially a self-flattening list. Any object created as an expression sequence (e.g. the result of a call to `op`) which has a length of 1 is automatically evaluated to its first (and only) element. That is, an object whose only potential length as an expression sequence is 1 is not an expression sequence. The natural lattice for this is $\mathbb{I}(\mathbb{N})$ (the set of intervals with natural number endpoints) with $\subseteq$ given by containment. The abstraction function maps all non-expseq Maple values to the degenerate interval $[1 \ldots 1]$, and expseq values to (an enclosure for) its length. Note that `NULL` (the empty expression sequence) maps to $[0 \ldots 0]$, and that unknown expression sequences map to $[0 \ldots \infty]$.

Variable dependence: Given a value, does it "depend" on a symbol (viewed as a mathematical variable)? The definition of 'depends' here is the same as the Maple command of that name. In other words, we want to know the complete list of symbols whose value can affect the value of the current variable. Note that this can sometimes be huge (given a symbolic input), but also empty (when a variable contains a static value with no embedded symbols). The natural lattice is the powerset of all currently known (to the system) symbols, along with an extra $\top$ to capture dynamically created symbols, with set containement ordering. Note that this comes in different flavours, depending on whether we treat a globally assigned name as a symbol or as a normal value.

Number of variable reads: In other words, for each local variable in a procedure, can we tell the number of times it will be read? The natural lattice is $L = V \to \mathbb{I}(\mathbb{N})$ with V the set of local variables of a procedure. If $s, t \in L$, then $s \sqcup t$ is defined component-wise as $\lambda v.[\max s_l(v), t_l(v), s_r(v) + t_r(v)]$ where $s(v) = [s_l(v), s_r(v)], t(v) = [t_l(v), t_r(v)]$.

Number of variable writes: A natural (semantic) dual to the number of reads, but operationally independent.

Reaching Definition: This is a classical analysis [9] which captures, at every program point, what assignments may have been been made and not overwritten. As in [9], the lattice here is $\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}^?_\star)$, ordered by set inclusion. Here $\mathbf{Var}_\star$ is finite set of variables which occur in the program, and $\mathbf{Lab}^?_\star$ is the finite set of program labels augmented by the symbol ?. Note that unlike $\mathbb{I}(\mathbb{N})$ this lattice satisfies the Ascending Chain Condition (because it is finite).

Summarizing, we will infer the following property of values (according to the definitions above): its surface type, its expression sequence length, and its variable dependencies. Note that, given a labelled program, we can speak of

values at a program point, by which we mean the value of one (or more) state variable(s) at that program point; of those values, we are interested in similar properties. For a program variable, we will work with the number of times it is read or written to. And for a program point, which assignments may have been made and not overwritten.

For the purposes of increased precision, these analyses are not performed in isolation. What is actually done is that a Reaching Definition analysis is first performed, and then the other analyses build on this result. Later, we should look at taking (reduced) tensor products of the analyses ([9] p. 254-256), although it is only clear how to do this for finite lattices.

3.2 Idiosyncrasies of Maple

Many of the analyses we wish to attempt are complicated by the particular semantics of Maple. Some of these, such as untypedness and the potential for an arbitary procedure to alter global state, are shared with many other programming languages. Others are specific to a CAS or to Maple alone. Following is a list of some key features.

1. **Symbols**: As Maple is a CAS, every variable (aside from parameters) which does not have an assigned value may be used as a symbol, and passed around as any other value. Should the variable later be assigned, any previous reference to it as a symbol will evaluate to its present value.
2. **Functions which return unevaluated**: Just as variables may be values or symbols, function calls may or may not choose to evaluate. Certain of Maple's built-in functions, sch as `gcd`, will return the function invocation unevaluated when presented with symbolic input.
3. **Side effects**: Any function invocation may affect global state, so one cannot assume state remains constant when evaluating an expression.

3.3 A formalization

Here we will give the formalization for the Galois connection associated to the *expression sequence length* property inference. The next section will complete the picture by giving the associated constraints.

The source lattice in this case is $\langle \mathbb{P}(\mathbf{Val}), \subseteq \rangle$ where **Val** represents the set of all possible Maple values. The target lattice, as mentionned above, is $\langle \mathbb{I}(\mathbb{N}), \subseteq \rangle$. The Galois connection in this case is the one given by the *representation function* $\beta : \mathbf{Val} \to \mathbb{I}(\mathbb{N})$ (see Chapter 4 of [9]). Explicitly, for $V \in \mathbb{P}(\mathbf{Val})$, $\alpha(V) = \bigsqcup\{\beta(b) \mid v \in V\}$, and $\gamma(l) = \{v \in \mathbf{Var} \mid \beta(v) \sqsubset l\}$. But this is completely trivial! For any *value* v which is neither `NULL` nor is an expression sequence, then $\beta(v) = 1..1$. Otherwise $\beta(\texttt{NULL}) = 0..0$ and $\beta(e) = \texttt{nops}([e])$ for e an expression sequence. What is much more interesting is, what is the monotone transfer function induced by β?

In other words, for all the expression constructors and all the statements of the language, what is the induced function on $\mathbb{I}(\mathbb{N})$? We want to know a

safe approximation to $\tilde{f} = \alpha \circ f \circ \gamma$. For all constructors c whose surface type is in {INTPOS, INTNEG, RATIONAL, COMPLEX, FLOAT, HFLOAT, STRING, EQUATION, INEQUAT, LESSEQ, LESSTHAN, DCOLON, RANGE, EXACTSERIES, HFARRAY, MODULE, PROC, SDPOLY, SERIES, SET, LIST, TABLE, ARRAY, VECTOR_COLUMN, VECTOR_ROW, VECTOR, NAME, MODDEF, NARGS}, $\tilde{c} = 1..1$, with the exception of the special name NULL which is 0..0. For those in {SUM, PROD, POWER, TABLEREF, MEMBER, EXPSEQ, ASSIGNEDLOCALNAME, ASSIGNEDNAME}, the best that can be said a priori is $0..\infty$. Some of these are expected (for example, an ASSIGNEDNAME can evaluate to an expression sequence of any length), but others are plain strange Maple-isms:

```
> (1,2) + (3,4);
                              4,6
```

But we can do better than that. Figure 3.3 shows a precise definition of the transfer function for SUM, EXPSEQ, and PROD. In the table for SUM, we implicitly assume that $a \leq b$ and $a..b \neq 1..1$; also, since adding two expression sequences of different lengths (other than the 1..1 case) results in an error [in other words, not-a-value], this case is not included in the table. In the table for PROD, we further assume that $a \geq 1, c \geq 1$, as well as $a..b \neq c..d$.

SUM	1..1	$a..b$
1..1	1..1	1..1
a..b	1..1	$a..b$

EXPSEQ$(a..b, c..d) = (a+c)..(b+d)$

PROD	1..1	$a..b$	$c..d$
1..1	1..1	$a..b$	$c..d$
a..b	$a..b$	1..1	1..1
c..d	$c..d$	1..1	1..1

Fig. 2. Some transfer functions associated to expression sequence length

Of course, statements and other language features that are only present inside procedures induce transfer functions too. Some are again quite simple: we know that a parameter (a PARAM) will always be 1..1. In all other cases, the transfer function associated to the statements of the language is quite simple: whenever it is defined, it is the identity. On the other hand, the transfer functions associated to many of the builtin functions (like map, op, type and so on) are very complex. We currently have chosen to take a pessimistic approach and always assume the worst situation. This is mostly a stop-gap measure to enable us to get results, and we plan on rectifying this in the future.

While it would have been best to obtain all transfer functions from a formal operational semantics for Maple, no such semantics exists (outside of the actual closed-source, proprietary implementation). We obtained the above by *expanding* the defining equation $\tilde{f} = \alpha \circ f \circ \gamma$, for each property and each f of interest, and the breaking down the results into a series of cases to examine. We then ran a series of experiments to obtain the actual results. We have to admit that, even though the authors have (together) more than 30 years' experience with Maple, several of the results (including some in figure 3.3) surprised us.

3.4 Applications

We chose those few simple analyses because they are foundational: they have many applications, and very many of the properties of interest of Maple code can most easily be derived from those analyses.

For example, if we can tell that a variable will never be read, then as long as the computation that produces that value has no (external) side-effects, then that computation can be removed[2]. Similarly, if it is only read once, then the computation which produces the value can be inlined at its point of use. Otherwise, no optimizations are safe. If we can tell that a local variable is never written to, then we can conclude that it is used as a *symbol*, a sure sign that some *symbolic* computations are being done (as opposed to numeric or other more pedestrian computations).

4 Constraints and Constraint Solving

If we took a strict abstract interpretation plus Monotone Framework approach [9], we would get rather disappointing results. This is because both forward-propagation and backward-propagation algorithms can be quite approximative in their results.

This is why we have moved to a general constraint-based approach. Unlike a Monotone Framework approach, for any given analysis we generate both forward and backward constraints. More precisely, consider the following code:

```
proc(a) local b;
    b := op(a);
    if b>1 then 1 else -1 end if;
end proc;
```

If we consider the expression sequence length analysis of the previous section, the best we could derive from the first statement is that b has length $\subseteq [0 \dots \infty)$. But from the $b > 1$ in a boolean context and our assumption that the code in its present state executes correctly, we can deduce that b must have length (exactly) 1 (encoded as $[1 \dots 1]$). In other words, for this code to be meaningful we have not only $b \subseteq [1 \dots 1]$ but also $[1 \dots 1] \subseteq b$.

More formally, given a complete lattice $L = (D, \sqcap, \sqcup, \sqsubset, =)$, we have the basic elements of a constraint language which consists of all constants and operators of L along with a (finite) set of variables from a (disjoint) set $\mathcal{V}$. The (basic) constraint language then consists of syntactically valid *formulas* using those basic elements, as well as the logical operator $\wedge$ (conjunction). A *solution* of such a constraint is a variable assignment which satisfies the formula.

For some lattices L, for example $\mathbb{I}(\mathbb{N})$, we also have and use the monoidal structure (here given by $\oplus$ and 0..0). This structure also induces a scalar (i.e. $\mathbb{N}$) multiplication, which we also use. In other words, we have added both $\oplus$ and a scalar $*$ to the constraint language when $L = \mathbb{I}(\mathbb{N})$.

A keen reader might have noted one discrepancy: in the language of constraints that we have just described, it is not possible to express the transfer

[2] in the sense that the resulting procedure p' will be such that $p \sqsubset p'$, for the natural order on functions. Such a transformation may cause some paths to terminate which previously did not – we consider this to desirable.

function (on $\mathbb{I}(\mathbb{N})$) induced by `SUM`! As this is indeed so, we have added a *constraint combinator* to the language of constraints. This takes the form $\mathcal{C}(\texttt{op})$ for any (named) function $\texttt{op} : L \to L$. In particular, we can thus use the transfer function induced by `SUM` and `PROD` in our constraint language. This also includes the expression-sequence constructor `,` (comma).

One feature of our approach beyond that of classical abstract interpretation is the addition of recurrence equations. When expressed in terms of our chosen properties, many loops and other control structures naturally induce recurrences, often very trivial ones. Consider the following:

```
fact := proc(a) local s;
    s := 1;
    for i from 1 to n do s := n*s; end if;
    return(s);
end proc;
```

At the program point corresponding to the assignment to `s` within the loop, a classical Reaching Definitions approach will always give two possibilities for the preceding assignment: the initial assignment or a previous loop iteration at the same program point, which complicates the analysis. One means of dealing with this self-dependency is to regard the problem as a recurrence over `s`.

Given a loop at program point ℓ, we introduce symbols $\mathrm{LIV}(\ell)$, $\mathrm{LFV}(\ell)$ into our constraint language to represent, respectively, the state at the start of the ith iteration and the state upon loop termination. At the program point mentioned earlier, there is now only one possibility for the preceding assignment: the symbolic quantity $\mathrm{LIV}(\ell)$.

At this point, we have to admit that we do not have a complete algorithm for the solution of all the constraint systems described. What we have does appear to work rather well, in that it terminates (even for large complex codes), and returns sensible answers. It works via a combination of successive passes of propagation of equalities, simplification of constraints, and least-fixed-point iteration. We are confident that we can prove that what we have implemented terminates and returns a proper solution of the constraint system.

5 Results

We wish to demonstrate the results of our analyses on various inputs. It is helpful to begin with some concrete examples for which the analysis can be replicated by the reader. Consider the following Maple procedure:

```
IsPrime := proc(n::integer) local S, result;
   S := numtheory:-factorset(n);
   if nops(S) > 1 then
       result := (false, S);
   else
       result := true;
   end if;
   return(result);
end proc:
```

IsPrime is an combined primality tester and factorizer. It factors its input n, then returns a boolean result which indicates whether n is prime. If it is composite, the prime factors are also returned.

This small example demonstrates the results of two of our analyses. In the Expression Sequence length analysis, we are able to conclude, even in the absence of any special knowledge or analysis of `numtheory:-factorset`, that `S` must be an expression because it is used in a call to the kernel function `nops` ("number of operands").

Combined with the fact that `true` and `false` are known to be expressions, we can estimate the size of `result` as $[2 \ldots 2]$ when the if-clause is satisfied and $[1 \ldots 1]$ otherwise. Upon unifying the two branches, our estimate for `result` becomes $[1 \ldots 2]$. For the Surface Type Analysis, we are able to estimate the `result` as {NAME,EXPSEQ}.

Our results can also be used for static inference of programming errors. We assume that the code, as written, reflects the programmers' intent. In the presence of a programming error which is captured by one of our properties, the resulting constraint system will have trivial solutions or no solutions at all.

For an illustration of this, consider the following example. The procedure `faulty` is bound to fail, as the arguments to `union` must be sets or unassigned names, not integers. As Maple is untyped, this problem will not be caught until runtime.

```
faulty := proc(c) local d, S;
    d := 1; S := {3,4,5};
    S union d;
end proc:
```

However, our Surface Type analysis can detect this: the two earlier assignments impose the constraints $X_1 \subseteq \{\texttt{INTPOS}\}$ and $X_2 \subseteq \{\texttt{SET}\}$, while `union` imposes on its arguments the constraints that $X_3, X_4 \subseteq \{\texttt{SET}\} \cup \mathcal{T}_{\text{name}}$. [3] No assignments to `d` or `S` could have occurred in the interim, we also have the constraints $X_1 = X_4$ and $X_2 = X_3$. The resulting solution contains $X_1 = \emptyset$, which demonstrates that this code will always trigger an error.

```
grows := proc(c)
    x := 2, 3, 4, 5;
    for y from 1 to 10 do
        x := x, y;
    end do;
    return(x);
end proc:
```

Here, we are able to express the relationship between the starting state, intermediate state, and final state of the for loop as a recurrence equation over the domain of the ExprSeqLength property. In the end we are able to conclude that the length of y is $[4 \ldots 4] + \mathrm{NL}(\ell_1) \cdot [1 \ldots 1]$, where $\mathrm{NL}(\ell_1)$ signifies the number of steps of the loop. Another analysis may later supply this fact.

[3] Here $\mathcal{T}_{\text{name}}$ denotes the set of tags corresponding to names, such as `NAME` and `LOCAL`; the full list is too lengthy to provide, but it does *not* contain `INTPOS`.

Results from a test library: We have run our tools against a private collection of Maple functions. This collection is chosen more for the variety of functions present within than a representative example of a working Maple library. Therefore, we focus on the results of our analyses on specific functions present within the database, rather than on summary statistics as a whole.

```
looptest := proc( n :: posint ) :: integer;
  local s :: integer, i :: integer, T :: table, flag :: true;
  ( s, i, flag ) := ( 0, 1, false );
  T := table();
  while i^2 < n do
     s := i + s;
     if flag then T[i] := s; end if;
     if type( s, 'even' ) then flag := true; break; end if;
     i := 1 + i
  end do;
  while type( i, 'posint' ) do
     if assigned(T[i]) then T[i] := T[i] - s; end if;
     if type( s, 'odd' ) then s := s - i^2 end if;
     i := i - 1
  end do;
  (s, T)
end proc:
```

This rather formidable procedure, while not doing anything particularly useful, is certainly complex. It contains two successive conditional loops which march in opposite directions, and both of which populating the table `T` along the way.

Here our analysis recognizes the fact that even though `flag` is written within the body of the first while loop, this write event cannot reach the if-condition on the preceding line because the write event is immediately followed by a `break` statement. We are also able to conclude that `s` is always an integer: though this is easy to see, given that all the write events to `s` are operations upon integer quantities.

Results from the Maple library: We present (in figure 3) some results from applying our tools to the Maple 10 standard library itself. This will serve as a useful glimpse of how our tools behave on an authentic, working codebase. Though our analysis focuses on absolutely all subexpressions within a procedure, here we focus on deriving useful information about a procedure's local variables from their context.

Expression Sequence Length	Procedures
Local with estimate $\neq [0 \dots \infty]$	862
Local with finite upper bound	593
Local with estimate $[1 \dots \infty]$	374
Local with estimate $[0 \dots 1]$	43
Solvable loop recurrences	127
Total analyzed	1276

Surface Type	Procedures
Local type is $\subsetneq \mathcal{T}_{\text{Expression}}$	827
Local w/ fully-inferred type	721
Local whose value is a posint	342
Local whose value is a list	176
Local whose value is a set	56
Solvable loop recurrences	267
Total analyzed	1330

Fig. 3. Results for analyses on Maple library source

For each analysis we sampled approximately 1300 procedures from the Maple standard library, each of which contained at least one local variable. We are particularly interested in boundary cases ($\top$, $\bot$ in our lattice, or singletons). For the Expression Sequence analysis, we obtained nontrivial results for at least one local variable in 862 of 1276 procedures; for 593, we can provide a finite bound $[a \dots b]$. For 609 locals, we have both a program point where its size is fully inferred ($[1 \dots 1]$) and another where nothing is known; an explanation for this apparent discrepancy is that locals may be assigned multiple times in different contexts. In the Surface Type analysis, we have nontrivial results for 827 of 1330 procedures; 721 have a local whose type is fully inferred.

6 Implementation

As we knew that we would be implementing many analyses, now and later, it was required that the design and implementation be as generic as possible. Because of Maple's excellent introspection facilities, but despite it being dynamically typed, we wrote the analysis in Maple itself.

This led us to design a generic abstract syntax tree (AST) traverser parametrized by whatever information gathering phase we wanted. In Object-Oriented terms, we could describe our main architecture as a combination of a Visitor pattern and a Decorator pattern. To a Haskell programmer, we would describe the architecture as a combination of a State Monad with a generic map (gmap). The data gathered are constraints expressed over a particular lattice (with an established abstract interpretation).

There are several reasons for using a constraint system as we have described in section 4: modularity, genericity, clarity and expressivity. We can completely decouple the constraint generation stage from the constraint solving stage (modularity), as is routinely done in modern type inference engines. All our analyses have the same structure, and share most of their code (genericity). Because of this generic structure, the constraints associated to each syntactic structure and each builtin function are very easy to see and understand. Furthermore, the rich language of constraints, built over a simple and well-understood mathematical theory (lattices, monoidal structures), provides an expressive language without leading too quickly into uncomputable or unsolvable systems.

For all properties, the constraint language generally consists of our chosen lattice, with its base type and lattice operations. These are extended with a set of symbols S representing unknown values in T, and a set of constraint transformers CT: these may be viewed as functions $T^* \to T$.

In general, our approach has three stages:

1. **Constraint assignment**: We traverse the AST: with each code fragment, we record constraints it imposes on itself and its subcomponents. For example, conditionals and while loops constrain their condition to be $\subset \mathcal{T}_{\text{bool}}$.
2. **Constraint propagation**: We traverse the AST again, propagating attached constraints upwards. Constraints arising from subcomponents are

inserted into a larger constraint system as appropriate to reflect the control flow. In some cases, this consists simply taking conjunction of all constraints arising from subcomponents.

3. **Constraint solving**: The solution method generally depends on the property, particularly as the constraint language itself changes depending on the property at hand. On the other hand, as we implement more solvers, we are seeing patterns emerge, which we aim to eventually take advantage of.
   In general, we proceed with a series of successive approximations. We first determine which type variables we seek to approximate: often, at a particular stage we will desire to find approximations for certain classes of symbols but leave others as symbols, untouched. (An example where symbols must be retained is with the symbols used in formulating recurrences.)
   We then step through all variables, incrementally refining our approximation for each variable based on its relations with other quantities. We are done when no better approximation is possible.

7 Conclusion

This work-in-progress shows that it is possible to apply techniques from Program Analysis to infer various simple properties from Maple programs, even rather complex programs like the Maple library. Our current techniques appear to scale reasonably well too.

One of the outcomes we expect from this work is a better-mint-than-mint[4]. As shown by some of our examples, we can already detect problematic code which mint would not flag with any warnings.

Aside from its genericity, one significant advantage of the constraint approach and the abstract interpretation framework is that analyses of different properties may be combined to refine the results of the first. For instance, if a variable instance was proven to be of size $[1 \dots 1]$ by our Expression Sequence analysis, the type tag `EXPSEQ` could be safely removed from its Surface Type results. We have yet to combine our analyses in this manner on a large scale, though this is a goal for future experimentation.

References

1. J. Carette and S. Forrest. Mining Maple code for contracts. In Ranise and Bigatti [10].
2. J. Carette and M. Kucera. Partial Evaluation for Maple. In *ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation*, 2007.
3. P. Cousot. Types as abstract interpretations, invited paper. In *Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–331, Paris, France, January 1997. ACM Press, New York, NY.

[4] mint is Maple's analogue of *lint*, the ancient tool to find flaws in C code, back when old compilers did not have many built-in warnings.

4. P. Cousot and R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form, invited paper. In P. Wolper, editor, *Proceedings of the Seventh International Conference on Computer Aided Verification, CAV '95*, pages 293–308, Liège, Belgium, Lecture Notes in Computer Science 939, 3–5 July 1995. Springer-Verlag, Berlin, Germany.
5. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
6. Brian A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.
7. P. DeMarco, K. Geddes, K. M. Heal, G. Labahn, J. McCarron, M. B. Monagan, and S. M. Vorkoetter. *Maple 10 Advanced Programming Guide*. Maplesoft, 2005.
8. M. Kucera and J. Carette. Partial evaluation and residual theorems in computer algebra. In Ranise and Bigatti [10].
9. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
10. Silvio Ranise and Anna Bigatti, editors. *Proceedings of Calculemus 2006*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.
11. Enric Rodríguez-Carbonell and Deepak Kapur. An abstract interpretation approach for automatic generation of polynomial invariants. In Roberto Giacobazzi, editor, *SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 280–295. Springer, 2004.
12. Mads Rosendahl. Introduction to abstract interpretation. http://akira.ruc.dk/ madsr/webpub/absint.pdf.
13. David Schmidt. Abstract interpretation and static analysis. Lectures at the Winter School on Semantics and Applications, WSSA'03, Montevideo, Uruguay, July 2003.

Towards Practical Reflection for Formal Mathematics

Martin Giese[1] and Bruno Buchberger[2]

[1] RICAM, Austrian Academy of Sciences,
Altenbergerstr. 69, A-4040 Linz, Austria
martin.giese@oeaw.ac.at
[2] RISC, Johannes Kepler University,
A-4232 Schloß Hagenberg, Austria
bruno.buchberger@risc.uni-linz.ac.at

Abstract. We describe a design for a system for mathematical theory exploration that can be extended by implementing new reasoners using the logical input language of the system. Such new reasoners can be applied like the built-in reasoners, and it is possible to reason about them, e.g. proving their soundness, within the system. This is achieved in a practical and attractive way by adding reflection, i.e. a representation mechanism for terms and formulae, to the system's logical language, and some knowledge about these entities to the system's basic reasoners. The approach has been evaluated using a prototypical implementation called Mini-Tma. It will be incorporated into the Theorema system.

1 Introduction

Mathematical theory exploration consists not only of inventing axioms and proving theorems. Amongst other activities, it also includes the discovery of algorithmic ways of computing solutions to certain problems, and reasoning about such algorithms, e.g. to verify their correctness. What is rarely recognized is that it also includes the discovery and validation of useful techniques for proving theorems within a particular mathematical domain. In some cases, these reasoning techniques might even be algorithmic, making it possible to implement and verify a specialized theorem prover for that domain.

While various systems for automated theorem proving have been constructed over the past years, some of them specially for mathematics, and some of them quite powerful, they essentially treat theorem proving methods as a built-in part of the services supplied by a system, in general allowing users only to state axioms and theorems, and then to construct proofs for the theorems, interactively or automatically. An extension and adaptation of the theorem proving capabilities themselves, to incorporate knowledge about appropriate reasoning techniques in a given domain, is only possible by stepping back from the theorem proving activity, and modifying the theorem proving software itself, programming in whatever language that system happens to be written.

We consider this to be limiting in two respects:

– To perform this task, which should be an integral part of the exploration process, the user needs to switch to a different language and a radically different way of interacting with the system. Usually it will also require an inordinate amount of insight into the architecture of the system.
– The theorem proving procedures programmed in this way cannot be made the object of the mathematical studies inside the system: e.g., there is no simple way to prove the soundness of a newly written reasoner within the system. It's part of the system's code, but it's not available as part of the system's knowledge.

Following a proposal of Buchberger [5, 6], and as part of an ongoing effort to redesign and reimplement the Theorema system [7], we will extend that system's capabilities in such a way that the definition of and the reasoning about new theorem proving methods is possible seamlessly through the same user interface as the more conventional tasks of mathematical theory exploration.

In this paper, we describe our approach as it has been implemented by the first author in a prototype called Mini-Tma, a Mathematica [18] program which does not share any of the code of the current Theorema implementation. Essentially the same approach will be followed in the upcoming new implementation of Theorema.

The second author's contributions are the identification and formulation of the problem addressed in this paper and the recognition of its importance for mathematical theory exploration [6], as well as a first illustrating example [5], a simplified version of which will be used in this paper. The first author has worked out the technical details and produced the implementation of Mini-Tma.

In Sect. 2, we introduce the required concepts on the level of the system's logical language. Sect. 3 shows how this language can be used to describe new reasoners, and how they can be applied. Sect. 4 illustrates how the system can be used to reason about the logic itself. These techniques are combined in Sect. 5 to reason about reasoners. We briefly disuss some foundational issues in Sect. 6. Related work is reviewed in Sect. 7, and Sect. 8 concludes the paper.

2 The Framework

To reason about the syntactic (terms, formulae, proofs,...) and semantic (models, validity...) concepts that constitute a logic, it is in principle sufficient to axiomatize these concepts, which is possible in any logic that permits e.g. inductive data type definitions, and reasoning about them. This holds also if the formalized logic is the same as the logic it is being formalized in, which is the case that interests us here.

However, to make this reasoning *attractive* enough to become a natural part of using a mathematical assistant system, we consider it important to supply a *built-in* representation of at least the relevant syntactic entities. In other words, one particular way of expressing statements about terms, formulae, etc. needs to be chosen, along with an appealing syntax, and made part of the logical language.

We start from the logic previously employed in the Theorema system, namely an untyped higher-order predicate logic with sequence variables. Sequence variables [16] represent sequences of values and have proven to be very convenient for expressing statements about operations with variable arity. For instance, the operation app that appends two lists can be specified by[3]

$$\underset{\overline{xs}}{\forall}\,\underset{\overline{ys}}{\forall}\,\mathsf{app}[\{\overline{xs}\},\{\overline{ys}\}] = \{\overline{xs},\overline{ys}\}$$

using two sequence variables $\overline{xs}$ and $\overline{ys}$. It turns out that sequence variables are also convenient in statements about terms and formulae, since term construction in our logic is a variable arity operation.

2.1 Quoting

Terms in our logic are constructed in two ways: *symbols* (constants or variables) are one kind of terms, and the other are *compound terms*, constructed by 'applying' a 'head' term to a number of 'arguments'.[4] For the representation of symbols, we require the signature to contain a *quoted version* of every symbol. Designating quotation by underlining, we write the quoted version of a as $\underline{\mathsf{a}}$, the quoted version of f as $\underline{\mathsf{f}}$, etc. Quoted symbols are themselves symbols, so there are quoted versions of them too, i.e. if a is in the signature, then so are $\underline{\mathsf{a}}$, $\underline{\underline{\mathsf{a}}}$, etc. For compound terms, the obvious representation would have been a dedicated term construction function, say mkTerm, such that $\mathsf{f}[\mathsf{a}]$ would be denoted by $\mathsf{mkTerm}[\underline{\mathsf{f}},\underline{\mathsf{a}}]$. Using a special syntax, e.g. fancy brackets, would have allowed us to write something like $\underline{\mathsf{f}}\llbracket\underline{\mathsf{a}}\rrbracket$. However, experiments revealed that (in an untyped logic!) it is easiest to reuse the function application brackets $[\cdots]$ for term construction and requiring that if whatever stands to the left of the brackets is a term, then term construction instead of function application is meant. Any axioms or reasoning rules involving term construction contain this condition on the head term. This allows us to write $\underline{\mathsf{f}}[\underline{\mathsf{a}}]$, which is easier to read and easier to input to the system. For reasoning, the extra condition that the head needs to be a term is no hindrance, since this condition usually has to be dealt with anyway.

To further simplify reading and writing of quoted expressions, Mini-Tma allows underlining a whole sub-expression as a shorthand for recursively underlining all occurring symbols. For instance, $\underline{\mathsf{f}[\mathsf{a},\mathsf{h}[\mathsf{b}]]}$ is accepted as shorthand for $\underline{\mathsf{f}}[\underline{\mathsf{a}},\underline{\mathsf{h}}[\underline{\mathsf{b}}]]$. The system will also output quoted terms in this fashion whenever possible. While this is convenient, it is important to understand that it is just a nicer presentation of the underlying representation that requires only quoting of symbols and complex term construction as function application.

[3] Following the notation of Mathematica and Theorema, we use square brackets $[\cdots]$ to denote function application throughout this paper. Constant symbols will be set in sans-serif type, and variable names in *italic*.

[4] See Sect. 2.2 for the issue of variable binding in quantifiers, lambda terms, and such.

2.2 Dealing with Variable Binding

In the literature, various thoughts can be found on how to appropriately represent variable binding operators, i.e. quantifiers, lambda abstraction, etc. The dominant approaches are 1. higher-order abstract syntax, 2. de Bruijn indices, and 3. explicit representation.

Higher-order abstract syntax (HOAS) [17] is often used to represent variable binding in logical frameworks and other systems built on higher-order logic or type theory. With HOAS, a formula $\underset{x}{\forall}\,\mathsf{p}[x]$ would be represented as $\underline{\mathsf{ForAll}}[\underset{\xi}{\underline{\lambda}}\,\underline{\mathsf{p}}[\xi]]$. The argument of the $\underline{\mathsf{ForAll}}$ symbol is a function which, for any term ξ delivers the result of substituting ξ for the bound variable x in the scope of the quantifier. This representation has its advantages, in particular that terms are automatically stored modulo renaming of bound variables, and that capture-avoiding substitution comes for free, but we found it to be unsuitable for our purposes: some syntactic operations, such as comparing two terms for syntactic equality are not effectively possible with HOAS, and also term induction, which is central for reasoning about logics, is not easily described. Hendriks has come to the same conclusion in his work on reflection for Coq [13].

Hendriks uses de Bruijn indices [10], which would represent $\underset{x}{\forall}\,\mathsf{p}[x]$ by a term like $\underline{\mathsf{ForAll}}[\underline{\mathsf{p}}[v_1]]$, where v_1 means the variable bound by the innermost binding operator, v_2 would mean to look one level further out, etc. This representation has some advantages for the implementation of term manipulation operations and also for reflective reasoning about the logic.

For Mini-Tma however, in view of the projected integration of our work into the Theorema system, we chose a simple explicit representation. The reason is mainly that we wanted the representations to be as readable and natural as possible, to make it easy to debug reasoners, to use them in interactive theorem proving, etc. A representation that drops the names of variables would have been disadvantageous. The only derivation from a straight-forward representation is that we restrict ourselves to λ abstraction as the only binding operator. Thus $\underset{x}{\forall}\,\mathsf{p}[x]$ is represented as

$$\underline{\mathsf{ForAll}}[\underline{\lambda}[\underline{x}, \underline{\mathsf{p}}[\underline{x}]]]$$

where $\underline{\lambda}$ is an ordinary (quoted) symbol, that does not have any binding properties. The reason for having only one binding operator is to be able to describe operations like capture avoiding substitution without explicitly naming all operators that might bind a variable. Under this convention, we consider the effort of explicitly dealing with α-conversion to be acceptable: the additional difficulty appears mostly in a few basic operations on terms, which can be implemented once and for all, after which there is no longer any big difference between the various representations.

2.3 An Execution Mechanism

Writing and verifying programs has always been part of the Theorema project's view of mathematical theory exploration [15]. It is also important in the context

of this paper, since we want users of the system to be able to define new reasoners, meaning programs that act on terms.

In order to keep the system's input language as simple and homogenous as possible, we use its logical language as programming language. Instead of fixing any particular way of interpreting formulae as programs, Mini-Tma supports the general concept of *computation mechanisms*. Computations are invoked from the user interface by typing[5]

$$\text{Compute}[\mathit{term}, \text{by} \rightarrow \mathit{comp}, \text{using} \rightarrow \mathit{ax}]$$

where *term* is the term which should be evaluated, *comp* names a computation mechanism, and *ax* is a set of previously declared axioms. Technically, *comp* is a function that is given *term* and *ax* as arguments, and which eventually returns a term. The intention is that *comp* should compute the value of *term*, possibly controlled by the formulae in *ax*. General purpose computation mechanisms require the formulae of *ax* to belong to a well-defined subset of predicate logic, which is interpreted as a programming language. A special purpose computation mechanism might e.g. only perform arithmetic simplifications on expressions involving concrete integers, and completely ignore the axioms. In principle, the author of a computation mechanism has complete freedom to choose what to do with the term and the axioms.

We shall see in Sect. 3 that it is possible to define new computation mechanisms in Mini-Tma. It is however inevitable to provide at least one built-in computation mechanism which can be used to define others. This 'standard' computation mechanism of Mini-Tma is currently based on conditional rewriting. It requires the axioms to be equational Horn clauses.[6] Program execution proceeds by interpreting these Horn clauses as conditional rewrite rules, applying equalities from left to right. Rules are exhaustively applied innermost-first, and left-to-right, and applicability is tested in the order in which the axioms are given. The conditions are evaluated using the same computation mechanism, and all conditions have to evaluate to True for a rule to be applicable. The system does not order equations, nor does it perform completion. Termination and confluence are in the responsibility of the programmer.

Mini-Tma does not include a predefined concept of *proving mechanism*. Theorem provers are simply realized as computation mechanisms that simplify a formula to True if they can prove it, and return it unchanged (or maybe partially simplified) otherwise.

3 Defining Reasoners

Since reasoners are just special computation mechanisms in Mini-Tma, we are interested in how to add a new computation mechanism to the system. This is

[5] Compute, by, using are part of the *User Language*, used to issue commands to the system. Keywords of the User Language will by set in a serif font.

[6] Actually, for convenience, a slightly more general format is accepted, but it is transformed to equational Horn clauses before execution.

done in two steps: first, using some existing computation mechanism, we define a function that takes a (quoted) term and a set of (quoted) axioms, and returns another (quoted) term. Then we tell the system that the defined function should be usable as computation mechanism with a certain name.

Consider for instance an exploration of the theory of natural numbers. After the associativity of addition has been proved, and used to prove several other theorems, we notice that it is always possible to rewrite terms in such a way that all sums are grouped to the right. Moreover, this transformation is often useful in proofs, since it obviates most explicit applications of the associativity lemma. This suggests implementing a new computation mechanism that transforms terms containing the operator Plus in such a way that all applications of Plus are grouped to the right. E.g., we want to transform the term Plus[Plus[a, b], Plus[c, d]] to Plus[a, Plus[b, Plus[c, d]]], ignoring any axioms. We start by defining a function that will transform *representations* of terms, e.g. $\underline{\text{Plus}}[\underline{\text{Plus}}[\underline{\text{a}}, \underline{\text{b}}], \underline{\text{Plus}}[\underline{\text{c}}, \underline{\text{d}}]]$ to $\underline{\text{Plus}}[\underline{\text{a}}, \underline{\text{Plus}}[\underline{\text{b}}, \underline{\text{Plus}}[\underline{\text{c}}, \underline{\text{d}}]]]$. We do this with the following definition:

$$
\begin{array}{l}
\text{Axioms}\,[\texttt{"shift parens"}, \text{any}[s, t, t_1, t_2, acc, l, ax, comp], \\
\quad \textsf{simp}[t, ax, comp] = \textsf{add-terms}[\textsf{collect}[t, \{\}]] \\
\quad \textsf{collect}\,[\underline{\text{Plus}}[t_1, t_2], acc] = \textsf{collect}[t_1, \textsf{collect}[t_2, acc]] \\
\quad \textsf{is-symbol}[t] \Rightarrow \textsf{collect}[t, acc] = \textsf{cons}[t, acc] \\
\quad \textsf{head}[t] \neq \underline{\text{Plus}} \Rightarrow \textsf{collect}[t, acc] = \textsf{cons}[t, acc] \\
\quad \textsf{add-terms}[\{\}] = \underline{0} \\
\quad \textsf{add-terms}[\textsf{cons}[t, \{\}]] = t \\
\quad \textsf{add-terms}[\textsf{cons}[s, \textsf{cons}[t, l]]] = \underline{\text{Plus}}[s, \textsf{add-terms}[\textsf{cons}[t, l]]] \\
]
\end{array}
$$

The main function is simp, its arguments are the term t, the set of axioms ax, and another computation mechanism *comp*, which will be explained later. simp performs its task by calling an auxiliary function collect which recursively collects the fringe of non-Plus subterms in a term, prepending them to an accumulator *acc* that is passed in as second argument, and that starts out empty. To continue our example, $\textsf{collect}[\underline{\text{Plus}}[\underline{\text{Plus}}[\underline{\text{a}}, \underline{\text{b}}], \underline{\text{Plus}}[\underline{\text{c}}, \underline{\text{d}}]], \{\}]$ evaluates to the list of (quoted) terms $\{\underline{\text{a}}, \underline{\text{b}}, \underline{\text{c}}, \underline{\text{d}}\}$. This list is then passed to a second auxiliary function add-terms which builds a $\underline{\text{Plus}}$-term from the elements of a list, grouping to the right. Note that this transformation is done completely without reference to rewriting or the associativity lemma. We are interested in programs that can perform arbitrary operations on terms.

The function is-symbol is evaluated to True if its argument represents a symbol and not a complex term or any other object. This and some other operations (equality of terms, ...) are handled by built-in rewriting rules since a normal axiomatization would not be possible, or in some cases too inefficient.

Given these axioms, we can now ask the system to simplify a term:

$$
\begin{array}{l}
\text{Compute}[\textsf{simp}[\underline{\text{Plus}}[\underline{\text{Plus}}[\underline{\text{a}}, \underline{\text{b}}], \underline{\text{Plus}}[\underline{\text{c}}, \underline{\text{d}}]]], \{\}, \{\}], \text{by} \rightarrow \text{ConditionalRewriting}, \\
\qquad \text{using} \rightarrow \{\text{Axioms}[\texttt{"shift parens"}], \ldots\}]
\end{array}
$$

We are passing in dummy arguments for *ax* and *comp*, since they will be discarded anyway. Mini-Tma will answer with the term Plus[a, Plus[b, Plus[c, d]]].

So far, this is an example of a computation that works on terms, and not very different from a computation on, say, numbers. But we can now make simp known to the system as a computation mechanism. After typing

$$\text{DeclareComputer}[\text{ShiftParens}, \mathsf{simp}, \text{by} \rightarrow \text{ConditionalRewriting}, \\ \text{using} \rightarrow \{\text{Axioms}[\texttt{"shift parens"}, \ldots\}]$$

the system recognizes a new computation mechanism named ShiftParens. We can now tell it to

$$\text{Compute}[\mathsf{Plus}[\mathsf{Plus}[\mathsf{a}, \mathsf{b}], \mathsf{Plus}[\mathsf{c}, \mathsf{d}]], \text{by} \rightarrow \text{ShiftParens}]$$

and receive the answer Plus[a, Plus[b, Plus[c, d]]]. No more quotation is needed, the behavior is just like for any built-in computation mechanism. Also note that no axioms need to be given, since the ShiftParens computation mechanism does its job without considering the axioms.

We now come back to the extra argument *comp*: Mini-Tma allows computation mechanisms to be combined in various ways, which we shall not discuss in this paper, in order to obtain more complex behavior. However, even when actual computations are done by different mechanisms, within any invocation of Compute, there is always one *global computation mechanism*, which is the top-level one the user asked for. It happens quite frequently that user-defined computation mechanisms would like to delegate the evaluation of subterms that they cannot handle themselves to the global computation mechanism. It is therefore provided as the argument *comp* to every function that is used as a computation mechanism, and it can be called like a function.

Calling a user-defined computation mechanism declared to be implemented as a function simp on a term t with some axioms ax under a global computation mechanism *comp* proceeds as follows: 1. t is quoted, i.e. a term t' is constructed that represents t, 2. $\mathsf{simp}[t', ax, comp]$ is evaluated using the computation mechanism and axioms fixed in the DeclareComputer invocation. 3. The result s' should be the representation of a term s, and that s is the result. If step 2 does not yield a quoted term, an error is signaled.

The ShiftParens simplifier is of course a very simple example, but the same principle can clearly be used to define and execute arbitrary syntactic manipulations, including proof search mechanisms within the system's logical language. Since most reasoning algorithms proceed by applying reasoning rules to some proof state, constructing a proof tree, the Theorema implementation will include facilities that make it easy to express this style of algorithm, which would be more cumbersome to implement in out prototypical Mini-Tma system.

4 Reasoning About Logic

To prove statements about the terms and formulae of the logic, we need a prover that supports structural induction on terms, or *term induction* for short.

An interesting aspect is that terms in Mini-Tma, like in Theorema, can have variable arity—there is no type system that enforces the arities of function applications—and arbitrary terms can appear as the heads of complex terms. Sequence variables are very convenient in dealing with the variable length argument lists. While axiomatizing operations like capture avoiding substitution on arbitrary term representations, we employed a recursion scheme based on the observation that a term is either a symbol, or a complex term with an empty argument list, or the result of adding an extra argument to the front of the argument list of another complex term, or a lambda abstraction. The corresponding induction rule is:[7]

$$\frac{\begin{array}{l}\underset{\text{is-symbol}[s]}{\forall} P[s] \\ \underset{\text{is-term}[f]}{\forall} (P[f] \Rightarrow P[f[]]) \\ \underset{\substack{\text{is-term}[f] \\ \text{is-term}[hd] \\ \text{are-terms}[\overline{tl}]}}{\forall} (P[hd] \wedge P[f[\overline{tl}]] \Rightarrow P[f[hd, \overline{tl}]]) \\ \underset{\substack{\text{is-term}[t] \\ \text{is-symbol}[x]}}{\forall} (P[t] \Rightarrow P[\underline{\lambda}[x, t]])\end{array}}{\underset{\text{is-term}[t]}{\forall} P[t]}$$

Using the mechanism outlined in Sect. 3, we were able to implement a simple term induction prover, that applies the term induction rule once, and then tries to prove the individual cases using standard techniques (conditional rewriting and case distinction), in less than 1000 characters of code. This naïve prover is sufficient to prove simple statements about terms, like e.g.

$$\underset{\substack{\text{is-term}[t] \\ \text{is-symbol}[v] \\ \text{is-term}[s]}}{\forall} (\text{not-free}[t, v] \Rightarrow t\{v \rightarrow s\} = t)$$

where $\text{not-free}[t, v]$ denotes that the variable v does not occur free in t, and $t\{v \rightarrow s\}$ denotes capture avoiding substitution of v by s in t, and both these notions are defined through suitable axiomatizations.

5 Reasoning About Reasoners

Program verification plays an important role in the Theorema project [15]. Using predicate logic as a programming language obviously makes it particularly easy to reason about programs' partial correctness. Of course, termination has to be proved separately.

With Mini-Tma's facilities for writing syntax manipulating programs, and for reasoning about syntactic entities, it should come as no surprise that it is

[7] $\underset{p[x]}{\forall} q[x]$ is just convenient syntax for $\underset{x}{\forall} (p[x] \Rightarrow q[x])$

possible to use Mini-Tma to reason about reasoners written in Mini-Tma. The first application that comes to mind is proving the soundness of new reasoners: they should not be able to prove incorrect statements. Other applications include completeness for a certain class of problems, proving that a simplifier produces output of a certain form, etc.

So far, we have concentrated mainly on soundness proofs. In the literature, we have found two ways of proving the soundness of reasoners: the first way consists in proving that the new reasoner cannot *prove* anything that cannot be proved by the existing calculus. Or, in the case of a simplifier like ShiftParens of Sect. 3, if a simplifier simplifies t to t', then there is a *rewriting* proof between t and t'. This approach is very difficult to follow in practice: it requires formalizing the existing calculus, including proof trees, possibly rewriting, etc. Often the soundness of a reasoner will depend on certain properties of the involved operations, e.g. ShiftParens requires the associativity of Plus, so the knowledge base has to be axiomatized as well. Moreover, to achieve reasonable proof automation, the axiomatization needs to be suitable for the employed prover: finding a proof can already be hard, making prover A *prove* that prover B will find a proof essentially requires re-programming B in the axiomatization. And finally, this correctness argument works purely on the syntactic level: any special reasoning techniques available for the mathematical objects some reasoner is concerned with are useless for its verification!

We have therefore preferred to investigate a second approach: we prove that anything a new reasoner can prove is simply *true* with respect to a model semantics. Or, for a simplifier that simplifies t to t', that t and t' have the same *value* with respect to the semantics. This approach has also been taken in the very successful NqThm and ACL2 systems [2, 14]. It solves the above problems, since it is a lot easier to axiomatize a model semantics for our logic, and the axiomatization is also very easy to use for an automated theorem prover. The knowledge base does not need to be 'quoted', since much of the reasoning is about the values instead of the terms, and for the same reason, any previously implemented special reasoners can be employed in the verification.

Similarly to ACL2, we supply a function $\mathsf{eval}[t, \beta]$ that recursively evaluates a term t under some assignment β that provides the meaning of symbols.[8] To prove the soundness of ShiftParens, we have to show

$$\mathsf{eval}[\mathsf{simp}[t, ax, comp], \beta] = \mathsf{eval}[t, \beta]$$

for any term t, any ax and $comp$ and any β with $\beta[\underline{\mathsf{0}}] = \mathsf{0}$ and $\beta[\underline{\mathsf{Plus}}] = \mathsf{Plus}$. To prove this statement inductively, it needs to be strengthened to

$$\mathsf{eval}[\mathsf{add\text{-}terms}[\mathsf{collect}[t, acc]], \beta] = \mathsf{eval}[\underline{\mathsf{Plus}}[t, \mathsf{add\text{-}terms}[acc]], \beta] \qquad (*)$$

for any acc, and an additional lemma

$$\mathsf{eval}[\mathsf{add\text{-}terms}[\mathsf{cons}[t, l]], \beta] = \mathsf{Plus}[\mathsf{eval}[t, \beta], \mathsf{eval}[\mathsf{add\text{-}terms}[l], \beta]]$$

[8] Care needs to be taken when applying eval to terms containing $\underline{\mathsf{eval}}$, as has already been recognized by Boyer and Moore [3].

is required. And of course, the associativity of Plus needs to known. Mini-Tma cannot prove (∗) with the term induction prover described in Sect. 4, since it is not capable of detecting the special role of the symbol Plus. However, using a modified induction prover which treats compound terms with head symbol Plus as a separate case, (*) can be proved automatically.

Automatically extracting such case distinctions from a program is quite conceivable, and one possible topic for future work on Mini-Tma.

Ultimately, we intend to improve and extend the presented approach, so that it will be possible to successively perform the following tasks within a single framework, using a common logical language and a single interface to the system:

1. define and prove theorems about the concept of Gröbner bases [4],
2. implement an algorithm to compute Gröbner bases,
3. prove that the implementation is correct,
4. implement a new theorem prover for statements in geometry based on coordinatization, and which uses our implementation of the Gröbner bases algorithm,
5. prove soundness of the new theorem prover, using the shown properties of the Gröbner bases algorithm,
6. prove theorems in geometry using the new theorem prover, in the same way as other theorem provers are used in the system.

Though the case studies performed so far are comparatively modest, we hope to have convinced the reader that the outlined approach can be extended to more complex applications.

6 Foundational Issues

Most previous work on reflection in theorem proving environments (see Sect. 7) has concentrated on the subtle foundational problems arising from adding reflection to an existing system. In particular, any axiomatization of the fact that a reflectively axiomatized logic behaves exactly like the one it is being defined in can easily lead to inconsistency. In our case, care needs to be taken with the evaluation function eval which connects the quoted logic to the logic it is embedded in.

However, within the Theorema project, we are not particularly interested in the choice and justification of a single logical basis. Any framework a mathematician considers appropriate for the formalization of mathematical content should be applicable within the system—be it one or the other flavor of set theory, type theory, or simply first-order logic. Any restriction to one particular framework would mean a restriction to one particular view of mathematics, which is something we want to avoid. This is why there is no such thing as *the* logic of Theorema. But if there is no unique, well-defined basic logic, then neither can we give a precise formal basis for its reflective extension. In fact, since the way in which such an extension is defined is itself an interesting mathematical subject, we do not even want to restrict ourselves to a single way of doing it.

This is of course somewhat unsatisfying, and it is actually not the whole truth. We *are* trying to discover a particularly viable *standard* method of adding reflection and reflective reasoners. And we are indeed worried about the soundness of that method. It turns out that one can convince oneself of the soundness of such an extension provided the underlying logic satisfies a number of reasonable assumptions.

Let a logical language $\mathcal{L}$ be given. In the context of formalization of mathematics, we may assume that syntactically, $\mathcal{L}$ consists of a subset of the formulae of higher order predicate logic. Typically, some type system will forbid the construction of certain ill-typed formulae, maybe there is also a restriction to first-order formulae.

Most logics permit using a countably infinite signature, in fact, many calculi require the presence of infinitely many constant symbols for skolemization. Adding a quoted symbol $\underline{\mathsf{a}}$ for any symbol a of $\mathcal{L}$ will then be unproblematic.

Next, we can add a function $\mathsf{is\text{-}symbol}$, which may be defined through a countably infinite and effectively enumerable family of axioms, which should not pose any problems. The function $\mathsf{is\text{-}term}$ can then be axiomatized recursively in any logic that permits recursive definitions. We can assume for the moment that the logic does not include quoting for $\mathsf{is\text{-}symbol}$ or $\mathsf{is\text{-}term}$, and that the functions will recognize the quotations of symbols and terms of $\mathcal{L}$, and not of the reflectiove extension of $\mathcal{L}$ we are constructing.

Likewise, if the evaluation of basic symbols is delegated to an assignment β, it should be possible to give an axiomatization of the recursive evaluation function eval within any logic that permits recursive definitions:

$$\mathsf{is\text{-}symbol}[t] \Rightarrow \mathsf{eval}[t, \beta] = \beta[t]$$
$$\mathsf{is\text{-}term}[f] \Rightarrow \mathsf{eval}[f[t], \beta] = \mathsf{eval}[f, \beta][\mathsf{eval}[t, \beta]]$$

The exact definitions will depend on the details of $\mathcal{L}$. For instance, if $\mathcal{L}$ is typed, it might be necessary to introduce a family of eval functions for terms of different types, etc. Still, we do not believe that soundness problems can occur here.

The interesting step is now the introduction of an unquoting function unq, which relates every quoted symbol $\underline{\mathsf{a}}$ to the entity it represents, namely a. We define unq by the axioms

$$\mathsf{unq}[\mathbf{s}'] = \mathbf{s}$$

for all symbols $\mathbf{s}$ of $\mathcal{L}$, where $\mathbf{s}'$ denotes the result of applying one level of reflection quoting to $\mathbf{s}$, i.e. $\mathsf{unq}[\underline{\mathsf{a}}] = \mathsf{a}$, $\mathsf{unq}[\underline{\mathsf{b}}] = \mathsf{b}$,... The formula $\mathsf{unq}[\underline{\mathsf{unq}}] = \mathsf{unq}$ is *not* an axiom since this would precisely lead to the kind of problems identified by Boyer and Moore in [3]. If they are only present for the symbols of the original logic, these axioms do not pose any problems.

All in all, the combined extension is then a conservative extension of the original logic, meaning that any model $\mathcal{M}$ for a set Φ of formulae of $\mathcal{L}$ can be extended to a model $\mathcal{M}'$ of Φ in the reflective extension, such that $\mathcal{M}'$ behaves like $\mathcal{M}$ when restricted to the syntax of $\mathcal{L}$. Moreover, in the extension, the formula

$$\mathsf{eval}[\mathbf{t}', \mathsf{unq}] = \mathbf{t}$$

holds for every term $\mathbf{t}$ of $\mathcal{L}$ with quotation $\mathbf{t}'$, which justifies using eval to prove the correctness of new reasoners.

To allow for several levels of quotation, this process can be iterated. It is easy to see that the is-symbol, is-term, and eval functions defined for consecutive levels can be merged. For the unq function, one possible solution is to use a hierarchy $\mathsf{unq}^{(i)}$ of unquoting functions, where there is an axiom $\mathsf{unq}^{(i)}[\underline{\mathsf{unq}^{(j)}}] = \mathsf{unq}^{(j)}$ if and only if $j < i$.

Another difficulty is the introduction of *new* symbols required by many calculi for skolemization, which can be jeopardized by the presence of knowledge about the unquoting of quoted symbols. Here, a possible solution is to fix the set of symbols for which unq axioms are required before proofs, as is done in ACL2.

7 Related Work

John Harrison has written a very thorough survey [11] of reflection mechanisms in theorem proving systems, and most of the work reviewed there is in some way connected to ours.

The most closely related approach is surely that of the NqThm and ACL2 systems, see e.g. [2, 14]. The proving power of these systems can be extended by writing simplifiers in the same programming language as that which can be verified by the system. Before using a new simplifier, its soundness has to be shown using a technique similar to that of Sect. 5. Our work extends theirs in the following respects:

- We use a stronger logic, ACL2 is restricted to first-order quantifier-free logic.
- Our framework allows coding full, possibly non-terminating theorem provers, and not just simplifiers embedded in a fixed prover.
- Through the *comp* argument, reasoners can be called recursively.
- The specialized quoting syntax and sequence variables make Mini-Tma more pleasant and practical to use.
- In Mini-Tma, Meta-programming can be used without being forced to prove soundness first, which is useful for experimentation and exploration.

Experiments in reflection have also recently been done in Coq [13], but to our knowledge these are restricted to first-order logic, and meta-programmed provers cannot be used as part of a proof construction. There has also been some work on adding reflection to Nuprl [1]. This is still in its beginnings, and its principal focus seems to be to prove theorems about logics, while our main goal is to increase the system's reasoning power.

Recent work on the self-verification of HOL Light [12] is of a different character. Here, the HOL Light system is not used to verify extensions of itself, but rather for the self-verification of the kernel of the system. Self-verification raises some foundational issues of its own that do not occur in our work.

In the context of programming languages, LISP has always supported quoting of programs and meta-programming, e.g. in macros. Amongst more modern languages, Maude should be mentioned for its practically employed reflective

capabilities, see e.g. [9]. A quoting mechanism is part of the language, and it is used to define the 'full' Maude language in terms of a smaller basic language. However, this reflection is just used for programming, there is no reasoning involved.

8 Conclusion and Future Work

We have reported on ongoing research in the frame of the Theorema project, that aims at making coding of new (special purpose) reasoners and reasoning about them, e.g. to prove their soundness, an integral part of the theory exploration process within the system. The approach has been evaluated in the prototypical implementation Mini-Tma.

The main features of our approach are to start from a logic with a built-in quoting mechanism, and to use the same logic for the definition of programs, and in particular reasoners. We have shown that this makes it possible to define reasoners which can be used by the system like the built-in ones. It also enables the user to reason about terms, formulae, etc. and also about reasoners themselves.

We have briefly discussed two alternatives for defining and proving the soundness of new reasoners, and concluded that an approach based on formalizing a model semantics is more suitable for automated deduction than one that is based on formalizing proof theory.

Future work includes improving the execution efficiency of programs written within the Theorema logic. Improvements are also required for the theorem proving methods, i.e. better heuristics for term induction, program verification, etc., but also the production of human-readable proof texts or proof trees, which are essential for the successful application of the theorem provers. All these developments will have to be accompanied by case studies demonstrating their effectiveness.

Acknowledgments

The authors would like to thank the other members of the Theorema Group, in particular Temur Kutsia and Markus Rosenkranz, for contributing to the numerous intense discussions about the presented work.

References

1. Eli Barzilay. *Implementing Reflection in Nuprl.* PhD thesis, Cornell University Computer Science, 2006.
2. Robert S. Boyer, Matt Kaufmann, and J Strother Moore. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.
3. Robert S. Boyer and J Strother Moore. The addition of bounded quantification and partial functions to a computational logic and its theorem prover. *J. Autom. Reasoning*, 4(2):117–172, 1988.

4. Bruno Buchberger. Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen Gleichungssystems. *Aequationes Math.*, 4:374–383, 1970. English translation published in [8].
5. Bruno Buchberger. Lifting knowledge to the state of inferencing. Technical Report TR 2004-12-03, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 2004.
6. Bruno Buchberger. Proving by first and intermediate principles, November 2, 2004. Invited talk at Workshop on Types for Mathematics / Libraries of Formal Mathematics, University of Nijmegen, The Netherlands.
7. Bruno Buchberger, Adrian Crăciun, Tudor Jebelean, Laura Kovács, Temur Kutsia, Koji Nakagawa, Florina Piroi, Nikolaj Popov, Judit Robu, Markus Rosenkranz, and Wolfgang Windsteiger. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, pages 470–504, 2006.
8. Bruno Buchberger and Franz Winkler. Gröbner bases and applications. In B. Buchberger and F. Winkler, editors, *33 Years of Gröbner Bases*, London Mathematical Society Lecture Notes Series 251. Cambridge University Press, 1998.
9. Manuel Clavel and José Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, 2002.
10. Nicolas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 34:381–392, 1972.
11. John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995.
12. John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *LNCS*, pages 177–191. Springer, 2006.
13. Dimitri Hendriks. Proof reflection in Coq. *Journal of Automated Reasoning*, 29(3–4):277–307, 2002.
14. Warren A. Hunt Jr., Matt Kaufmann, Robert Bellarmine Krug, J Strother Moore, and Eric Whitman Smith. Meta reasoning in ACL2. In Joe Hurd and Thomas F. Melham, editors, *Proc. Theorem Proving in Higher Order Logics, TPHOLs 2005, Oxford, UK*, volume 3603 of *LNCS*, pages 163–178. Springer, 2005.
15. Laura Kovács, Nikolaj Popov, and Tudor Jebelean. Verification environment in Theorema. *Annals of Mathematics, Computing and Teleinformatics (AMCT)*, 1(2):27–34, 2005.
16. Temur Kutsia and Bruno Buchberger. Predicate logic with sequence variables and sequence function symbols. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Proc. 3rd Intl. Conf. on Mathematical Knowledge Management, MKM'04*, volume 3119 of *LNCS*, pages 205–219. Springer Verlag, 2004.
17. Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proc. ACM SIGPLAN 1988 Conf. on Programming Language design and Implementation, PLDI '88, Atlanta, United States*, pages 199–208. ACM Press, New York, 1988.
18. Stephen Wolfram. *The Mathematica Book*. Wolfram Media, Inc., 1996.

On the Efficiency of Geometry Theorem Proving by Gröbner Bases

Shuichi Moritsugu and Chisato Arai

University of Tsukuba,
Tsukuba 305-8550, Ibaraki, JAPAN,
{moritsug, arai}@slis.tsukuba.ac.jp

Abstract. We show experimental results for proving Euclidean geometry theorems by Gröbner basis method. In 1988, Chou Shang-Ching proved 512 theorems by Wu's method, and reported that 35 among them remained unsolvable by Gröbner basis method. In this paper, we tried to prove these 35 theorems by Gröbner basis method, and we succeeded in proving 26 theorems but have found that the other 9 theorems are essentially difficult to compute Gröbner bases. We show the table of timing data and discuss several devices to complete the proof by solving radical membership problem.

1 Introduction

In the area of mechanical proving geometry theorems, Wu's method [21] has been widely and successfully used since Wu Wen-Tsün introduced the original algorithm in 1977. Meanwhile, another approach [9, 10, 20] based on Gröbner basis method [2] was also proposed and has been studied.

In 1988, Chou Shang-Ching [3] published an extensive collection of 512 theorems that were proved by Wu's method. He also applied Gröbner basis method and succeeded in proving 477 theorems among them. However, it is reported that none of the computation for the other 35 theorems finished within 4 hours.

Since then, there seems to have been few attempts to reconfirm Chou's results, even though the inferiority of Gröbner basis method to Wu's method is sometimes pointed out from the viewpoint of computational efficiency. However, a recent project [7] is in progress, which is intended to collect and construct the benchmark including the Chou's problems.

Independently, our group has been trying to prove the above 35 theorems by Gröbner basis method since 2004, and we succeeded in proving 26 theorems among them [13]. On the other hand, we consider that the 9 theorems left are essentially difficult to compute the Gröbner bases.

In this paper, we show the results of computation by both of Gröbner basis method and Wu's method, and we discuss the comparison of several ways to solve the radical membership using Maple11 [12] and Epsilon library [19].

2 Algebraic Proof by Gröbner Basis Method

2.1 Radical Membership Problem

We translate the geometric hypotheses in the theorem into polynomials

$$f_1, \ldots, f_\ell \ \in \ \mathbf{Q}(u_1, \ldots, u_m)\,[x_1, \ldots, x_n]\,.$$

According to Chou [3], we construct the points in order, using two types of variables:

- u_i: parameters whose values are arbitrarily chosen,
- x_j: variables whose values depend on other u_i, x_k.

Next, we translate the conclusion of the theorem into

$$g \ \in \ \mathbf{Q}(u_1, \ldots, u_m)\,[x_1, \ldots, x_n]\,.$$

In these formulations, we apply the following proposition to "prove the theorem":

$$g \text{ follows generically from } f_1, \ldots, f_\ell \quad \Leftrightarrow \quad g \in \sqrt{(f_1, \ldots, f_\ell)}.$$

There exist several methods to solve this type of radical membership problem, and we adopt the following proposition [4] whose algorithmic computations are based on Gröbner bases. More precise description of the algorithms and further references can be found in [18].

Proposition 1 (Radical Membership) *Let K be an arbitrary field and let $I = (f_1, \ldots, f_\ell) \subset K\,[x_1, \ldots, x_n]$ be a polynomial ideal. We compute the Gröbner basis G of I with any term order, and let $g \in K\,[x_1, \ldots, x_n]$ be another polynomial. Then, $g \in \sqrt{I}$ is equivalent to each of the following three conditions.*

(a) $\exists s \in \mathbf{N}, \quad g^s \xrightarrow{G} 0.$ *(Compute for $s = 1, 2, \ldots$.)*
(b) *For $J = (I, 1 - yg) \subset K\,[x_1, \ldots, x_n, y]$, we have its Gröbner basis with any term order becomes* (1).
(c) *For $J = (I, g - y) \subset K\,[x_1, \ldots, x_n, y]$, if we compute the Gröbner basis with block term order $\{x_1, \ldots, x_n\} \overset{\text{lex}}{>} y$, then the basis contains a univariate polynomial such as y^s $(\exists s \in \mathbf{N})$.* ∎

Since $K = \mathbf{Q}(u_1, \ldots, u_m)$ in our formulation, the formula (c) is not efficient due to the term order and is not suitable for proving geometric theorems.

When we apply the formula (a), we first try $g \xrightarrow{G} 0$. Since you have $s = 1$ in almost all cases for geometry theorems [3, 4], this method seems practical. Actually, we obtained $g \xrightarrow{G} 0$ in all the 26 theorems we succeeded in proof by Gröbner basis method.

The experimental results imply that the formula (b) deserves consideration from the viewpoint of computational efficiency. Hence, later we discuss the comparison of the methods (a) and (b).

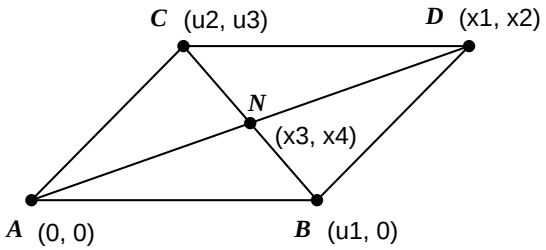


Fig. 1. Parallelogram

2.2 Example

In order to describe the flow of the implemented algorithms, we show an example of proof by Gröbner basis method and Wu's method [3, 4, 19].

Example 1 (Parallelogram) We show the proof for the theorem " the two diagonals of any parallelogram intersect at a point which bisects both diagonals". The following polynomial expressions are the output of our program, where the computations are carried out over $\mathbf{Q}(u_1, u_2, u_3)\,[x_1, x_2, x_3, x_4]$.

(1) Let $A(0,0)$, $B(u_1,0)$, $C(u_2,u_3)$, $D(x_1,x_2)$, $N(x_3,x_4)$ be the points shown in Fig.1. We translate the geometric hypotheses (in order) as follows.
  (i) $AB \parallel CD \quad \Rightarrow \quad f_1 := u_1x_2 - u_3u_1$
  (ii) $AC \parallel BD \quad \Rightarrow \quad f_2 := u_2x_2 - u_3x_1 + u_3u_1$
  (iii) A, N, D are collinear $\quad \Rightarrow \quad f_3 := -x_4x_1 + x_3x_2$
  (iv) B, N, C are collinear $\quad \Rightarrow \quad f_4 := -(u_2 - u_1)x_4 + u_3x_3 - u_3u_1$

(2) We translate the conclusions of the theorem.
  (i) $AN = ND \quad \Rightarrow \quad g_1 := 2x_4x_2 + 2x_3x_1 - x_2^2 - x_1^2$
  (ii) $BN = NC \quad \Rightarrow \quad g_2 := 2u_3x_4 + 2(u_2 - u_1)x_3 - u_3^2 - u_2^2 + u_1^2$

(3) Proof 1: Gröbner basis method
  (i) Using the lexicographic order with $x_4 > x_3 > x_2 > x_1$, we compute the Gröbner basis for the ideal $I = (f_1, f_2, f_3, f_4)$:

$$G = \{2x_4 - u_3, \quad 2x_3 - (u_2 + u_1), \quad x_2 - u_3, \quad x_1 - (u_2 + u_1)\}\,.$$

  (Note) Collecting the prime factors in denominators throughout the Gröbner basis computation, we obtain subsidiary conditions $\{u_1 \neq 0,\ u_2 \neq 0,\ u_2 - u_1 \neq 0,\ u_3 \neq 0\}$. However, we restrict ourselves to the "generic case" [4], and here we do not discuss the constraint for the parameters u_i.
  (ii) Reducing the conclusion g_1 by G, we obtain $g_1 \xrightarrow{G} 0$, hence $g_1 \in I$ is proved. For the conclusion g_2, it is similarly proved that $g_2 \in I$. ∎

(4) Proof 2: Wu's method
We apply the functions 'CharSet' and 'prem' using Epsilon library [19] on Maple11 [12].

(i) Using the order $x_4 > x_3 > x_2 > x_1$, we compute the characteristic polynomials for $\{f_1, f_2, f_3, f_4\}$ and let them
$CS = \{x_1 - u_2 - u_1, \quad u_1x_2 - u_3u_1, \quad -(u_2 - u_1)x_3 + x_3x_1 - u_1x_1,$
$-(u_2 - u_1)x_4 + u_3x_3 - u_3u_1\} \quad =: \quad \{h_1, h_2, h_3, h_4\}$

(ii) For the conclusion g_1 and CS, we compute the sequence of pseudo remainders with x_4, x_3, x_2, x_1 in order:
$g_{13} := \mathrm{prem}(g_1,\ h_4,\ x_4)$
$= -2u_3x_3x_2 - 2(u_2 - u_1)x_3x_1 + (u_2 - u_1)x_2^2 + (u_2 - u_1)x_1^2 + 2u_3u_1x_2,$
$g_{12} := \mathrm{prem}(g_{13},\ h_3,\ x_3)$
$= (u_2 - u_1)(x_2^2x_1 + x_1^3 - (u_2 - u_1)x_2^2 - (u_2 + u_1)x_1^2 - 2u_3u_1x_2),$
$g_{11} := \mathrm{prem}(g_{12},\ h_2,\ x_2) \ = \ u_1^2(u_2 - u_1)(x_1^2 + u_3^2)(x_1 - u_2 - u_1),$
$g_{10} := \mathrm{prem}(g_{11},\ h_1,\ x_1) = 0.$
This result means that the conclusion g_1 is proved.

(iii) For the conclusion g_2, we compute the sequence $g_{23},\ g_{22},\ g_{21},\ g_{20}$ similarly and obtain $g_{20} = 0$, which means that g_2 is proved. ∎

3 Results of Experiment

3.1 Environment and Results of Experiment

Table 1. Environment for Maple & Epsilon

CPU	Pentium 4 (3.6 GHz)
OS	Windows XP professional ed.
Main Memory	2.0 GB

In the computational environment shown in Table 1, we tried to prove 35 among Chou's 512 theorems that were not solved by Gröbner basis method in 1988. We extracted them from the list of timing data by Chou [3] in its Appendix.

Using the graded-reverse-lex (grevlex) order with $x_n > x_{n-1} > \cdots > x_1$, the Gröbner bases were computered over the coefficient field $\mathbf{Q}(u_1, \ldots, u_m)$ by Maple11 [12]. For comparison, Wu's method was also applied using Epsilon library [19] over Maple11.

As a result, we succeeded in proving 26 among the 35 theorems, but the other 9 theorems remained unsolvable. In our previous paper [13], we tried to prove them using three computer algebra systems : Reduce3.6 [8] and Risa/Asir [14], adding to Maple10 [11]. However, none of these systems has succeeded yet in computing Gröbner bases for the same 9 theorems, mainly because of the lack of memory. (Maple10 seems just to take very long time to exhaust the memory.)

For the computation of Gröbner bases by Maple11, we used the option `method=maplef4` first, but it failed in some cases. Then we tried again using

the option `method=buchberger`, and we succeeded in proving 26 theorems in total. It does not seem clear yet which option is suitable for these rational expression coefficient cases $\mathbf{Q}(u_i)[x_i]$.

We show the precise timing data in Table 2, where the columns and symbols indicate the following. In the next subsections, we show the details of devices for computation ($\diamondsuit$, $\clubsuit$).

$\#x_i$	number of dependent variables in the hypotheses
$\#u_i$	number of free parameters in the hypotheses
$\#h_i$	number of polynomials for the hypotheses
$\times$	failure ($*$: insufficient memory)
$\bigcirc$	success by direct computation
$\diamondsuit$, $\clubsuit$	success by some devices
Maple(1)	Using the formula (a) in Proposition 1
Maple(2)	Using the formula (b) in Proposition 1
Epsilon(1)	Using the functions 'CS' and 'prem'
Epsilon(2)	Using the function 'RIM' for radical ideal membership

Wu's method (Epsilon(1)) and Wang's method [16] (Epsilon (2)) seem unstable for a few examples, but we consider that these 35 theorems in total can be proved by Epsilon library.

3.2 Device 1: Incremental Computation ($\diamondsuit$)

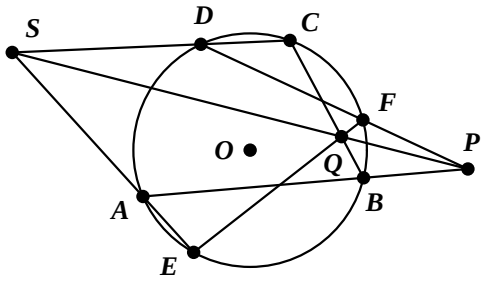


Fig. 2. Example 48

In some theorems, we succeeded in computing the Gröbner basis by grouping the ideal of hypotheses $I = (f_1, \ldots, f_\ell)$, where we changed the input order of these polynomials such as $((\ldots((f_1, \ldots, f_k), f_{k+1}), \ldots), f_\ell)$.

We can see the relations of polynomials f_i by their inclusion of variables, because the geometric hypotheses are constructed in some kind of order. However, if the polynomials $\{f_1, \ldots, f_\ell\}$ are input at a time to computer algebra systems,

Table 2. Success / Failure and CPU-Time(sec) for Chou's Examples

No.	$\#x_i$	$\#u_i$	$\#h_i$	Maple(1)		Maple(2)	Epsilon(1)	Epsilon(2)
ex6	12	11	12	×	*	*	1.33	2.06
ex7	12	11	12	×	*	*	38.73	9.00
ex8	11	8	13	◯	0.92	7.00	0.59	0.31
ex10	20	6	23	×	*	*	0.52	7.38
ex11	20	6	23	×	*	*	0.72	9.31
ex12	20	6	23	×	*	*	1.70	6.45
ex13	17	6	19	×	*	*	0.27	31.53
ex14	17	6	19	×	*	*	0.16	0.86
ex19	17	6	19	×	*	*	3.67	*
ex21	11	4	13	◯	0.69	0.58	0.06	0.58
ex26	13	7	14	◯	0.39	0.16	1.99	2.63
ex40	15	3	15	◯	18.97	42.48	3.91	14.86
ex45	14	3	14	◯	0.22	0.14	0.06	0.20
ex48	10	6	11	♢	545.11	515.31	0.20	0.44
ex63	15	6	19	◯	0.19	0.13	1.17	0.78
ex72	10	6	13	♢♣	0.41	1.30	1468.17	56.08
ex80	14	5	16	×	*	*	19.34	10.03
ex94	7	3	8	◯	4.58	4.50	0.02	0.05
ex96	7	4	7	♢	11.27	11.11	0.02	0.05
ex99	10	4	13	♣	33.28	3.77	2.73	0.42
ex106	8	4	9	◯	2.09	0.06	1.59	0.28
ex109	7	6	11	♣	2.80	0.11	1247.34	7.24
ex115	8	3	10	♣	1.41	0.33	0.11	0.09
ex240	10	3	10	♣	9.69	0.36	0.55	162.95
ex310	14	5	16	♢	6.34	2.70	295.22	17.41
ex311	13	4	17	◯	0.27	0.24	0.05	0.20
ex315	20	4	23	♢♣	1.97	1.92	0.19	0.59
ex316	24	4	31	♢	11.34	2.27	1004.00	493.69
ex367	14	5	18	◯	17.34	2.63	11.25	0.08
ex379	9	4	11	◯	0.59	0.44	0.05	0.22
ex395	5	3	6	◯	0.16	0.14	0.02	0.38
ex396	14	5	16	♣	3.05	2.22	2.13	138.33
ex401	7	6	9	◯	21.53	0.02	1.27	0.14
ex492	17	3	18	◯	0.38	0.23	3.75	1.75
ex507	8	7	8	◯	1.45	0.84	0.49	0.47

the optimal way is not necessarily followed in the inner function for Gröbner bases. Consequently, this incremental computation worked effectively for some examples, even though it is heuristic and not algorithmic.

Example 2 (Example 48 [3]: Fig.2) *If five of six vertices of a hexagon lie on a circle, and the three pairs of opposite sides meet a three collinear points, then the sixth vertex lies on the same circle.*

Hypotheses We translate the following conditions in order : $OA = OC$, $OA = OB$, $DO = OA$, $EO = OA$, P is on line AB, S is on line EA, S is on line CD, Q is on line BC, Q is on line SP, F is on line QE, F is on line PD. Then we obtain 11 polynomials: $f_1, \ldots, f_{11}$.

Conclusion We let $OA = OF$ be expressed by g.

Proof We compute the Gröbner basis G in two steps : $((f_1, \ldots, f_9, f_{11}), f_{10})$, because f_{10} has longer form than others. Then we obtain $g \xrightarrow{G} 0$. ∎

3.3 Device 2: Decomposition of the ideal (♣)

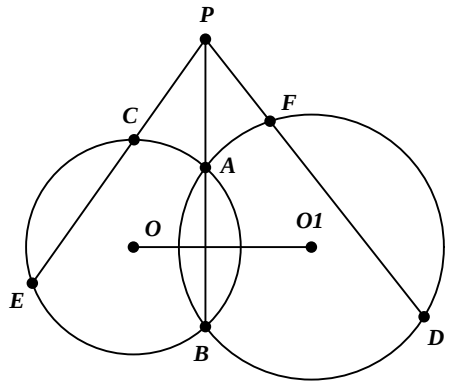


Fig. 3. Example 109

In some cases, we cannot obtain the conclusion $g \xrightarrow{G} 0$ because of insufficient hypotheses. The prover developed by Chou [3] found automatically the nondegenerate conditions that should be added to the hypotheses. Using Chou's results, we added such nondegenerate conditions for x_i's and recomputed the Gröbner bases.

Example 3 (Example 109 [3]: Fig.3) *From a point P on the line joining the two common points A and B of two circles O and O_1, two secants PCE and PFD are drawn to the circles respectively. Show that $PC \cdot PE = PF \cdot PD$.*

Hypotheses We translate the following conditions in order : O_1 is on line OX, $AX \perp XO$, X is the midpoint of AB, P is on line AB, $EO = OA$, $CO = OA$, C is on line PE, $FO_1 = O_1A$, $DO_1 = O_1A$, D is on line PF. Then we obtain 11 polynomials: $f_1, \ldots, f_{11}$.

Conclusion We let $PC \cdot PE = PF \cdot PD$ be expressed by g.

Proof For $I = (f_1, \ldots, f_{11})$, we have $g \notin I$. Hence we need to add the following nondegenerate conditions.

$C(x_4, x_3) \neq E(x_2, u_5) \quad \Rightarrow \quad h_1 := (x_3 - u_5)z_1 - 1 = 0$
$D(x_7, x_6) \neq F(x_5, u_6) \quad \Rightarrow \quad h_2 := (x_6 - u_6)z_2 - 1 = 0$

If we add h_1, h_2 to $I = (f_1, \ldots, f_{11})$ and let $I' = (I, h_1, h_2)$, then we have $g \in I'$ and complete the proof.

Note 1 The above nondegenerate conditions can be also computed by the Gröbner basis of $I = (f_1, \ldots, f_{11})$. If we compute the minimal polynomials of x_3 and x_6 in I, we obtain the following $(k = 3, 6)$:

$$I \ni (x_3 - u_5) \cdot \varphi_3(x_3), \quad (x_6 - u_6) \cdot \varphi_6(x_6) \qquad \varphi_k(x_k) \in \mathbf{Q}(u_1, \ldots u_6)[x_k].$$

If we decompose the ideal $I = (f_1, \ldots, f_{11})$, and we restrict ourselves into $\tilde{I} = (I, \varphi_3, \varphi_6)$, then we have $g \in \tilde{I}$ and complete the proof. This implies that $x_3 - u_5 \neq 0$ and $x_6 - u_6 \neq 0$ are necessary as nondegenerate conditions.

Note 2 The theorem itself remains true for the cases where $C = E$ or $D = F$. Above nondegenerate conditions means that the same set of polynomials cannot express such tangent cases in common. This kind of automatic derivation of nondegenerate conditions has been already discussed by several authors such as [1, 15]. ∎

The following example is not included in the 512 theorems by Chou [3], but it is known as the case where the decomposition of components and rather complicated computation are needed to confirm the conclusion. Several authors have succeeded in proving this theorem so far [17], but there does not seem to be any attempt to apply Gröbner method to it. We proved this theorem by the following way based on Gröbner basis algorithms.

Example 4 (Thèbault-Taylor) We follow the second formulation in Chou [3](pp.67-68), where some auxiliary points are added to Fig.4.

Hypotheses We translate the conditions in order and obtain $f_1, \ldots, f_{14} \in \mathbf{Q}(u_2, u_3, u_4)[x_1, \ldots, x_{14}]$.

Conclusion We let the tangent condition of two circles be expressed by g.

Step 1 Computing the Gröbner basis of $I = (f_1, \ldots, f_{14})$, we have $g \notin I$. Actually, this computation fails because g is not reduced to 0 by the Gröbner basis, but its normal form will explode.

Step 2 We try to find a reducible univariate polynomial in the ideal I, and first obtain $\varphi_5(x_5) \cdot \varphi'_5(x_5) \in I$, where the degree in x_5 of each factor is two.

Step 3 We let $\tilde{I} = (I, \varphi_5(x_5))$, but again we have $g \notin \tilde{I}$ by computing the Gröbner basis of $\tilde{I}$. Then, we try to find a reducible univariate polynomial in $\tilde{I}$, and obtain $\varphi_{11}(x_{11}) \cdot \varphi'_{11}(x_{11}) \in \tilde{I}$, where the degree in x_{11} of each factor is two.

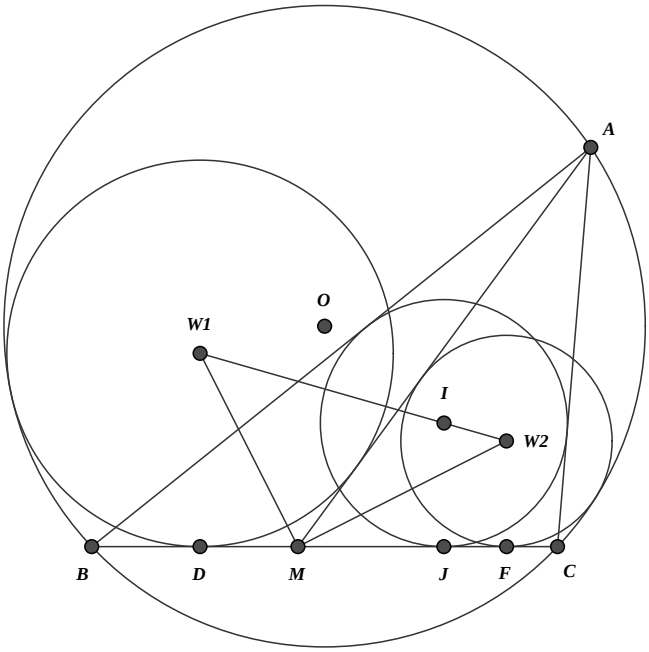


Fig. 4. Thèbault-Taylor's Theorem

Step 4 We let $\tilde{I'} = (I, \varphi_5'(x_5))$, but again we have $g \notin \tilde{I'}$ by computing the Gröbner basis of $\tilde{I'}$. Then, we try to find a reducible univariate polynomial in $\tilde{I'}$, and obtain $\varphi_{11}''(x_{11}) \cdot \varphi_{11}'''(x_{11}) \in \tilde{I'}$, where the degree in x_{11} of each factor is two.

Step 5 Thus the hypotheses ideal I is decomposed into the following 4 components:

$$
\begin{aligned}
I_1 &= ((I, \varphi_5(x_5)),\ \varphi_{11}(x_{11})),\\
I_2 &= ((I, \varphi_5(x_5)),\ \varphi_{11}'(x_{11})),\\
I_3 &= ((I, \varphi_5'(x_5)),\ \varphi_{11}''(x_{11})),\\
I_4 &= ((I, \varphi_5'(x_5)),\ \varphi_{11}'''(x_{11})).
\end{aligned}
$$

Then, we obtain $g \in I_1$ and $g \notin I_2, I_3, I_4$ by computing each Gröbner basis of I_j. Therefore, the conclusion is confirmed to be true only in the ideal I_1.

∎

It took about 1900 seconds as a whole for the above computation in the same environment as Table 1. More than 95% of the CPU time was used for computing Gröbner bases of $I, \tilde{I}$ and $\tilde{I'}$ in steps 1, 3 and 4 in total. Since this formulation is based on a rather naive way to decompose an ideal, its improvement should be considered for a future work.

4 Concluding Remarks

Through all the experiments, we find that the following 9 among Chou's 512 theorems are essentially difficult to compute their Gröbner bases by any means in a moderate computational environment at present.

Ex.6,7,10,11,12	Pascal's theorem and related ones
Ex.13	Steiner's theorem
Ex.14	Kirkman's theorem
Ex.19	Brianchon's theorem (The dual of Pascal's theorem)
Ex.80	Theorem of Pratt-Wu

Except for Pratt-Wu, 8 of the 9 theorems are related to Pascal's theorem (figures constructed from 6 points on a conic). Consequently, these figures yield rather complicated polynomial systems with more variables and parameters than the other solvable 26 systems. Therefore, it seems still difficult to compute Gröbner bases with rational expression coefficient $\mathbf{Q}(u_1, \ldots, u_m)$ for such systems.

Finally, we itemize the remarks on our present results.

(1) Formulae (a) and (b) in Proposition 1 are comparable. As shown in Table 2, it is usually faster to compute the Gröbner basis of $(f_1, \ldots, f_\ell, 1 - yg)$ directly. However, we should confirm $(f_1, \ldots, f_j, \ldots, f_\ell) \neq (1)$ at first, because we may have $(f_1, \ldots, \tilde{f}_j, \ldots, f_\ell) = (1)$ by some mistakes during the translation.

(2) If we clear the common denominator and compute in $\mathbf{Q}[u_i][x_i]$, then intermediate expressions explode seriously. The total number $(m+n)$ of variables has severe influence, and the computation of Gröbner basis becomes much more difficult by reverse effect.

(3) It is not known yet how efficiently new algorithms such as F_4 [5], F_5 [6] work in the rational expression coefficient cases $\mathbf{Q}(u_i)[x_i]$. In Maple11, the option `method=maplef4` is usually faster but requires more memory space than the option `method=buchberger`.

Acknowledgements We are grateful to Ms. R.Kikuchi for permitting us to convert her original program written in Reduce.

References

1. Bazzotti, L., Dalzotto, G., and Robbiano, L.: Remarks on Geometric Theorem Proving, *Automated Deduction in Geometry 2000* (Richter-Gebert, J. and Wang, D., eds.), *LNAI*, **2061**, Zurich, Springer, 2001, 104–128.
2. Buchberger, B.: *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*, PhD thesis, Universität Innsbruck, 1965.
3. Chou, S.-C.: *Mechanical Geometry Theorem Proving*, D.Reidel, Dordrecht, 1988.
4. Cox, D., Little, J., and O'Shea, D.: *Ideals, Varieties, and Algorithms (2nd ed.)*, Springer, N.Y., 1997.

5. Faugère, J.-C.: A New Efficient Algorithm for Computing Gröbner Bases (F_4), *J. Pure and Applied Algebra*, **139**, 1999, 61–88.
6. Faugère, J.-C.: A New Efficient Algorithm for Computing Gröbner Bases without Reduction to Zero (F_5), *ISSAC 2002* (Mora, T., ed.), Lille, ACM, 2002, 75–83.
7. Gräbe, H.-G.: The SymbolicData Project, http://www.symbolicdata.org/, 2000–2006.
8. Hearn, A. C.: *Reduce User's Manual (Ver. 3.6)*, RAND Corp., Santa Monica, 1995.
9. Kapur, D.: Using Gröbner Bases to Reason About Geometry Problems, *J.Symbolic Computation*, **2**(4), 1986, 399–408.
10. Kutzler, B. and Stifter, S.: On the Application of Buchberger's Algorithm to Automated Geometry Theorem Proving, *J.Symbolic Computation*, **2**(4), 1986, 389–397.
11. Maplesoft: *Maple 10 User Manual*, Maplesoft, Tokyo, 2005. (in Japanese).
12. Maplesoft: *Maple 11 User Manual*, Maplesoft, Tokyo, 2007. (in Japanese).
13. Moritsugu, S. and Arai, C.: On the Efficiency of Geometry Theorem Proving by Gröbner Bases, *Trans.Japan Soc.Indust.Appl.Math.*, **17**(2), 2007. (to appear; in Japanese).
14. Noro, M. and Takeshima, T.: Risa/Asir - A Computer Algebra System, *ISSAC '92* (Wang, P., ed.), Berkeley, ACM, 1992, 387–396.
15. Recio, T. and Vélez, M. P.: Automatic Discovery of Theorems in Elementary Geometry, *J. of Automated Reasoning*, **23**(1), 1999, 63–82.
16. Wang, D.: An Elimination Method for Polynomial Systems, *J.Symbolic Computation*, **16**(2), 1993, 83–114.
17. Wang, D.: Geometry Machines: From AI to SMC, *AISMC 3* (Calmet, J., Campbell, J. A., and Pfalzgraf, J., eds.), *LNCS*, **1138**, Steyr, Springer, 1996, 213–239.
18. Wang, D.: Gröbner Bases Applied to Geometric Theorem Proving and Discovering, *Gröbner Bases and Applications* (Buchberger, B. and Winkler, F., eds.), *London Mathematical Society Lecture Note Series*, **251**, Cambridge Univ. Press, Cambridge, 1998, 281–301.
19. Wang, D.: *Elimination Practice: Software Tools and Applications*, Imperial College Press, London, 2004.
20. Winkler, F.: A Geometrical Decision Algorithm Based on the Gröbner Bases Algorithm, *ISSAC '88* (Gianni, P., ed.), *LNCS*, **358**, Rome, Springer, 1988, 356–363.
21. Wu, W.-T.: On the decision problem and the mechanization of theorem-proving in elementary geometry, *Automated Theorem Proving: After 25 Years* (Bledsoe, W. and Loveland, D., eds.), *Contemporary Mathematics*, **29**, AMS, Providence, 1983, 213–234.

A Document-Oriented Coq Plugin for TEXmacs

Lionel Elie Mamane and Herman Geuvers

ICIS, Radboud University Nijmegen, NL

Abstract. We discuss the integration of the *authoring* of a mathematical document with the *formalisation* of the mathematics contained in that document. To achieve this we are developing a Coq plugin for the TEXMACS scientific editor, called tmEgg. TEXMACS allows the wysiwyg editing of mathematical documents, much in the style of LATEX. Our plugin allows to integrate into a TEXMACS document mathematics formalised in the Coq proof assistant: formal definitions, lemmas and proofs. The plugin is still undergoing active development and improvement.
As opposed to what is usual for TEXMACS plugins, tmEgg focuses on a *document consistent* model of interaction. This means that a Coq command is evaluated in a context defined by other Coq commands in the document. In contrast, TEXMACS plugins usually use a *temporal* model of interaction, where commands are evaluated in the order (in time) of the user requests. We will explain this distinction in more detail in the paper.
Furhermore, Coq proofs that have been done using tmEgg are stored completely in the document, so they can be browsed without running Coq.

1 Introduction

TEXMACS [1] is a tool for editing mathematical documents in a wysiwyg style. The input an author types is close to LATEX, but the output is rendered on screen in real time as it will be on paper. TEXMACS supports *structure editing* and it stores the files in a structured way using *tags*, which is close to XML. So, a TEXMACS document is a labelled tree. The labels (tags) provide information that can be used as *content* or *display* information. For a specific label, the user can choose a specific way of rendering the subtrees under a node with that label, for example rendering all subtrees in math mode. But a user may also choose a specific action for the subtrees, for example sending the subtrees as commands to the computer algebra package Maple. Of course, many labels are predefined, like in LATEX, so a user is not starting from scratch.

TEXMACS facilitates interaction with other applications: within TEXMACS one can open a "session", for example a Maple session, and then input text within that session is sent to a Maple process that is running in the background. The Maple output is input to the TEXMACS document, and rendered accordingly. In this way, TEXMACS can be used as an interface for Maple, with the additional possibility to add text or mathematical formulas around the Maple session, creating a kind

of *interactive mathematical document*. Here the interaction lies in the possibility to execute parts of the document in the background application.

In this paper we present tmEgg, a Coq plugin for TEXMACS. The plugin allows the user to call Coq from within a TEXMACS document, yielding a TEXMACS document interleaved with Coq sessions. It also provides special commands for Coq, like stating a definition or a lemma. The plugin does not provide its own proof language, but leverages any proof language that Coq understands or will understand in the future, such as [2]. This means that when doing a proof, the user types actual Coq commands (usually tactics) in the TEXMACS document, which are then sent to Coq as-is and the Coq output is rendered by TEXMACS. This is in contrast with the approach of e.g. [3], [4] or [5], that seek to change the way a proof is written or the way a user interface interacts with the prover.

A crucial aspect of the plugin is that it views the sequence of Coq sessions within a document as one Coq file. So, when one opens a document and executes a command within a Coq session, first all *previous* Coq commands are executed and the present command is then executed in the Coq state thus obtained. So the TEXMACS document as a whole also constitutes a valid Coq development. Additionally, tmEgg automatically reexecutes any command that is modified; no command is locked and unmodifiable.

From the Coq perspective, one can thus see the TEXMACS document as a documentation of the underlying Coq file. Using TEXMACS, one adds pretty printed versions of the definitions and lemmas. The plugin further supports this by a folding (hiding) mechanism: a lemma statement has a folded version, showing only the pretty printed (standard mathematical) statement of the lemma, and an unfolded version, showing also the Coq statement of the lemma. A further unfolding also shows the Coq proof of the lemma.

Altogether there are four ways of seeing the tmEgg TEXMACS plugin. These are not disjoint or orthogonal, but it is good to distinguish them and to consider the various requirements that they impose upon our plugin.

A Coq interface. One can call Coq from within TEXMACS, thus providing an interface to Coq. When the user presses the return key in a Coq interaction field, the Coq commands in this field are sent to Coq and Coq returns the result to TEXMACS. The plugin doesn't do any pretty printing of Coq output (yet), but it allows to save a Coq development as a TEXMACS file which can be replayed.

A documented Coq formalisation. A Coq formalisation usually has explanatory comments to give intuitions of the definitions, lemmas and proofs or to give a mathematical (e.g. in LATEX) explanation of the formal Coq code. The plugin can be used for doing just that: the traditional TEXMACS elements are used for commenting the underlying Coq file. In this respect, tmEgg can play the same role as Coqdoc [6], but goes beyong this. Coqdoc extracts document snippets from specially formatted comments in Coq scripts and creates an HTML or LATEX document containing these snippets and the vernacular statements with some basic pretty-printing of terms. In Coqdoc, there is no Coq interaction possible

from within this HTML or LaTeX document. tmEgg enables the user to have a mathematical document (in TeXmacs), whose formal definitions and proofs can also be executed in Coq. Moreover, the formal proofs can also be read without Coq, because the full Coq interaction was stored within the document at the time it was created.

Taking this use case to its extreme, one arrives at a notion of *literate proving*, by analogy to literate programming: a system that allows to write formal definitions and proofs in one document together with their (high-level) mathematical documentation.

A document with a Coq formalisation underneath. One can write a mathematical article in TeXmacs, like one does in LaTeX. With tmEgg, one can take a mathematical article and extend it with formal statements and proofs. Due to the folding mechanism, the "view" of the article where everything is folded can be the original article one started with. It should be noted that, if one adds a Coq formalisation underneath this, not everything needs to be formalised: lemmas can be left unproven etc., as long as the Coq file is *consistent*, i.e. no notions are used unless they are defined. In this sense, tmEgg makes a step in the direction of the Formal Proof Sketches idea of [7].

Course notes with formal definitions and proofs. We can use the TeXmacs document for course notes (handouts made by the teacher for students). An added value of our plugin is that we have formal definitions and proofs underneath, but we don't expect that to be a very appealing feature for students. On the other hand, we also have full access to Coq, so we can have exercises that are to be done with Coq, like "prove this statement" or "define this concept such that such and such property holds". This is comparable in its intent to ActiveMath [8].

In the following we present our plugin tmEgg, including some technical details and a fragment of a TeXmacs document with underlying Coq formalisation. We will discuss the four views on the plugin as mentioned above in detail. An essential difference between the tmEgg Coq plugin that we have created and other TeXmacs plugins, e.g. the one for Maple, is that we take a *document oriented* approach. This we will describe first.

2 The document-consistent model

The TeXmacs plugins to computer algebra or proof systems usually obey a temporal model of interaction, that is, the expressions given to the plugin by the user are evaluated in the chronological order the user asks for their evaluation, irrespective of their relative position in the document and dependencies. In other words, the TeXmacs plugin system ignores the fact that the interpreter it is interfacing with has an internal state which is modified by the commands TeXmacs

gives it. This can lead to the result of a command in the current session to be irreproducible in later sessions because the sequence of commands leading to the state in which the interpreter was when evaluating the command is lost. See Fig. 1 for an example, with the CAS Axiom. The user first assigns the value 5 to `a`, and asks for the value of `a`. The correct answer is given. The user then redefines a to be 6 and goes back up to the command `a` and asks for its reexecution. The answer given is 6, which corresponds to the chronological order of execution of the commands, but not to the order in which the said commands are read by a somebody that hasn't seen the chronological history. While in that simple case, one may guess what has happened, if the user deletes the assignation of 6 of `a` or even both definitions (third row in the figure), the explanation is gone, and the behaviour of TEXMACS and Axiom is seemingly unexplainable to someone that walks in at that moment and finds TEXMACS and Axiom in that state. If the document is saved and reloaded, one will not get the same results again.

Contrast with Fig. 2, showing a tmEgg Coq session. `Empty_set` is predefined in Coq's standard library, and gets redefined in the second command. However, independently of the order in which the user asks for evaluation of the commands, it will always give the same result, shown in the figure. E.g. if the user asks for evaluation of the second command (defining `Empty_set` to be 5) and then asks for the evaluation of the first one, the first command will always answer "`Empty_set` is an inductively defined type of sort `Set` without any constructor", not "`Empty_set` is 5". Similarly, if the user opens the document and evaluates the third command, it will answer `Empty_set = 5` because the second command will have been automatically executed before the third one.

The risk of inconsistency brought by the temporal model is naturally even more undesirable in the context of writing formal mathematics, leading to a *document-consistent* model of interaction: a statement is always evaluated in the context defined by evaluating all statements before it in the document, in document order, starting from a blank state.

2.1 Implementation

Coq 8.1 thankfully provides the features essential for implementing the document-consistent model, in the form of a backtrack command that can restore the state to a past point B. It works under the condition that no object (definition, lemma, ...) whose definition is currently finished was incomplete at point B. If this condition is not satisfied, tmEgg backtracks up to a point before B where this condition does hold and then replays the statements between that point and B. This condition always holds somewhere at or before B: it holds at the very beginning of the document, where no definition is started.

The arguments given to the backtrack command are derived from state information that Coq gives after completion of each command, in the prompt. tmEgg stores the information on the Coq state *before* a command as a *state marker* next to the command itself, that is a document subtree whose rendering is the empty string. This state information consists (roughly speaking) of the number of def-

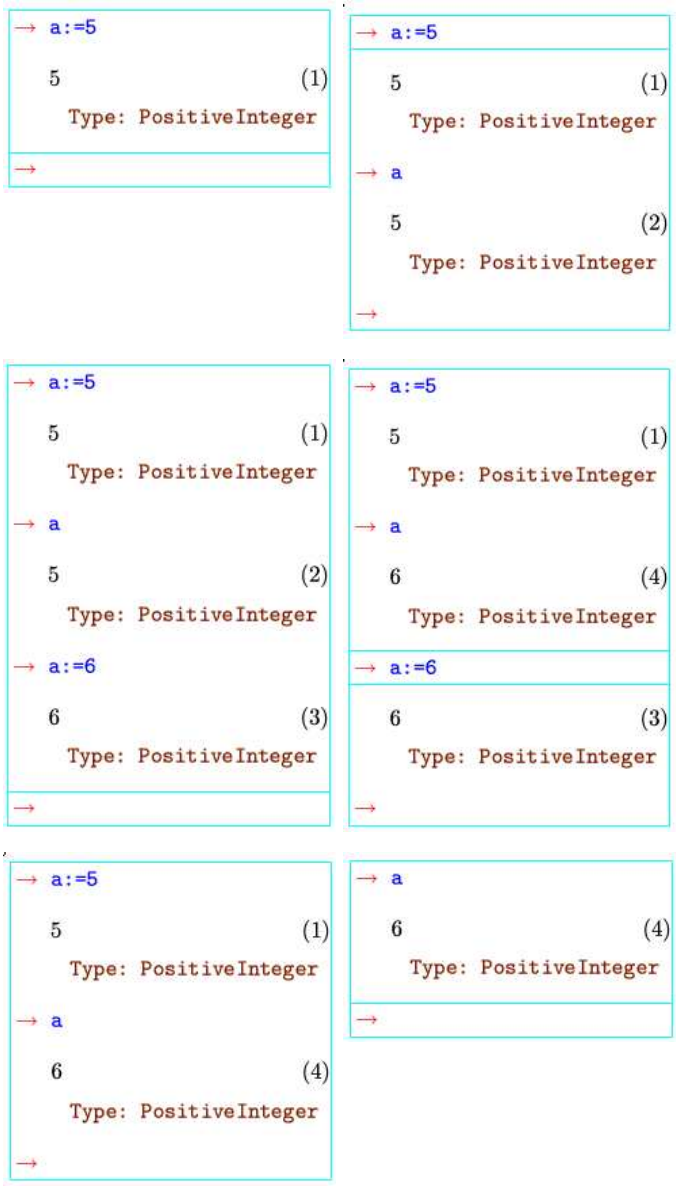


Fig. 1. Example of inconsistent output

initions made in the current session, the list of unfinished definitions and the number of steps made in the current unfinished definition, if any.

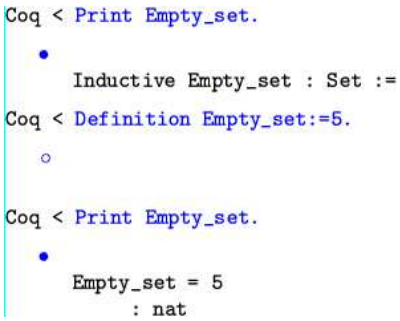


Fig. 2. Example of consistent output

tmEgg also keeps a copy in memory of the Coq commands that have been executed; when the user asks for evaluation of a Coq command, tmEgg checks whether an already executed command was modified (respectively deleted, or a new one inserted between already executed commands) in the document since its execution, and if any was, automatically reexecutes it.

2.2 A better model

The underlying model is that the document contains a sequence of Coq commands to be evaluated exactly in that order. This model will be familiar to users of other popular interfaces, such as Proof General/Emacs and CoqIDE, where one edits directly a "`.v` file", that is a text file made of a concatenation of Coq commands. The tmEgg document is just a superset of that, that contains both Coq commands and other document snippets that get ignored by Coq.

This presents the restriction that the definition of an object (e.g. a lemma) has to precede any use of it in the document. This forces the order of presentation of objects in the document to be a valid order in the formal logical meaning. While this is considered a feature by overly formalist people (such as one of the authors), it is considered a hindrance for writing documents optimised for reading by the rest of humanity. Indeed, the author of a document may e.g. consider it better to first present the main theorem, making liberal use of lemmas that are best read *after* understanding their role in the overall picture. He may even wish to banish an uninteresting technical lemma to an appendix.

Also, from a performance point of view, if an object `T` is followed by several objects that do not use `T` and then one object `S` that does use `T`, changing the definition of `T` will lead to unnecessarily cancelling and redoing the definitions of the intermediary objects that are guaranteed not to be affected by a change in `T`; only `S` needs to be cancelled. A similar situation arises when the user works on several unfinished definitions in a temporally interleaved way; the already executed steps of the objects placed lower in the document will constantly be cancelled and reexecuted, for no good reason.

In order to better accommodate these usage scenarios, a future version of tmEgg will have a different model: the document will be seen as containing a set of Coq objects (definitions, lemmas, theorems, ...). When the user opens a document and asks for reexecution of the definition of an object `A`, all the objects necessary for `A`, but no more, are redefined, irrespective of their position in the document. Similarly, if `A` is changed, only the objects using it will have their definition removed from the Coq session, not all those that happen to be defined later in the document.

Furthermore, in this model, if the user jumps between two unfinished definitions, there is no need to abort either of them; they can be simply suspended and resumed, without cancelling proof steps that don't need to be.

However, the proof script of one particular proof will - at least in a first version - still be considered as a strictly linear sequence.

Coq makes that model easier to implement than other systems. Indeed, Coq does not allow any redefinition[1]. Any document will thus have only *one* definition of any (fully qualified) name, and there will be no ambiguity on which definition of `B` shall be used to define `A`, if the definition of `A` uses `B`. tmEgg can then store the dependencies (hidden) in the document at the time a definition is finished.

3 Presentation of tmEgg

tmEgg extends TeXmacs with Coq interaction fields. The user enters Coq commands in one of these fields and presses enter to have the command executed. Coq's answer is placed below the input field in the document itself. One can naturally freely interleave Coq interaction fields with usual document constructs, permitting one to interleave the formal mathematics in Coq and their presentation in LaTeX-level mathematics or comments about the formalisation. Each Coq interaction can be folded away at the press of a button, as well as each specific result of a command individually. The output of the previous command is automatically folded upon evaluation of a following command. See Fig. 3 for an example: The empty circles indicate a folded part and can be clicked to unfold that part, and the full circles indicate a foldable unfolded part and can be clicked to fold it. Here, the formal counterpart to hypothesis 2 is completely folded, while the statement of lemma 3 is unfolded and its proof folded. The proof of lemma 4 is unfolded, but the result of most of its steps is folded.

Note that the result of each Coq command is inserted into the document statically (and replaced upon reevaluation), just after the command itself, before the next command; this means that they can be copied and pasted like any part of the document, but also that the saved file contains them, so that the development can be followed without running Coq, a potentially lengthy operation. As a

[1] Fig. 1 may seem to be a counter-example to this assertion, but it is not: What happens here is merely shadowing of the library definition by one in the current namespace, but this affects only the unqualified name `Empty_set`. The library object is still available under its fully qualified name, namely `Coq.Init.Datatypes.Empty_set`.

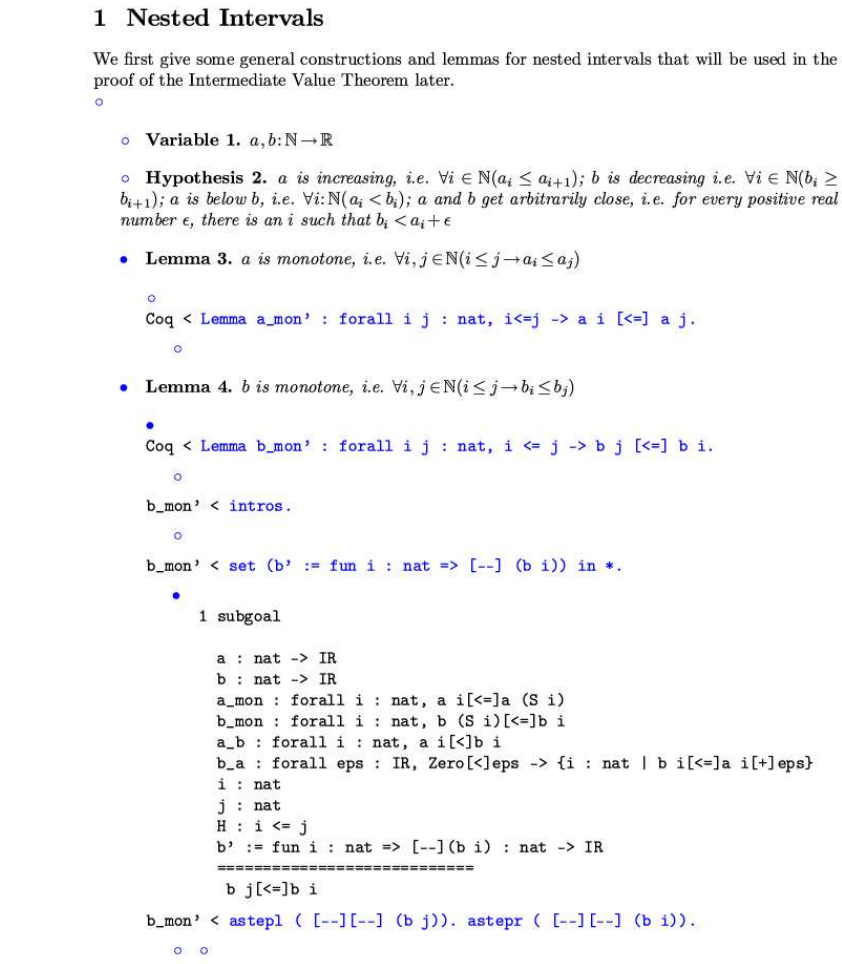


Fig. 3. tmEgg screenshot

corollary, the development can even be followed (but not independently checked) on a computer lacking Coq.

This choice of placing the Coq output visibly in the document itself was partly an experiment; traditionally the user interfaces place the prover's answer/state in a fixed-size reserved area of the screen. Interleaving the Coq output with its input has proven well suited to small toy examples. Mainly, it avoids having to constantly switch eye focus between the separate edition area and the Coq output area, leading to a smoother experience. It also has the advantage that it permits having several consecutive Coq outputs on screen simultaneously, making comparing them easier. This is especially useful when reading a proof, when one is trying to figure out what a Coq command is doing.

However, it has proven unpopular with users doing bigger proofs, mainly because it is not as spill-resistant as a fixed-size reserved area when the proof state reaches a moderate size or because it "clutters up the screen".

As both approaches have inherent advantages, future versions of tmEgg will support both approaches. Coq output will be saved in the document, but can be completely hidden globally. A separate window, which can be shown or hidden, will contain the output corresponding to the current Coq command.

In order to help the user create the proposed "formal and informal version of the same mathematics" structure (particularly in the "mathematical document with a Coq formalisation underneath" scenario), we present him with a menu where he can choose a Coq statement type (such as Lemma, Hypothesis, Definition, ...) and that will create an empty template to fill made of:

- the corresponding TEXMACS theorem-like environment for the informal statement;
- a foldable Coq interaction field for the formal statement;
- a foldable Coq interaction field for the formal proof, if appropriate;

This is illustrated in Fig. 4.

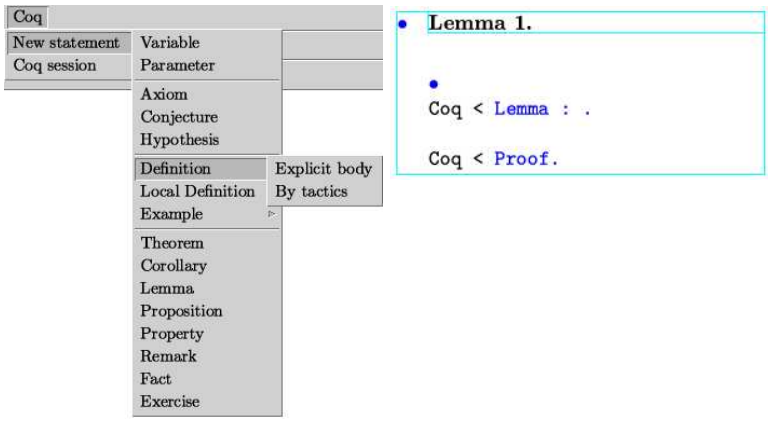


Fig. 4. New statement menu, empty lemma structure

3.1 Architecture

We have decided to avoid putting TEXMACS-specific code in Coq. That's why, rather than adapt Coq to speak the TEXMACS plugin protocol by itself, we have implemented a wrapper in OCaml that translates from Coq to TEXMACS (see Fig. 5). We try to keep that wrapper as simple and stateless as possible, putting most of the intelligence of the plugin in Scheme in TEXMACS.

Fig. 5. tmEgg architecture

3.2 Adaptations in Coq for tmEgg

However, a few generic enhancements to Coq were necessary:

- One could not backtrack into a finished section (that is, from a point where this section was finished to a point where it is unfinished). This is now possible.
- There are two protocols to interact with Coq: the "emacs" protocol and the "Pcoq" protocol. The Pcoq protocol has the huge advantage of clearly stating which of the commands you gave to Coq failed or succeeded, while the emacs protocol leaves you to carefully parse the output to see whether there is an error message contained in it. On the other hand, the Pcoq protocol was tied to a different term printer than the one usual to Coq users and a different history management model than the one described above.
  We have untied the term printer and communication protocol, so that either printer can be used with either protocol, allowed disabling the Pcoq history management mechanism and added the backtracking state information of the emacs protocol to the Pcoq protocol. This allows us to use a robust communication protocol (the Pcoq one), while still displaying terms in the same syntax the users can type them in and leveraging the backtrack command.

4 How well does the plugin do?

In the introduction, we have described four views (possible applications) on the tmEgg plugin. We now want to discuss to which extent the plugin satisfies the requirements for each of those views.

A Coq interface. One can do Coq from within a TEXMACS document using our plugin, if one has the patience or a machine fast enough to put up with TEXMACS's slowness. However, as detailed above, compared to well-known interfaces like Proof General [9] and CoqIde [6], the display of the proof state *inside* the document can be a disadvantage. Other things that our plugin does not (yet) support but are in principle possible to add in TEXMACS are: menus for special tactics and pretty printing (but Proof General and CoqIde don't have this either). Pretty printing is of course interesting to add in the context of TEXMACS, because it has various LATEX-like facilities to add it. However, it should be noted that, if we want to use our plugin as an interface for Coq, the syntax should be accepted as *input* syntax too, so as to not confuse the user. The user may also prefer to use the default Coq pure text syntax rather than graphical mathematical notations; this will always be supported.

Compared to traditional user interfaces, tmEgg has the advantage that one can scroll to any point in the proof script and reexamine Coq's state. One can then always edit the Coq command there freely, and tmEgg will do whatever is necessary to make Coq aware of that. Traditional user interfaces lock already executed commands, that is they cannot be edited.

A documented Coq formalisation. As a documentation tool, the plugin works fine. One can easily add high level mathematical explanations. One can import a complete uncommented Coq file and start adding annotations. It would be better if existing Coq comments, in particular Coqdoc annotations, were imported and converted to TEXMACS document snippets, but this is not implemented yet. Note however that there is no (formal) link between the formal Coq and the high level explanation in TEXMACS, because the high level translation is not a translation of the Coq code, but added by a human. This is different from, e.g. the work in the Mowgli [10] project, where we have a high level rendering of the formal Coq statements.

A document with a Coq formalisation underneath. This is a way the plugin can be used now. One would probably want to hide even more details, so more folding would be desirable, e.g. folding a whole series of lemmas into one "main lemma" which is the conclusion of that series. Thus one would be able to create a higher level of abstraction that is usual in mathematical documents. Of course this can already be done in TEXMACS, but our plugin does not specifically propose it automatically. If such nested folding were added, it would also be advisable to be able to display the "folding structure" separately, to give the high level structure of the document.

Course notes with formal definitions and proofs. In general, proof assistants are tools that require quite some maturity to be used, so therefore we don't expect students to easily make an exercise in their TEXMACS course notes using the underlying proof assistant Coq, i.e. as an exercise in the mathematics studied rather than as an exercise in Coq. This situation may improve in the future though, depending on the maturity of proof assistant technology. It should also be noted that the plugin does not (yet) explain/render the Coq formalised proofs, like e.g. the Helm tool [11] does (by translating a formal proof into a mathematically readable proof). See also [12].

5 More Future Outlooks

5.1 Mathematical input/output

Current TEXMACS interfaces to computer algebra systems include conversion to and from mathematical notations (see Fig. 6). Doing the same with Coq brings some difficulties in a more acute way than with a CAS:

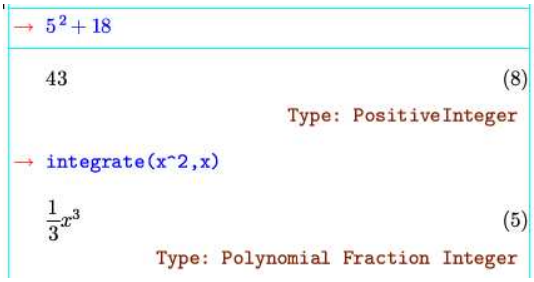


Fig. 6. Mathematical notation input/output with Axiom

- Different developments will call for the same notation to map to different Coq objects; there are for example several different real numbers implementations for Coq.
- Similarly, the best notation to use for the same Coq construct will vary depending on the document, where in the document one is, or even more subtle factors. A prime example of this is parentheses around associative operators: One usually doesn't want a full parenthesising in statements, but if one always leaves out "unnecessary" parentheses, the statement of the associativity lemma itself looks quite pointless, as do the proof steps consisting of applying the associativity lemma.
- Some Coq constructs (such as some ways to define division) need information that is not part of usual mathematical notation (such as proof that the divisor is not zero).

Ideally, the notations would thus probably have to be highly dynamic; if making good choices automatically proves impossible, maybe a good compromise will be to let the author of the document choose on a case-by-case basis. What can be achieved sanely is still to be explored.

Once at least the conversion *to* mathematical notation is satisfying, we can make a TEXMACS command that takes a Coq term (or the name of one) and whose rendering is the "nice" mathematical rendering for that term. This means that users will be able to put Coq terms in their documents and have them look like LATEX-level mathematics.

This conversion from and to "normal" mathematical notation might also form a usable mechanism for informal and unsafe exchange of terms between different computer algebra systems and proof assistants. E.g. if the Coq goal to prove is $x^{18} - 5x^7 + 5 = 0 \to x > 2$, the user could select in the goal the expression $x^{18} - 5x^7 + 5 = 0$ (duly converted from Coq term to mathematical notation by tmEgg), paste it into a CAS session and ask the CAS to solve that equation (where the TEXMACS-CAS integration plugin will duly convert it to the syntax of the CAS being used) to quickly check whether the goal is provable, or use the CAS as an oracle to find the roots and use knowledge of the roots to make the proof easier to write.

It was originally planned to use the Pcoq term printer to get the Coq terms as pure λ-term trees, and handle all the transformation to TEX-level presentational notations in tmEgg itself, e.g. through mapping Coq terms to TEX_{MACS} document macros. This would have allowed to easily use different notations in (different places of) different documents, but it means loosing the ability to look at the type of a term to make a presentation decision. In consultation with the Coq team, we finally decided we will add a term pretty-printer to TEX_{MACS} syntax in Coq itself, sharing most infrastructure with the existing Coq ASCII/Unicode text term printer.

5.2 Miscellaneous

Once the basic framework of tmEgg has matured and works well, all kinds of small, but highly useful, features can be imagined:

- Import of Coq files containing Coqdoc document snippets, leveraging the LATEX import of TEX_{MACS}.
- Automatic generation of table of Coq constructs in the document and corresponding index.
- Similarly, menu command to jump to the definition of a particular Coq object.
- Make any place where a Coq object (e.g. a lemma) is used a hyperlink to its definition. This could even eventually be expanded up to making tmEgg a Coq library browser.

References

1. van der Hoeven, J.: GNU TEX_{MACS}. SIGSAM Bull. **38**(1) (2004) 24–25
2. Corbineau, P.: Declarative proof language for coq. http://www.cs.ru.nl/ corbinea/mmode.html (2006)
3. Théry, L.: Formal proof authoring: an experiment. In Lüth, C., Aspinall, D., eds.: UITP2003 International Workshop on User Interfaces for Theorem Provers, informal proceedings. Volume 189 of Technical Report., Institut für Informatik Albert-Ludwigs-Universität Freiburg, Aracne (2003) 143–159
4. Dixon, L., Fleuriot, J.: A proof-centric approach to mathematical assistants. Journal of Applied Logic: Special Issue on Mathematics Assistance Systems (2005) 35 To be published.
5. Aspinall, D., Lüth, C., Wolff, B.: Assisted proof document authoring. In Kohlhase, M., ed.: MKM 2005, Mathematical Knowledge Management: 4th International Conference. Volume 3863 of Lecture Notes in Computer Science., Springer Verlag (2006) 65–80
6. The Coq Development Team: The Coq Proof Assistant Reference Manual. (LogiCal Project - INRIA Futurs)
7. Wiedijk, F.: Formal proof sketches. In Berardi, S., Coppo, M., Damiani, F., eds.: Types for Proofs and Programs: Third International Workshop, TYPES 2003, Torino, Italy. Volume 3085 of LNCS., Springer (2004) 378–393

8. Melis, E., Andres, E., Büdenbender, J., Frischauf, A., Goduadze, G., Libbrecht, P., Pollet, M., Ullrich, C.: ActiveMath: A generic and adaptive web-based learning environment. Artifical Intelligence and Education **12**(4) (2001)
9. Aspinall, D.: Proof general - a generic tool for proof development. In S. Graf, M.S., ed.: TACAS 2000. Volume 1785 of LNCS. (2000)
10. Asperti, A., Wegner, B.: MoWGLI - a new approach for the content description in digital documents. In: Proceedings of the Ninth International Conference on Electronic Resources and the Social Role of Libraries in the Future. Volume 1., Autonomous Republic of Crimea (2002) (Section 4).
11. Asperti, A., Padovani, L., Coen, C.S., Guidi, F., Schena, I.: Mathematical knowledge management in HELM. Annals of Mathematics and Artificial Intelligence, Special Issue on Mathematical Knowledge Management **38**(1-3) (2003) 27–46
12. Asperti, A., Geuvers, H., Loeb, I., Mamane, L.E., Coen, C.S.: An interactive algebra course with formalised proofs and definitions. In Kohlhase, M., ed.: Mathematical Knowledge Management: 4th International Conference, MKM 2005, Bremen, Germany. Volume 3863 of Lecture Notes in Computer Science., Springer Verlag (2006) 315–329
13. Audebaud, P., Rideau, L.: TEX$_{\text{MACS}}$ as authoring tool for formal developments. In Aspinall, D., Lüth, C., eds.: Proceedings of the User Interfaces for Theorem Provers Workshop, UITP 2003. Volume 103 of Electronic Notes in Theoretical Computer Science., Rome, Italy, Elsevier (2004) 27–48
14. INRIA Sophia-Antipolis Lemme Team: PCoq, a graphical user-interface for Coq. (http://www-sop.inria.fr/lemme/pcoq/)

Software Specification Using Tabular Expressions and OMDoc*

Dennis K. Peters[1], Mark Lawford[2], and Baltasar Trancón y Widemann[3]

[1] Electrical and Computer Engineering
Faculty of Engineering and Applied Science
Memorial University of Newfoundland
St. John's, Newfoundland Canada
dpeters@engr.mun.ca
http://www.engr.mun.ca/~dpeters
[2] Dept. of Computing and Software
Faculty of Engineering, McMaster University
Hamilton, Ontario, Canada
lawford@mcmaster.ca
http://www.cas.mcmaster.ca/~lawford
[3] Software Quality Research Laboratory
Computer Science and Information Systems Building
University of Limerick, Limerick, Ireland
Baltasar.Trancon@ul.ie
http://www.sqrl.ul.ie

Abstract. Precise specifications or descriptions of software system behaviour often involve fairly complex mathematical expressions. Research has shown that these expressions can be effectively presented as tabular expressions, but that tool support is needed to do this well. Traditional documentation tools (e.g., plain text or word processors) are insufficient because they do not i) have good support for mathematical expressions, particularly in tabular forms, and ii) retain sufficient semantic information about the expressions to permit the use of tools such as automated reasoning systems, and code or oracle generators. This paper presents initial work in the development of a suite of tools, using OMDoc as an exchange medium, for development, analysis and use of tabular software specifications. It shows by some simple examples how these tools can work together with other systems to add significant value to the documentation process.

1 Software Specifications

Researchers in the area that has come to be called "software engineering" have, over the years, proposed many techniques for documenting required or actual behaviour and designs for software based systems. Despite the purported benefits

* This work was carried out while all three authors were at the Software Quality Research Laboratory, University of Limerick.

of these techniques with respect to the quality of the software produced, very few of these have found widespread use in industrial software practice. It is suggested that developers are reluctant to use these techniques because they are not seen to add enough value to the development process to justify the effort required to produce and maintain the documentation. In this work we hope to improve this situation by developing tools that support both the production and maintenance of good documentation and the application of this documentation to such tasks as design analysis, verification and testing.

Some of our goals for the tools that we are developing are that they should:

- Free authors from devoting inappropriate effort to the presentation details of the document – the effort should be focused on the content.
- Assist the authors to avoid typographical mistakes, for example through content assist techniques similar to those found in integrated development environments.
- Support checking of consistency both within a document (self consistency) and between documents, including code where appropriate.
- Assist in design analysis and verification, possibly using tools such as proof systems, model checkers, or computer algebra systems.
- Support automated specification based testing, for example by test case and oracle generation.

To achieve these goals the documentation being produced must be in a form that has a precisely defined syntax and semantics – that is, it must be *formal* – and it must be in a form that enables access to the semantic content. Such formal documentation techniques usually make use of a substantial amount of reasonably complicated mathematics for which general purpose documentation production tools (e.g., word processing software) are less than ideal because they focus on the presentation of the information, rather than its semantic content. The mathematical content markup language OMDoc[1] addresses this problem and serves as a basis on which to build our tools.

1.1 Tabular Expressions

The nature of computer system behaviour often is that the system must react to changes in its environment and behave differently under different circumstances. The result is that the mathematics describing this behaviour consists of a large number of conditions and cases that must be described. It has been recognized for some time that tables can be used to help in the effective presentation of such mathematics [2–5]. In this work we view such tabular representations of relations and functions as an important factor in making the documentation more readable, and so we have specialized our tools to support them [6–8].

A full discussion of tabular expressions is beyond the scope of this paper, so interested readers are referred to the cited publications. In their most basic form, tabular expressions represent conditional expressions, so for example, (in Janicki's style [7]) the function definition given in (1), could be represented by the tabular expression in (2).

$$\mathrm{f}(x,y) \stackrel{\mathrm{df}}{=} \begin{cases} x+y & \text{if } x>1 \wedge y<0 \\ x-y & \text{if } x\leq 1 \wedge y<0 \\ x & \text{if } x>1 \wedge y=0 \\ xy & \text{if } x\leq 1 \wedge y=0 \\ y & \text{if } x>1 \wedge y>0 \\ x/y & \text{if } x\leq 1 \wedge y>0 \end{cases} \tag{1}$$

$$\mathrm{f}(x,y) \stackrel{\mathrm{df}}{=} \tag{2}$$

	$x>1$	$x\leq 1$
$y<0$	$x+y$	$x-y$
$y=0$	x	xy
$y>0$	y	x/y

In OMDoc it is straightforward to add support for tabular expressions, simply by defining appropriate (OpenMath) symbols to denote them: we use a symbol for "table", which, following the model presented in [8], takes four argument expressions representing

1. The *evaluation term*, which expresses how the value of a tabular expression is defined in terms of the expressions in its grids. For (2) this expression would express that the value is that of the element of the central grid, $T[0]$, that is indexed by indices of the true elements of each of the "header" grids, $T[1]$ and $T[2]$, as follows: $T[0][\mathit{select}(T[1]), \mathit{select}(T[2])]$, where *select* is a function on a predicate grid that gives the index of the cell that is true.
2. The *static restriction*, which defines a condition that must be true of the grids, independent of the expressions in the grids, but possibly dependent on their types. This is used, for example, to assert the conditions on the number and size of the grids (i.e., the shape of the table). For (2) this would express that the index set of the central grid should be the power set of the index sets of the header grids, and that the header grids must contain predicate expressions.
3. The *dynamic restriction*, which defines a condition that must be true of the grid expressions. This is used to assert constraints on the table to ensure that it has a well defined meaning. For (2) this would assert than the header grids, $T[1]$ and $T[2]$, must be "proper" – only one cell expression should be true for any assignment.
4. A list of *grids*, which are indexed sets, represented by n-ary applications with symbol "grid" and taking pairs of cell index and cell contents as its arguments.

Figure 3 illustrates the OMDoc representation of a tabular expression.

Although (1) and (2) are clearly a nonsensical example, they are representative of the kind of conditional expression that occurs often in documentation of software based systems. We have found that the tabular form of the expressions is not only easier to read, but, perhaps more importantly, it is also easier to *write* correctly. Of particular importance is that they make it very clear what the cases are, and that all cases are considered.

Modern general purpose documentation tools, of course, have support for tables as part of the documents, but they are often not very good at dealing with tables as part of mathematical expressions. These tools also encourage authors to focus efforts on the wrong things: authors will work very hard to try to get the appearance of the table right, sometimes even to the detriment of readability (e.g., shortening variable names so that expressions fit in the columns).

One could argue that the two alternative presentations given in (1) and (2) are simply presentational styles and so should not be our focus, and we would have to agree to a point. As should be clear from the above discussion, however, our encoding of tabular expressions in OpenMath does not encode the presentational aspects other than implicitly in the symbol names – it simply defines new kinds of conditional (piecewise) expressions where the conditions are defined in indexed sets that we call grids. The symbols defined in the "piece1" standard OpenMath Content Dictionary[4] are not sufficient for our purposes since they group the conditions with the value expression, as in (1), rather than along other dimensions. The latter form improves readability and allows for clear expression of "properness" constraints (e.g., that the expressions in a grid must cover all cases and not overlap).

1.2 Classes of Documents

Although tabular expressions could be useful in many forms of documentation, our particular emphasis has been on documents that either specify or describe behaviour of software entities using relations on the quantities that are input and output from the component [9]. Rather than define a specification language, *per se*, we use standard mathematics together with some special functions or notations that are particular to the kind of document and are defined using standard mathematics [10]. The following are the particular kinds of documents that we are targeting.

System or Software Requirements documents define the required or actual behaviour of an entity by giving the acceptable values of the "controlled" quantities (outputs) at any time in terms of the history and current value of the "monitored" quantities (inputs) [11–13].

Module interface documents define the required behaviour of a software module (component) by giving the values of all output variables in terms of the sequence of past program calls, events and outputs of that module [14].

Module internal design documents describe the internal design of a module by identifying the data structure used, giving the abstraction relation that relates this data structure to the module interface specification, and defining the relations on values of the data structure and output variables before and after a call to an access program [15].

Note that our documents are not documents *about* mathematics, but rather make use of mathematics as a means to communicate. Also note that our documents will not normally include proofs but may be used as input to proof

[4] http://www.openmath.org/cd/piece1.xhtml

systems, as illustrated in section 2.2, for example to reason about the properties of a design.

1.3 Specification Document Model

A review of the contents of the above document types leads us to propose a document model consisting of the following elements.

Theory is the main structural element of our documents. Each document will contain one or more theories. Theories may include sub-theories either directly or via import references.

Symbols represent constants, variables, relations, functions or types. A specification document fundamentally is about identifying the symbols that are relevant and, where appropriate, defining their value in terms of other symbols.

Types declare the mathematical type of a symbol.

Definitions declare the meaning of a symbol (e.g., an expression describing the relation).

Code is unparsed formal text that, although it doesn't play a role in the documents we have mentioned, is likely to be needed for some documents.

Text is unparsed informal text that is included for readability of the document.

Readers familiar with OMDoc will recognize the above elements and see that our documents clearly fit within the OMDoc model. We have found, however, that the standard OMDoc attributes are insufficient for our purposes, so we have added a few that are specific to this project and have identified these by a namespace for the project (http://www.fillmoresoftware.ca/ns), which we use the prefix "tts" to represent. The attributes are as follows:

tts:role is used for symbols to denote the role that the symbol plays in the document. A symbol might represent, for example, an output value relation, an auxiliary definition or a variable.

tts:kind is used for theories to denote the kind of specification document that the theory represents (requirements, module interface, module internal design).

OMDoc supports both OpenMath [16] and Content MathML [17] for mathematical content, but since our intention is to use tabular expressions, we need to use an extensible notation, so we use only OpenMath in this version.

2 Tool Support

The set of tools that may be appropriate outcomes from this project is very large and includes powerful editors, document consistency checkers, verification systems, oracle generators, test case generators and model checkers, to name a few. Clearly to develop all of these from scratch would be a major undertaking

far beyond the resources of this project. However, we strongly believe in the value of building on the strengths of existing tools where appropriate, so we are focusing our initial efforts on ways to leverage existing systems to our advantage in this project. The OMDoc representation of a tabular specification with its embedded semantics is the common glue that allows us to easily bind together components as diverse as a Eclipse plugin GUI, the PVS theorem prover and a prototype function based specification system that also acts as a Java code generator. Once development is completed to enable these tools to extract the general table representation and semantics of [8], support will be available for all known types of tabular specifications and any future ones that can be represented within this general table model. The current state of these three components of the table tool system are outlined below.

2.1 Prototype Eclipse Plugin GUI

Eclipse (www.eclipse.org) is an open development platform that supports extension through a plugin mechanism. The platform provides an advanced integrated development environment for software development, and a wide range of available plugins to support such tasks as testing, modeling and documentation. By developing a plugin to support production of the documents described above, we hope to be able to build on the strengths of Eclipse and to help integrate the documentation into the development process, for example by supporting navigation between a specification and the code that implements the specification or by generating oracles or test cases that integrate with automated testing using JUnit (www.junit.org) and the JUnit plugin.

The initial version of this plugin, which is pictured in figure 1, provides a "multi-page editor" (which provides different views of the same source file) for ".tts" files, which are OMDoc files. One page of the editor is a structured view of the document, while another shows the raw XML representation. The support libraries in Eclipse provide techniques to ensure that the views of the document are consistent. The plugin is built using several open source libraries including the RIACA OpenMath Library[5].

This plugin is seen as a primary means for the human users to interact with specification documents. Currently it supports basic verification and validation of tabular specifications via export to the Prototype Verification System (PVS) [18] using XSLT to translate the OMDoc into PVS, as described below.

2.2 Example Verification and Validation Environment

PVS is a "proof assistant" that can automatically check for completeness (coverage) and determinism (disjointness) of several types of tables [19], i.e. PVS checks that a table defines a total function. This is typically very important in safety critical environments since the engineers want to avoid any unspecified

[5] http://www.mathdox.org/projects/openmath/lib/2.0/

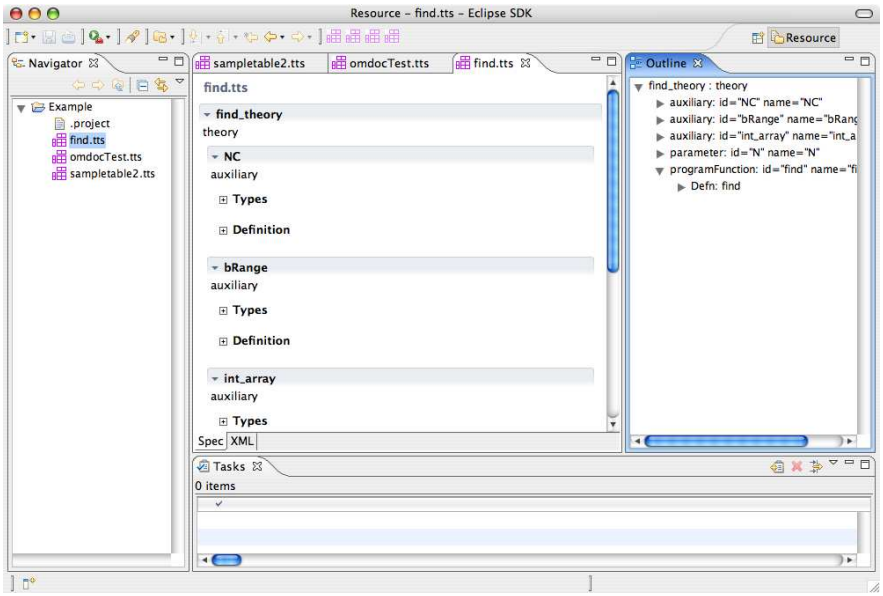


Fig. 1. Screenshot of Eclipse Plugin

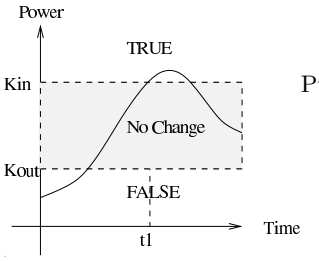
behaviour. Although PVS has a steep learning curve for users, with further development effort we can design our table tools and software process to "shield" the users from PVS. Further, new features in PVS such as the random test [20] and execution of a subset of the PVS specification language via the ground evaluator [21] can be easily translated into new table tool features.

We illustrate these capabilities with an example, a simple Reactor Shutdown System (SDS) component. An SDS is a watchdog system that monitors system parameters. It shuts down (trips) the reactor if it observes "bad" behaviour. The process control is performed by a separate Digital Control computer (DCC) since that functionality is not as critical.

We will consider a "Power Conditioning" subsystem. Often sensors have a power threshold below (or above) which readings are unreliable so it's "conditioned out" for certain power levels. A deadband is used to eliminate sensor "chatter". Since there are many different sensor types with similar power conditioning requirements, during the design phase it was decided to write one general routine and pass in sensor parameters for different sensors, thereby taking advantage of code reuse.

Consider the General Power Conditioning Function illustrated in Figure 2 When $Power$:

- drops below $Kout$, sensor is unreliable so it's "conditioned out" ($PwrCond = FALSE$).
- exceeds Kin, the sensor is "conditioned in" and is used to evaluate the system.



PwrCond(Prev:bool, Power, Kin, Kout:posreal):bool =

$Power \leq Kout$	$Kout < Power < Kin$	$Power \geq Kin$
$FALSE$	$Prev$	$TRUE$

Fig. 2. General power conditioning function with deadband from [22]

– is between $Kout$ and Kin, the value of $PwrCond$ is left unchanged by setting it to its previous value, $Prev$.

For the graph of $Power$ above, $PwrCond$ would start out FALSE, then become TRUE at time $t1$ and remain TRUE.

The PVS Specification of the General $PwrCond$ Function can be generated from the OMDoc tabular specification shown in Figure 3 by applying a modified version of the original `omdoc2pvs.xsl` by Kolhase that is available from the OMDoc subversion repository[6]

The PVS generated by applying the stylesheet is shown Figure 4. We note that white space has been manual added to the figure to improve its readability, though this does not change the semantics of the generated file. This PVS specification of the $PwrCond$ table produces the following proof obligations or "TCCs" (Type Correctness Conditions).

```
% Disjointness TCC generated (at line 14, column 55) for
  % unfinished
PwrCond_TCC1: OBLIGATION
  FORALL (Kin, Kout: posreal, Power):
    NOT (Power <= Kout AND Power > Kout & Power < Kin) AND
     NOT (Power <= Kout AND Power >= Kin) AND
      NOT ((Power > Kout & Power < Kin) AND Power >= Kin);

% Coverage TCC generated (at line 14, column 55) for
  % proved - complete
PwrCond_TCC2: OBLIGATION
  FORALL (Kin, Kout: posreal, Power):
    (Power <= Kout OR              % Column1
     (Power > Kout & Power < Kin)  % Column2
     OR Power >= Kin)              % Column3
```

When type-checking the `PwrCond` table the coverage TCC is automatically proved by PVS. Thus we conclude that at least one column is always satisfied for every input. But PVS attempt to prove the Disjointness TCC fails,

[6] Available at https://svn.omdoc.org/repos/omdoc/branches/omdoc-1.2.

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE omdoc PUBLIC "-//OMDoc//DTD OMDoc Spec 1.2//EN"
  "http://www.omdoc.org/omdoc/dtd/omdoc-spec.dtd" [] >
<omdoc modules="@spec" version="1.2" xml:id="sampletable2.omdoc"
    xmlns="http://www.omdoc.org/omdoc" xmlns:cc="http://creativecommons.org/ns"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:tts="http://www.fillmoresoftware.ca/ns">
  <theory xml:id="sampletable2_theory">
    <symbol name="sampletable2" scope="global" tts:role="auxiliary"
        xml:id="sampletable2">
      <type system="pvs">
        <OMOBJ xmlns="http://www.openmath.org/OpenMath">
          <OMA>
            <OMS cd="pvs" name="funtype" /> <OMS cd="booleans" name="bool" />
            <OMS cd="reals" name="real" /> <OMS cd="reals" name="real" />
            <OMS cd="reals" name="real" /> <OMS cd="booleans" name="bool" />
          </OMA>
        </OMOBJ>
      </type>
    </symbol>
    <definition for="#sampletable2" type="simple" xml:id="sampletable2-def">
      <OMOBJ xmlns="http://www.openmath.org/OpenMath">
        <OMBIND> <OMS cd="pvs" name="lambda" />
          <OMBVAR>
            <OMATTR>
              <OMATP> <OMS cd="pvs" name="type" /> <OMS cd="booleans" name="bool" />
              </OMATP>
              <OMV name="Prev" /> </OMATTR>
            <OMATTR>
              <OMATP> <OMS cd="pvs" name="type" /> <OMS cd="reals" name="real" />
              </OMATP>
              <OMV name="Power" /> </OMATTR>
            <OMATTR>
              <OMATP> <OMS cd="pvs" name="type" /> <OMS cd="reals" name="real" />
              </OMATP>
              <OMV name="Kin" /> </OMATTR>
            <OMATTR>
              <OMATP> <OMS cd="pvs" name="type" /> <OMS cd="reals" name="real" />
              </OMATP>
              <OMV name="Kout" /> </OMATTR>
          </OMBVAR>
          <OMA> <OMS cd="table" cdbase="http://www.fillmoresoftware.ca/cd"
                  name="table" />
            <!-- Evaluation term: normal(0) : normal table, value grid = 0. -->
            <OMA> <OMS cd="table" cdbase="http://www.fillmoresoftware.ca/cd"
                    name="normal" /> <OMI> 0 </OMI> </OMA>
            <!-- Static restriction: rectStructure(1, <3>) -->
            <OMA> <OMS cd="table" cdbase="http://www.fillmoresoftware.ca/cd"
                    name="rectStructure" />
              <OMI> 1 </OMI>
              <OMA> <OMS cd="linalg2" cdbase="http://www.openmath.org/cd"
                      name="vector" /> <OMI> 3 </OMI> </OMA>
            </OMA>
            <!-- dynamic restriction: proper(1) -->
            <OMA> <OMS cd="table" cdbase="http://www.fillmoresoftware.ca/cd"
                    name="proper" /> <OMI> 1 </OMI> </OMA>
            <OMA>
              <!-- List of grids -->
              <OMS cd="list1" cdbase="http://www.openmath.org/cd" name="list" />
                <!-- Grid 0: false | Prev | true -->
              <OMA> <OMS cd="table" cdbase="http://www.fillmoresoftware.ca/cd"
                      name="grid" />
                <OMA> <OMS cd="products" cdbase="http://www.openmath.org/cd"
                        name="pair" />
                  <OMA> <OMS cd="linalg2" name="vector" /> <OMI> 0 </OMI> </OMA>
                  <OMS cd="logic1" cdbase="http://www.openmath.org/cd"
                      name="false" />
                </OMA>
                ...
              </OMA>
              <OMA>
                <!-- Grid 1: Power <= Kout | Power > Kout & Power < Kin
                                | Power >= Kin -->
                <OMS cd="table" cdbase="http://www.fillmoresoftware.ca/cd"
                    name="grid" />
                <OMA cdbase="http://www.openmath.org/cd">
                  <OMS cd="products" name="pair" />
                  <OMA>
                    <OMS cd="linalg2" name="vector" /> <OMI> 0 </OMI> </OMA>
                  <OMA> <OMS cd="relation1" name="leq" />
                    <OMV name="Power" /> <OMV name="Kout" /> </OMA>
                </OMA>
                ...
              </OMA> </OMA> </OMA> </OMBIND> </OMOBJ> </definition> </theory>
</omdoc>
```

Fig. 3. Partial OMDoc representation of General Power Conditioning

```
PwrCond(Prev:bool, Power, Kin, Kout:posreal):bool = TABLE
  %-------------------------------------------------%
  |[Power<=Kout | Power>Kout & Power<Kin | Power>=Kin]|
  %-------------------------------------------------%
  |   FALSE      |        Prev            |   TRUE   ||
  %-------------------------------------------------%
ENDTABLE
```

Fig. 4. PVS Specification generated modified `omdoc2pvs.xsl` stylesheet

indicating that the columns might overlap. The resulting unprovable sequent for the disjointness TCC is given below along with the results of running the `(random-test)` prover command to attempt to generate a counter example for the sequent:

```
PwrCond_TCC1 :
[-1]    Kin!1 > 0
[-2]    Kout!1 > 0
[-3]    Power!1 > 0
[-4]    Power!1 <= Kout!1
[-5]    (Kin!1 <= Power!1)
  |-------
[1]     FALSE
Rule? (random-test)
The formula is falsified with the substitutions:
 Power ==> 67 / 80
 Kin ==> 31 / 85
 Kout ==> 42 / 25
```

This command generates and evaluates a "theorem" on random inputs to look for counter examples, printing the first counter example (if any) found [20]. To confirm the counter example and locate problem we can use the PVSio evaluator [21] to check the headers of all columns at once on the above counterexample values.

```
<PVSio> let (Prev,Power,Kin,Kout) = (FALSE, 67/80, 31/85, 42/25)
  in (Power<=Kout, Power>Kout & Power<Kin, Power>=Kin);
==>
(TRUE, FALSE, TRUE)
```

Thus we conclude that columns 1 and 3 overlap.

While the above steps in PVS were done manually, there is no reason why these steps could not be automated via the Eclipse plugin using PVS's batch processing mode, thus "shielding" the user from the theorem prover under the hood of the table tool system. For example, the plugin could simply provide the counter example and highlight the overlapping columns in a visual display

rendered as display MathML and xhtml by modifying existing XSLT stylesheets contained in the OMDoc distribution.

2.3 Functional Specification and Code Generation

Applied mathematics in science and engineering are traditionally formulated in first-order predicate logic. With the advent of theoretical computer science and computer assisted formalization, alternative logical foundations have emerged. Many automated theorem provers such as PVS or Isabelle/HOL [23] are based on higher-order logic. Via the Curry-Howard isomorphism, higher-order logic is closely related to the typed lambda-calculus [24], the foundation of functional programming. Writing formal specifications in a style based on functions and higher-order logic has several benefits:

- *Type systems for lambda-calculus are precise, powerful and well understood.* Checking and inference algorithms are well documented. Specifications can be typechecked for consistency, catching many simple errors and ambiguities.
- *Type systems for lambda-calculus are largely self-contained.* Algebraic datatypes such as integers, tuples, enumeration and record types and the associated operations can be defined within the formalism, instead of being given by axioms or external reference. Paremetrization is for free in the lambda-calculus. Hence complex specifications can refer to a common library of basic definitions, rather than requiring special support in every processing tool.
- *Function-based specification is computationally constructive.* Standard interpretation and compilation techniques for functional programming languages apply, yielding direct and universal evaluation algorithms and code generators for agile specification tool support, simulation and oracle generation.

We have constructed the prototype of a tool that provides basic support for function-based specification. It has a frontend syntax similar to a functional programming or theorem prover language, and a semantic intermediate representation based on OpenMath objects for individual types and definitions, and OMDoc for theory-level structure. A typechecker supports the Calculus of Constructions [25] (CC). This is a subset of the Extended Calculus of Construction [26] (ECC), the proposed higher-order type system for OpenMath [27]. Executable code can be generated from the typechecked intermediate representation. The tool is implemented in Java, and currently only Java code generation is supported. Specification modules processed by the tool fulfill several roles:

Generic Library Some datatypes and operations common to all tabular expressions. The grids of a table are organized as hierarchical arrays, lists or associative lists of cell expressions. Evaluation and restriction terms are conveniently defined in terms of well-known higher-order operations such as `map`, `filter` and `fold`, extending the work of [28].
The logic of tables in [8] is total. For the transparent embedding of partial functions into cell expressions, a monadic error-handling framework [29] is provided.

Specific Tables Individual tables can be extracted from tts files and translated to function-based style. An automatic translation procedure is currently being implemented. It assumes that the expressions in a table do not involve infinite quantification, which has no direct effective translation to lambda-calculus. Table cell expressions are represented as functions of all free variables, so that each cell is a closed expression and can be checked and compiled individually. For example, the table (2) would be rendered as:

$$\mathrm{f}(x,y) \stackrel{\mathrm{df}}{=} \tag{3}$$

	$\lambda x, y.\, x > 1$	$\lambda x, y.\, x \leq 1$
$\lambda x, y.\, y < 0$	$\lambda x, y.\, x + y$	$\lambda x, y.\, x - y$
$\lambda x, y.\, y = 0$	$\lambda x, y.\, x$	$\lambda x, y.\, x * y$
$\lambda x, y.\, y > 0$	$\lambda x, y.\, y$	$\lambda x, y.\, x/y$

In this form, the assignment of values to the variables requires no reinterpretation or substitution of cell expressions, because it can be expressed by simply applying each cell function to the tuple of values. For example, the assignment $\{x := 2, y := 4\}$ is given by the value tuple $(2, 4)$.

Specifications represented in function-based style and processed using this tool have two important properties. Firstly, they are defined in a self-contained and unambiguous way in pure typed lambda-calculus. Together with the OMDoc-based format, this makes a good starting point for interaction with various theorem proving tools. Secondly, all properties that do not involve infinite quantification are directly computable. Hence the static restriction check for a table and the evaluation and dynamic restriction check for a table and a given variable assignment can be interpreted or compiled to executable code, whereas the dynamic restriction check for all possible values still requires the use of a theorem prover.

By combining both properties, substantial support for constructing new specification tools can be given. We have defined table types from [8] not yet supported by either PVS or any other available tool as part of the domain-independent library. Such a table type definition can be written by a skilled functional programmer in one day. By using our Java code generator, the core of an oracle generator is obtained immediately.

3 Related Work

A very good discussion of the need for mathematical content markup such as OMDoc is given in Part I of [1], so it will not be repeated here. In summary, general purpose documentation tools and presentation markup languages (e.g., LaTeX, HTML, Presentation MathML) are insufficient for our purposes since they encode the appearance of the mathematics, not its intended meaning. For example, given the expression "$x + 2$", we are concerned primarily with the fact that this represents the sum of a variable x and the constant 2. The choice of presentation of this expression in this or another form (e.g., "$x\,2+$" or "$\mathrm{sum}(x, 2)$")

is a matter of style that will be determined by the conventions adopted by the author and the intended readers.

The use of XSLT to translate the OMDoc into PVS input represents a lightweight approach to providing support for OMDoc specification in existing tools. A more rigorous approach to preservation of table semantics in multiple verification environments such as PVS and the prototype functional specification/code generation environment might be based upon the Heterogeneous Tool Set (Hets) [30], a parsing, static analysis and proof management tool combining various tools for different specification languages. Currently the Hets system does not support PVS or Java code generation, though it does support the Isabelle interactive higher-order theorem prover and SPASS automatic first-order theorem prover and can generate Haskell code. A list of other projects using OMDoc is found in Ch. 26 of [1].

Several projects have addressed the problem of developing tools for use of tabular expressions in documents [22, 31–35]. All of these efforts, with the exception of [33], are limited in the set of tabular expressions or document types that they targeted, and none used a standard interchange format such as OMDoc to take advantage of other tools. The table tools developed by the CANDU owners group and used at Ontario Power Generation (OPG) on the Darlington Nuclear Generating Station Shutdown Systems, used standard wordprocessors to create documents containing tabular specification. These documents were saved in RTF format and then custom tools extracted the tables and exported them to PVS [22, 35]. One of the difficulties faced in developing the tools is that the table semantics had to be inferred from the table layout in RTF. This limited the tools' capabilities to support new table formats.

4 Conclusions and Future Work

This is a development project in its early stages, and we expect that it will evolve as it progresses by the enhancement of the existing tools and the addition of new tools. Our early results show that there is promise in the chosen techniques – the model supports the needs of our documentation and the ability to interact with other tools such as PVS shows the potential to achieve significant leverage from these tools.

Near term targets for the tools are to enhance the plugin such that it is a reasonably complete and user-friendly editor, to continue to work with translation to PVS so that we can effectively check properties of our documents, and to add oracle generation similar to [36] and [13].

Acknowledgements This work draws on the contributions of many people who have worked on tabular methods in the past primarily at McMaster University and the University of Limerick. In particular we are grateful to Dr. David L. Parnas for his inspiration, support, and helpful comments and to Jin (Kim) Ying and Adam Balaban for their work on the formalization of the semantics of tabular expressions.

The authors gratefully acknowledge support received for this work from the Science Foundation Ireland (SFI) through the Software Quality Research Laboratory (SQRL) at the University of Limerick and from the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Kohlhase, M.: OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2). Number 4180 in LNAI. Springer Verlag (2006)
2. Heninger, K.L., Parnas, D.L., Shore, J.E., Kallander, J.: Software requirements for the A-7E aircraft. Technical Report MR 3876, Naval Research Laboratory (1978)
3. Parnas, D.L.: Inspection of safety critical software using function tables. In: Proc. IFIP Congress. Volume I., North Holland (1994) 270–277
4. Weiss, G.H., Hohendorf, R., Wassyng, A., B.Quigley, Borsch, M.R.: Verification of the shutdown system software at the darlington nuclear generating station. In: Int'l Conf. Control & Instrumentation in Nuclear Installations. Number 4.3, Glasgow, United Kingdom, Institution of Nuclear Engineers (1990)
5. Abraham, R.F.: Evaluating generalized tabular expressions in software documentation. M. Eng. thesis, McMaster University, Dept. of Electrical and Computer Engineering, Hamilton, ON (1997)
6. Parnas, D.L.: Tabular representation of relations. CRL Report 260, Communications Research Laboratory, Hamilton, Ontario, Canada (1992)
7. Janicki, R., Khedri, R.: On a formal semantics of tabular expressions. Science of Computer Programming **39**(2–3) (2001) 189–213
8. Balaban, A., Bane, D., Jin, Y., Parnas, D.: Mathematical model of tabular expressions. SQRL Document (2006)
9. Parnas, D.L., Madey, J.: Functional documentation for computer systems. Science of Computer Programming **25**(1) (1995) 41–61
10. Parnas, D.L.: A family of mathematical methods for professional software documentation. In Romijn, J.M.T., Smith, G.P., van de Pol, J.C., eds.: Proc. Int'l Conf. on Integrated Formal Methods. Number 3771 in LNCS, Springer-Verlag (2005) 1–4
11. van Schouwen, A.J., Parnas, D.L., Madey, J.: Documentation of requirements for computer systems. In: Proc. Int'l Symp. Requirements Eng. (RE '93), IEEE (1993) 198–207
12. Peters, D.K.: Deriving Real-Time Monitors from System Requirements Documentation. PhD thesis, McMaster University, Hamilton ON (2000)
13. Peters, D.K., Parnas, D.L.: Requirements-based monitors for real-time systems. IEEE Trans. Software Engineering **28**(2) (2002) 146–158
14. Quinn, C., Vilkomir, S., Parnas, D., Kostic, S.: Specification of software component requirements using the trace function method. In: Int'l Conf. on Software Engineering Advances, Los Alamitos, CA, IEEE Computer Society (2006) 50
15. Parnas, D.L., Madey, J., Iglewski, M.: Precise documentation of well-structured programs. IEEE Trans. Software Engineering **20**(12) (1994) 948–976
16. The OpenMath Society: The OpenMath Standard. 2.0 edn. (2004) http://www.openmath.org/standard/om20-2004-06-30/.
17. W3C: Mathematical Markup Language (MathML) Version 2.0. Second edn. (2003) http://www.w3.org/TR/MathML2/.
18. Owre, S., Rushby, J., Shankar, N., von Henke, F.: Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. IEEE Transactions on Software Engineering **21**(2) (1995) 107–125

19. Owre, S., Rushby, J., Shankar, N.: Integration in PVS: Tables, types, and model checking. In Brinksma, E., ed.: Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97). Volume 1217 of LNCS, Enschede, The Netherlands, Springer-Verlag (1997) 366–383
20. Owre, S.: Random testing in pvs. In: Proceedings of Automated Formal Methods (AFM06), SRI International (2006) Available online: http://fm.csl.sri.com/AFM06/.
21. Muñoz, C.A.: PVSio Reference Manual. National Institute of Aerospace (NIA), Formal Methods Group, 100 Exploration Way, Hampton VA 23666. Version 2.b (draft) edn. (2005) Available at http://research.nianet.org/ munoz/PVSio/.
22. Lawford, M., Froebel, P., Moum, G.: Application of tabular methods to the specification and verification of a nuclear reactor shutdown system. Accepted for publication in Oct 2004. http://www.cas.mcmaster.ca/ lawford/papers/ (To appear in FMSD)
23. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
24. Barendregt, H.: Lambda calculi with types. In Abramsky, Gabbay, Maibaum, eds.: Handbook of Logic in Computer Science. Volume 2. Clarendon (1992)
25. Coquand, T., Huet, G.: The calculus of constructions. Inf. Comput. **76**(2-3) (1988) 95–120
26. Luo, Z.: An Extended Calculus of Constructions. PhD thesis, University of Edinburgh (1990)
27. The OpenMath Society: A Type System for OpenMath. 1.0 edn. (1999) http://www.openmath.org/standard/ecc.pdf.
28. Kahl, W.: Compositional syntax and semantics of tables. SQRL Report 15, Software Quality Research Laboratory, Department of Computing and Software, McMaster University (2003)
29. Spivey, M.: A functional theory of exceptions. Sci. Comput. Program. **14**(1) (1990) 25–42
30. Mossakowski, T.: HETS User Guide. Department of Computer Science and Bremen Institute for Safe Systems, University of Bremen, Germany. (2006) Online: http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/UserGuide.pdf.
31. Heitmeyer, C.L., Bull, A., Gasarch, C., Labaw, B.G.: SCR*: A toolset for specifying and analyzing requirements. In: Proc. Conf. Computer Assurance (COMPASS), Gaithersburg, MD, National Institute of Standards and Technology (1995) 109–122
32. Hoover, D.N., Chen, Z.: Tablewise, a decision table tool. In: Proc. Conf. Computer Assurance (COMPASS), Gaithersburg, MD, National Institute of Standards and Technology (1995) 97–108
33. Parnas, D.L., Peters, D.K.: An easily extensible toolset for tabular mathematical expressions. In: Proc. Fifth Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems. Number 1579 in LNCS, Springer-Verlag (1999) 345–359
34. Lawford, M., McDougall, J., Froebel, P., Moum, G.: Practical application of functional and relational methods for the specification and verification of safety critical software. In Rus, T., ed.: Proc. of AMAST 2000. Volume 1816 of LNCS., Iowa City, Iowa, USA, Springer (2000) 73–88
35. Wassyng, A., Lawford, M.: Software tools for safety-critical software development. International Journal on Software Tools for Technology Transfer (STTT) **8**(4–5) (2006) 337–354
36. Peters, D.K., Parnas, D.L.: Using test oracles generated from program documentation. IEEE Trans. Software Engineering **24**(3) (1998) 161–173

Reasoning inside a formula and ontological correctness of a formal mathematical text

Andrei Paskevich[1], Konstantin Verchinine[1],
Alexander Lyaletski[2], and Anatoly Anisimov[2]

[1] Université Paris 12, IUT Sénart/Fontainebleau, 77300 Fontainebleau, France,
andrei@capet.iut-fbleau.fr verko@capet.iut-fbleau.fr
[2] Kyiv National Taras Shevchenko University, Faculty of Cybernetics,
03680 Kyiv, Ukraine,
lav@unicyb.kiev.ua ava@unicyb.kiev.ua

Abstract. Dealing with a formal mathematical text (which we regard as a structured collection of hypotheses and conclusions), we often want to perform various analysis and transformation tasks on the initial formulas, without preliminary normalization. One particular example is checking for "ontological correctness", namely, that every occurrence of a non-logical symbol stems from some definition of that symbol in the foregoing text. Generally, we wish to test whether some known fact (axiom, definition, lemma) is "applicable" at a given position inside a statement, and to actually apply it (when needed) in a logically sound way.
In this paper, we introduce the notion of a locally valid statement, a statement that can be considered true at a given position inside a first-order formula. We justify the reasoning about "innards" of a formula; in particular, we show that a locally valid equivalence is a sufficient condition for an equivalent transformation of a subformula. Using the notion of local validity, we give a formal definition of ontological correctness for a text written in a special formal language called ForTheL.

1 Introduction

In a mathematical text, be it intended for a human reader or formalized for automated processing (Mizar Mathematical Library [1] is a classical example, see also [2]), we rarely meet "absolute", unconstrained rules, definitions, statements. Usually, everything we use is supplied with certain conditions so that we have to take them into consideration. For example, we can not reduce the fraction $\frac{xy}{x}$ until we prove that x is a nonzero number.

Let us consider a formula of the form $(\cdots \forall x\,(x \in \mathbb{R}^+ \supset (\cdots \frac{xy}{x} \cdots)) \cdots)$. It seems to be evident that we can replace $\frac{xy}{x}$ with y, but could we justify that? The task itself seems to be absurd: as soon as a term depends on bound variables, we can not reason about it. In the traditional fashion, we should first split our big formula up to the quantifier that binds x, make a substitution (or skolemization), separate $x \in \mathbb{R}^+$, and only then make the simplification.

However, while the statement "x is non-zero" is surely meaningless, we can say that "x is non-zero in this occurrence of $\frac{xy}{x}$". Our intuition suggests that

along with the usual notion of validity, a certain *local validity* of a proposition can be defined with respect to some position in a formula. A statement that is generally false or just meaningless can become locally valid being considered in the corresponding context. In what follows, we call such a proposition a *local property* of the position in question.

It can be argued that there is no gain in any simplifications when a formula to be simplified lies deep inside. We would split our big formula anyway to use the properties of that fraction in a proof. However, we believe that it is useful and instructive to simplify a problem in its initial form as far as possible in order to select the most perspective direction of the proof search.

Local properties are also necessary to verify (and even to define!) what we call *ontological correctness* of a mathematical text. Informally, we consider a text ontologically correct whenever it contains no occurrence of a non-logical symbol that comes from nowhere: for every such occurrence there must be an applicable definition or some other "introductory" premise. The purpose of ontological correctness may be not immediately obvious: for example, the famous disjunction "*to be or not to be*" is perfectly valid (at least, in classical logic) even if the sense of being has never been defined. Nevertheless, ontologically correct texts are preferable in many aspects.

Ontological correctness is closely related to type correctness in typed languages (especially, in weakly typed systems such as WTT [3]). It allows to spot formalization errors which otherwise could hardly be detected. Indeed, an accidental ontological incorrectness most often implies logical incorrectness (i.e. presence of false or unprovable claims). And it is much harder to trace a failure log of a prover back to an invalid occurrence than to discover it in the first place.

Moreover, during ontological verification, we obtain information about applicability of definitions and other preliminary facts at individual positions in the text in question. As long as subsequent transformations (e.g. during the logical verification phase) preserve ontological correctness and other local properties (and that should always be the case) we can unfold definitions and apply lemmas without further checking.

This paper is devoted to formalization of ontological correctness for a particular language of formal mathematical texts, called ForTheL [4]. To this purpose, we develop a theoretical background for reasoning about local properties based on the notion of *local image*. The rest of the paper is organized as follows. In Section 2, we briefly describe the ForTheL language and provide an informal (at the moment) definition of ontological correctness of a ForTheL text. Section 3 introduces preliminary notions and notation which we use to define and investigate the notion of local image in Section 4. With the help of local images, we give a precise definition of ontological correctness in Section 5.

2 ForTheL language

Like any usual mathematical text, a ForTheL text consists of definitions, assumptions, affirmations, theorems, proofs, etc. The syntax of a ForTheL sentence

follows the rules of English grammar. Sentences are built of units: statements, predicates, notions (that denote classes of objects) and terms (that denote individual entities). Units are composed of syntactical primitives: nouns which form notions (e.g. "`subset of`") or terms ("`closure of`"), verbs and adjectives which form predicates ("`belongs to`", "`compact`"), symbolic primitives that use a concise symbolic notation for predicates and functions and allow to consider usual quantifier-free first-order formulas as ForTheL statements. Of course, just a little fragment of English is formalized in the syntax of ForTheL.

There are three kinds of sentences in the ForTheL language: assumptions, selections, and affirmations. Assumptions serve to declare variables or to provide some hypotheses for the following text. For example, the following sentences are typical assumptions: "`Let S be a finite set.`", "`Assume that m is greater than n.`". Selections state the existence of representatives of notions and can be used to declare variables, too. Here follows an example of a selection: "`Take an even prime number X.`". Finally, affirmations are simply statements: "`If p divides n - p then p divides n.`". The semantics of a sentence is determined by a series of transformations that convert a ForTheL statement to a first-order formula, so called *formula image*.

Example 1. The formula image of the statement "`all closed subsets of any compact set are compact`" is:

$$\forall\, \texttt{A}\,((\texttt{A is a set} \wedge \texttt{A is compact}) \supset \forall\, \texttt{B}\,((\texttt{B is a subset of A} \wedge \texttt{B is closed}) \supset \texttt{B is compact}))$$

ForTheL sections are: sentences, sentences with proofs, cases, and top-level sections: axioms, definitions, signature extensions, lemmas, and theorems. A top-level section is a sequence of assumptions concluded by an affirmation. Proofs attached to affirmations and selections are simply sequences of low-level sections. A case section consists of an assumption called *case hypothesis* followed by a sequence of low-level sections (the proof of a case).

Any section $\mathbb{A}$ or sequence of sections Δ has a formula image, denoted $|\mathbb{A}|$ or, respectively, $|\Delta|$. The image of a sentence with proof is the same as the image of that sentence taken without proof. The image of a case section is the implication ($H \supset$ `thesis`), where H is the formula image of the case hypothesis and `thesis` is a placeholder that will be replaced by the statement being proved during verification. The formula image of a top-level section is simply the image of the corresponding sequence of sentences.

The formula image of a sequence of sections $\mathbb{A}, \Delta$ is a conjunction $|\mathbb{A}| \wedge |\Delta|$, whenever $\mathbb{A}$ is a conclusion (affirmation, case section, lemma, theorem); or a universally quantified implication $\forall \boldsymbol{x}_{\mathbb{A}}(|\mathbb{A}| \supset |\Delta|)$, whenever $\mathbb{A}$ is a hypothesis (assumption, selection, case hypothesis, axiom, definition, signature extension). Here, $\boldsymbol{x}_{\mathbb{A}}$ denotes the set, possibly empty, of variables declared in $\mathbb{A}$ (this set also depends on the logical context of $\mathbb{A}$, since any variable which is declared above $\mathbb{A}$ in the text must not be bound in $|\mathbb{A}|$). The formula image of the empty sequence is $\top$, the truth.

Thus, a ForTheL text as a whole, being a sequence of section, can be expressed as a single logical formula. In what follows, we often use formulas, sections and sequence of sections interchangeably: whenever a section or a sequence of sections is mentioned where a formula is expected, the corresponding formula image should be considered.

In this syntax, we can express various proof schemes like proof by contradiction, by case analysis, and by general induction. The last scheme merits special consideration. Whenever an affirmation is marked to be proved by induction, the system constructs an appropriate induction hypothesis and inserts it into the statement to be verified. The induction hypothesis mentions a binary relation which is declared to be a well-founded ordering, hence, suitable for induction proofs. Note that we cannot express the very property of well-foundness in ForTheL (since it is essentially a first-order language), so that the correctness of this declaration is unverifiable and we take it for granted. After that transformation, the proof and the transformed statement can be verified in a first-order setting, without any specific means to build induction proofs.

Example 2. Consider the following ForTheL formalization of Newman's lemma about term rewriting systems. We give it in an abridged form, with some preliminary definitions and axioms omitted. The expression "`x -R> y`" means that `y` is a reduct of `x` in the rewriting system `R`; `R+` and `R*` denote, respectively, the transitive and the reflexive transitive closure of `R`.

```
Let a,b,c,d,u,v,w,x,y,z denote elements.
Let R,S,T denote rewriting systems.

Definition CRDef.   R is confluent iff
  for all a,b,c  such that a -R*> b and a -R*> c
  there exists d such that b -R*> d and c -R*> d.
Definition WCRDef.  R is locally confluent iff
  for all a,b,c  such that a -R> b  and a -R> c
  there exists d such that b -R*> d and c -R*> d.
Definition TrmDef.  R is terminating iff
  for all a,b  a -R+> b => b -<- a.
Definition NFRDef.  A normal form of x in R is
  an element y such that x -R*> y and y has no reducts in R.

Lemma TermNF.   Let R be a terminating rewriting system.
                Every element x has a normal form in R.
Proof by induction. Obvious.

Lemma Newman.
  Any locally confluent terminating rewriting system is confluent.
Proof.
  Let R be locally confluent and terminating.
  Let us demonstrate by induction that
        for all a,b,c  such that a -R*> b and a -R*> c
        there exists d such that b -R*> d and c -R*> d.
```

```
    Assume that a -R+> b and a -R+> c.
    Take u such that a -R> u and u -R*> b.
    Take v such that a -R> v and v -R*> c.
    Take w such that u -R*> w and v -R*> w.
    Take a normal form d of w in R.

    Let us show that b -R*> d.
      Take x such that b -R*> x and d -R*> x.
    end.
    Let us show that c -R*> d.
      Take y such that c -R*> y and d -R*> y.
    end.
  end.
qed.
```

Our formalization is simplified in that the notion "`element`" takes no arguments, i.e. we consider rewriting systems to be defined on a common (implicit) universum. Also, in our current implementation of ForTheL, one can not declare a given binary relation to be well-founded, and therefore a rewriting system is defined to be terminating iff its inverted transitive closure is a subset of *the* well-founded relation "`-<-`" (Definition `TrmDef`). The induction hypothesis (namely, that any reduct of `a` is confluent) is used to verify the selections "`Take x`..." and "`Take y`..." at the end of the proof. Note that we do not consider cases where `b` or `c`, or both are equal to `a` — these cases are trivial enough so that the system can handle them without our assistance.

The ForTheL proof of Newman's lemma, while being quite terse and close to a hand-written argument, is formal and has been automatically verified by the SAD proof assistant, using SPASS 2.2, E 0.99, and Vampire 7.45 as background provers. We refer the reader to the papers [4, 5] and to the website `http://ea.unicyb.kiev.ua` for a description of SAD and further examples.

We call a ForTheL text *ontologically correct* whenever: (a) every non-logical symbol (constant, function, notion or relation) in the text is either a signature symbol or is introduced by a definition; and (b) in every occurrence of a non-logical symbol, the arguments, if any, satisfy the guards of the corresponding definition or signature extension. Since ForTheL is a one-sorted and untyped language, these guards can be arbitrary logical formulas. Therefore, the latter condition cannot be checked by purely syntactical means nor by type inference of any kind. Instead, it requires proving statements about terms inside complex formulas, possibly, under quantifiers. The following sections provide a theoretical basis for such reasoning.

3 Preliminary notions

We consider a one-sorted first-order language with equality ($\approx$), the standard propositional connectives $\neg$, $\wedge$, $\vee$, $\supset$, $\equiv$, the quantifier symbols $\forall$ and $\exists$, and the boolean constant $\top$, denoting truth. The respective order of subformulas is

significant for some of our definitions, therefore we consider $F \wedge G$ and $G \wedge F$ as different formulas (the same is true for $\vee$, $\equiv$, and $\approx$). We write the negated equality $\neg(s_1 \approx s_2)$ as $s_1 \not\approx s_2$ and the negated truth $\neg\top$ as $\bot$.

We suppose that the sets of free and bound variables in any term or formula are always disjoint. Also, a quantifier on a variable may never appear in the scope of another quantifier on the same variable.

We consider substitutions as functions which map variables to terms. For any substitution ϕ, if $x\phi$ is different from x, we call the term $x\phi$ a *substitute* of x in ϕ. A substitution is *finite* whenever the set of its substitutes is finite. We write finite substitutions as sequences of the form $[t_1/x_1, \ldots, t_n/x_n]$.

Position is a word in the alphabet $\{0, 1, \ldots\}$. In what follows, positions are denoted by Greek letters τ, μ and ν; the letter ϵ denotes the null position (the empty word). Positions point to particular subformulas and subterms in a formula or term.

The *set of positions* in an atomic formula or a term E, denoted $\Pi(E)$, is defined as follows (below $i.\Pi$ stands for $\{\, i.\tau \mid \tau \in \Pi \,\}$):

$$\Pi(P(s_0, \ldots, s_n)) = \{\epsilon\} \cup \bigcup i.\Pi(s_i) \qquad \Pi(s \approx t) = \{\epsilon\} \cup 0.\Pi(s) \cup 1.\Pi(t)$$
$$\Pi(f(s_0, \ldots, s_n)) = \{\epsilon\} \cup \bigcup i.\Pi(s_i) \qquad \Pi(\top) = \{\epsilon\}$$

The *set of positions* in a formula H, denoted $\Pi(H)$, is the disjoint union

$$\Pi(F) = \Pi^+(F) \cup \Pi^-(F) \cup \Pi^\circ(F)$$

of the *set of positive positions* $\Pi^+(H)$, the *set of negative positions* $\Pi^-(H)$, and the *set of neutral positions* $\Pi^\circ(H)$ (in what follows, A stands for an atomic formula):

$$\Pi^+(F \equiv G) = \{\epsilon\} \qquad \Pi^+(\forall x\, F) = \{\epsilon\} \cup 0.\Pi^+(F)$$
$$\Pi^+(F \supset G) = \{\epsilon\} \cup 0.\Pi^-(F) \cup 1.\Pi^+(G) \qquad \Pi^+(\exists x\, F) = \{\epsilon\} \cup 0.\Pi^+(F)$$
$$\Pi^+(F \vee G) = \{\epsilon\} \cup 0.\Pi^+(F) \cup 1.\Pi^+(G) \qquad \Pi^+(\neg F) = \{\epsilon\} \cup 0.\Pi^-(F)$$
$$\Pi^+(F \wedge G) = \{\epsilon\} \cup 0.\Pi^+(F) \cup 1.\Pi^+(G) \qquad \Pi^+(A) = \Pi(A)$$

$$\Pi^-(F \equiv G) = \varnothing \qquad \Pi^-(\forall x\, F) = 0.\Pi^-(F)$$
$$\Pi^-(F \supset G) = 0.\Pi^+(F) \cup 1.\Pi^-(G) \qquad \Pi^-(\exists x\, F) = 0.\Pi^-(F)$$
$$\Pi^-(F \vee G) = 0.\Pi^-(F) \cup 1.\Pi^-(G) \qquad \Pi^-(\neg F) = 0.\Pi^+(F)$$
$$\Pi^-(F \wedge G) = 0.\Pi^-(F) \cup 1.\Pi^-(G) \qquad \Pi^-(A) = \varnothing$$

$$\Pi^\circ(F \equiv G) = 0.\Pi(F) \cup 1.\Pi(G) \qquad \Pi^\circ(\forall x\, F) = 0.\Pi^\circ(F)$$
$$\Pi^\circ(F \supset G) = 0.\Pi^\circ(F) \cup 1.\Pi^\circ(G) \qquad \Pi^\circ(\exists x\, F) = 0.\Pi^\circ(F)$$
$$\Pi^\circ(F \wedge G) = 0.\Pi^\circ(F) \cup 1.\Pi^\circ(G) \qquad \Pi^\circ(\neg F) = 0.\Pi^\circ(F)$$
$$\Pi^\circ(F \vee G) = 0.\Pi^\circ(F) \cup 1.\Pi^\circ(G) \qquad \Pi^\circ(A) = \varnothing$$

For the sake of consistency, we set $\Pi^+(t) = \Pi(t)$ and $\Pi^-(t) = \Pi^\circ(t) = \varnothing$ for any term t.

Among positions, we distinguish those of formulas ($\Pi_{\mathbf{F}}$), those of atomic formulas ($\Pi_{\mathbf{A}}$), and those of terms ($\Pi_{\mathbf{t}}$). Obviously, $\Pi(F) = \Pi_{\mathbf{t}}(F) \cup \Pi_{\mathbf{F}}(F)$, $\Pi_{\mathbf{A}}(t) = \Pi_{\mathbf{F}}(t) = \varnothing$, $\Pi_{\mathbf{A}}(F) \subseteq \Pi_{\mathbf{F}}(F)$, $\Pi(t) = \Pi_{\mathbf{t}}(t)$. We split the sets $\Pi_{\mathbf{t}}$, $\Pi_{\mathbf{A}}$, and $\Pi_{\mathbf{F}}$ into positive, negative, and neutral parts, too.

Given a formula H and a position $\pi \in \Pi(H)$, the position $\widehat{\pi}$ is the maximal prefix of π in $\Pi_{\mathbf{F}}(H)$. In what follows, we will often use this conversion to extend notions and operations defined on positions from $\Pi_{\mathbf{F}}$ to the whole Π.

A formula or a term E along with a position $\tau \in \Pi(E)$ defines an *occurrence*.

Let us say that π is a *textual predecessor* of τ whenever $\pi = \omega.i.\mu$ and $\tau = \omega.j.\eta$ and $i < j$. Such positions will be called *adjacent*. If $\mu = \epsilon$, we will say that π is a *logical predecessor* of τ. By default, "predecessor" means "logical predecessor".

Given a formula or a term E and a position τ in $\Pi(E)$, we will denote by $E|_\tau$ the subformula or subterm occurring in that position. In what follows, $(*\,F)$ stands for $(\neg F)$, $(\forall x\, F)$, or $(\exists x\, F)$; and $(F\ *\ G)$ stands for $(F \equiv G)$, $(F \supset G)$, $(F \wedge G)$, or $(F \vee G)$:

$$\begin{aligned} E|_\epsilon &= E & (*\,F)|_{0.\tau} &= F|_\tau \\ (F * G)|_{0.\tau} &= F|_\tau & (F * G)|_{1.\tau} &= G|_\tau \\ P(s_0, \ldots, s_n)|_{i.\tau} &= s_i|_\tau & (s \approx t)|_{0.\tau} &= s|_\tau \\ f(s_0, \ldots, s_n)|_{i.\tau} &= s_i|_\tau & (s \approx t)|_{1.\tau} &= t|_\tau \end{aligned}$$

Given a formula or a term E, a position τ in $\Pi(E)$, and a formula or a term e, we will denote by $E[e]_\tau$ the result of replacement of $E|_\tau$ with e:

$$\begin{aligned} E[e]_\epsilon &= e & (*\,F)[e]_{0.\tau} &= *\,F[e]_\tau \\ (F * G)[e]_{0.\tau} &= F[e]_\tau * G & (F * G)[e]_{1.\tau} &= F * G[e]_\tau \\ P(s_0, \ldots, s_n)[e]_{i.\tau} &= P(s_0, \ldots, s_i[p]_\tau, \ldots, s_n) & (s \approx t)[e]_{0.\tau} &= s[e]_\tau \approx t \\ f(s_0, \ldots, s_n)[e]_{i.\tau} &= f(s_0, \ldots, s_i[p]_\tau, \ldots, s_n) & (s \approx t)[e]_{1.\tau} &= s \approx t[e]_\tau \end{aligned}$$

The expression e must be a term if $\tau \in \Pi_{\mathbf{t}}(E)$, and a formula otherwise. Free variables of e may become bound in $F[e]_\tau$.

4 Local validity and local properties

Given a formula F, a position $\pi \in \Pi_{\mathbf{F}}(F)$, and a formula U, we define the *local image* of U w.r.t. F and π, denoted $\langle U \rangle_\pi^F$, as follows:

$$\begin{aligned} \langle U \rangle_{0.\pi}^{F \equiv G} &= \langle U \rangle_\pi^F & \langle U \rangle_{1.\pi}^{F \equiv G} &= \langle U \rangle_\pi^G & \langle U \rangle_{0.\pi}^{\forall x F} &= \forall x\, \langle U \rangle_\pi^F \\ \langle U \rangle_{0.\pi}^{F \supset G} &= G \vee \langle U \rangle_\pi^F & \langle U \rangle_{1.\pi}^{F \supset G} &= F \supset \langle U \rangle_\pi^G & \langle U \rangle_{0.\pi}^{\exists x F} &= \forall x\, \langle U \rangle_\pi^F \\ \langle U \rangle_{0.\pi}^{F \wedge G} &= G \supset \langle U \rangle_\pi^F & \langle U \rangle_{1.\pi}^{F \wedge G} &= F \supset \langle U \rangle_\pi^G & \langle U \rangle_{0.\pi}^{\neg F} &= \langle U \rangle_\pi^F \\ \langle U \rangle_{0.\pi}^{F \vee G} &= G \vee \langle U \rangle_\pi^F & \langle U \rangle_{1.\pi}^{F \vee G} &= F \vee \langle U \rangle_\pi^G & \langle U \rangle_\varepsilon^F &= U \end{aligned}$$

Roughly, the formula $\langle U\rangle^F_\pi$ says "U is true at the position π in F". Note that this formula does not depend on the subformula $F|_\pi$. For a position $\pi \in \Pi_{\mathbf{t}}(F)$, we define $\langle U\rangle^F_\pi$ to be $\langle U\rangle^F_{\widehat{\pi}}$, where $\widehat{\pi}$ is the longest prefix of π in $\Pi_{\mathbf{F}}(F)$.

Example 3. Let F be the formula

$$\forall x\,(x \in \mathbb{N} \supset \forall n\,(n \in \mathbb{N} \supset (x \approx \mathbf{fib}(n) \equiv \\ \equiv ((n \leq 1 \wedge x \approx 1) \vee x \approx (\mathbf{fib}(n-1) + \mathbf{fib}(n-2))))))$$

This formula represents a recursive definition. We want to verify that the arguments $(n-1)$ and $(n-2)$ satisfy the guards of the definition and are strictly less than n.

Consider the second argument. Let π denote its position, 0.1.0.1.1.1.1.1.0. We want to prove $\langle (n-2) \in \mathbb{N} \wedge (n-2) < n\rangle^F_\pi$. The latter formula is equal to

$$\forall x\,(x \in \mathbb{N} \supset \forall n\,(n \in \mathbb{N} \supset ((n \leq 1 \wedge x \approx 1) \vee ((n-2) \in \mathbb{N} \wedge (n-2) < n))))$$

But this formula is false given $n = x = 0$. And that reveals an error in our definition: $x = 0$ makes false the left side of the disjunction $F|_{0.1.0.1.1}$, so we have to consider the right side with $n = 0$ in order to evaluate the truth value of the whole disjunction. Now it is easy to build a good definition F' of **fib**:

$$\forall x\,(x \in \mathbb{N} \supset \forall n\,(n \in \mathbb{N} \supset (x \approx \mathbf{fib}(n) \equiv \\ \equiv ((n \leq 1 \wedge x \approx 1) \vee (n \geq 2 \wedge x \approx (\mathbf{fib}(n-1) + \mathbf{fib}(n-2)))))))$$

Obviously, the local image $\langle (n-2) \in \mathbb{N} \wedge (n-2) < n\rangle^{F'}_{0.1.0.1.1.1.1.1.1.0}$ is a valid formula:

$$\forall x\,(x \in \mathbb{N} \supset \forall n\,(n \in \mathbb{N} \supset \\ \supset ((n \leq 1 \wedge x \approx 1) \vee (n \geq 2 \supset ((n-2) \in \mathbb{N} \wedge (n-2) < n)))))$$

Lemma 1. *For any F, $\pi \in \Pi(F)$, and a formula U, $\forall U \vdash \langle U\rangle^F_\pi$.*

Proof. Here, $\forall U$ denotes the universal closure of U. The formula $\langle U\rangle^F_\pi$ is equivalent to a universally quantified disjunction and U is a positive component of this disjunction. □

Lemma 2. (local modus ponens) $\vdash \langle U \supset V\rangle^F_\pi \supset (\langle U\rangle^F_\pi \supset \langle V\rangle^F_\pi)$

Lemma 2 can be proved by a simple induction on the length of π.

The lemmas above show that we can consistently reason about local properties. They are powerful enough to prove some interesting corollaries.

Corollary 1. $\vdash \langle U \equiv V\rangle^F_\pi \supset (\langle U\rangle^F_\pi \equiv \langle V\rangle^F_\pi)$

Proof. By Lemma 1 we have $\vdash \langle (U \equiv V) \supset (U \supset V)\rangle^F_\pi$. Hence by Lemma 2, $\vdash \langle (U \equiv V)\rangle^F_\pi \supset (\langle U \supset V\rangle^F_\pi)$. Again by local *modus ponens*, $\vdash \langle (U \equiv V)\rangle^F_\pi \supset (\langle U\rangle^F_\pi \supset \langle V\rangle^F_\pi)$. In the same way, $\vdash \langle (U \equiv V)\rangle^F_\pi \supset (\langle V\rangle^F_\pi \supset \langle U\rangle^F_\pi)$. □

Corollary 2. $\vdash \langle U \wedge V\rangle_\pi^F \equiv (\langle U\rangle_\pi^F \wedge \langle V\rangle_\pi^F)$

Proof. In order to prove the necessity, we take the propositional tautologies $(U \wedge V) \supset U$ and $(U \wedge V) \supset V$. In order to prove the sufficiency, we take the propositional tautology $U \supset (V \supset (U \wedge V))$. Then we "immerse" a chosen tautology inside the formula F by Lemma 1 and apply local *modus ponens*. □

Corollary 3. *For any quantifier-free context C,*

$$\vdash \big(\langle U_1 \equiv V_1\rangle_\pi^F \wedge \cdots \wedge \langle U_n \equiv V_n\rangle_\pi^F \wedge \langle t_1 \approx s_1\rangle_\pi^F \wedge \cdots \wedge \langle t_m \approx s_m\rangle_\pi^F\big) \supset \\ \supset \langle C[U_1, \ldots, U_n, t_1, \ldots, t_m] \equiv C[V_1, \ldots, V_n, s_1, \ldots, s_m]\rangle_\pi^F$$

The term "context" stands here for a formula with "holes", in which formulas or terms can be inserted, completing the context up to a well-formed formula. The corollary can be proved similarly to previous statements.

The key property of local images is given by the following theorem.

Theorem 1. *For any formulas F, U, V*

$$\begin{aligned} \pi \in \Pi_{\mathbf{F}}(F) \quad &\Rightarrow \quad \vdash \langle U \equiv V\rangle_\pi^F \supset (F[U]_\pi \equiv F[V]_\pi) \\ \pi \in \Pi_{\mathbf{F}}^+(F) \quad &\Rightarrow \quad \vdash \langle U \supset V\rangle_\pi^F \supset (F[U]_\pi \supset F[V]_\pi) \\ \pi \in \Pi_{\mathbf{F}}^-(F) \quad &\Rightarrow \quad \vdash \langle V \supset U\rangle_\pi^F \supset (F[U]_\pi \supset F[V]_\pi) \end{aligned}$$

This theorem is proved by induction on the length of π. The proof is quite straightforward and we omit it because of lack of space.

By Theorem 1, we can safely replace subformulas not only by equivalent formulas but by locally equivalent ones as well. Note that the inverse of the theorem holds in the propositional logic: $\vdash_0 \langle U \equiv V\rangle_\pi^F \equiv (F[U]_\pi \equiv F[V]_\pi)$. Local equivalence is there a criterion of substitutional equivalence. It is not the case for the first-order logic, where $(\exists x\, x \approx 0)$ is equivalent to $(\exists x\, x \approx 1)$.

Remark 1. In what follows, we often apply Theorem 1 and related results to positions from $\Pi_{\mathbf{t}}$, having in mind the position of the enclosing atomic formula. Note that any statement which is locally true in a term position is also locally true in the position of the enclosing atomic formula, since the local images are the same.

Corollary 4. *For any formula F, a position $\pi \in \Pi_{\mathbf{t}}(F)$, and terms s and t,*

$$\vdash \langle s \approx t\rangle_\pi^F \supset (F[s]_\pi \equiv F[t]_\pi)$$

Follows from Theorem 1 and Corollary 3.

Corollary 5. *For any formula F, a position $\pi \in \Pi_{\mathbf{F}}(F)$, and formulas U, V*

$$\begin{array}{rr} \vdash \langle U\rangle_\pi^F \supset (F[V]_\pi \equiv F[U \wedge V]_\pi) & \vdash \langle U\rangle_\pi^F \supset (F[V]_\pi \equiv F[U \supset V]_\pi) \\ \vdash \langle V \supset U\rangle_\pi^F \supset (F[V]_\pi \equiv F[U \wedge V]_\pi) & \vdash \langle U \supset V\rangle_\pi^F \supset (F[V]_\pi \equiv F[U \vee V]_\pi) \end{array}$$

Consider a closed formula H of the form $\forall \boldsymbol{x}\,(C \supset (A \equiv D))$, where A is an atomic formula. Consider a formula F and a position $\pi \in \mathit{\Pi}_{\mathbf{A}}(F)$ such that $F|_\pi = A\sigma$ for some substitution σ. If we can prove $\langle C\sigma\rangle^F_\pi$, then we have $\langle A\sigma \equiv D\sigma\rangle^F_\pi$ by Lemma 1 and Corollary 2 (provided that H is among the premises). Then we can replace $A\sigma$ with $D\sigma$ by Theorem 1 (we generalize this technique in the following section). Returning to Example 3, we can guarantee that such an expansion is always possible (since $\langle n-1 \in \mathbb{N} \ \wedge\ n-2 \in \mathbb{N}\rangle^F_\pi$ holds) and is never infinite (since $\langle n-1 < n \ \wedge\ n-2 < n\rangle^F_\pi$ holds).

However, the notion of a local image introduced above has a disadvantage: it is not invariant w.r.t. transformations at adjacent positions.

Example 4. Since $\langle A\rangle^{A\wedge A}_0$ is valid, $(A\wedge A)$ is equivalent to $(\top\wedge A)$ by Theorem 1. But $\langle A\rangle^{A\wedge A}_1$ is also valid, whereas $\langle A\rangle^{\top\wedge A}_1$ is not.

Generally, we can build a formula F whose two subformulas U and V assure certain local properties for each other. Using these properties, we replace U with a locally equivalent formula U'. But thus we can lose the local properties of V.

This does not play an important role when we consider one-time transformations, e.g. simplifications. Indeed, one should check that simplification is possible just before doing it. But there are also certain local properties that we would prefer keep intact during the entire proof.

For example, we can verify the ontological correctness of a given occurrence of a function symbol in the initial task and it is quite desirable to preserve further this correctness in order to expand the definition of that symbol at any moment, without extra verifications.

To that aim, we slightly change the definition of a local image in such a way that only the formulas at *precedent* positions get into the context. Psychologically, this is natural, since assertions of that kind (type declarations, limits, etc) are usually written before "significant" formulas.

The *directed local image* of a formula U w.r.t. a formula F and a position $\pi \in \mathit{\Pi}_{\mathbf{F}}(F)$, denoted $\langle\!|U|\!\rangle^F_\pi$, is defined as follows:

$$\begin{array}{lll}
\langle\!|U|\!\rangle^{F\equiv G}_{0.\pi} = \langle\!|U|\!\rangle^F_\pi & \langle\!|U|\!\rangle^{F\equiv G}_{1.\pi} = \langle\!|U|\!\rangle^G_\pi & \langle\!|U|\!\rangle^{\forall x F}_{0.\pi} = \forall x\,\langle\!|U|\!\rangle^F_\pi \\
\langle\!|U|\!\rangle^{F\supset G}_{0.\pi} = \langle\!|U|\!\rangle^F_\pi & \langle\!|U|\!\rangle^{F\supset G}_{1.\pi} = F \supset \langle\!|U|\!\rangle^G_\pi & \langle\!|U|\!\rangle^{\exists x F}_{0.\pi} = \forall x\,\langle\!|U|\!\rangle^F_\pi \\
\langle\!|U|\!\rangle^{F\wedge G}_{0.\pi} = \langle\!|U|\!\rangle^F_\pi & \langle\!|U|\!\rangle^{F\wedge G}_{1.\pi} = F \supset \langle\!|U|\!\rangle^G_\pi & \langle\!|U|\!\rangle^{\neg F}_{0.\pi} = \langle\!|U|\!\rangle^F_\pi \\
\langle\!|U|\!\rangle^{F\vee G}_{0.\pi} = \langle\!|U|\!\rangle^F_\pi & \langle\!|U|\!\rangle^{F\vee G}_{1.\pi} = F \vee \langle\!|U|\!\rangle^G_\pi & \langle\!|U|\!\rangle^F_\varepsilon = U
\end{array}$$

For a position $\pi \in \mathit{\Pi}_{\mathbf{t}}(F)$, we define $\langle\!|U|\!\rangle^F_\pi$ to be $\langle\!|U|\!\rangle^F_{\widehat{\pi}}$, where $\widehat{\pi}$ is the longest prefix of π in $\mathit{\Pi}_{\mathbf{F}}(F)$.

First, note that all statements proved so far about "indirected" images hold for directed ones, too. In some sense, directed image is just a reduction, with some conditions and alternatives eliminated. This is illustrated by the following trivial lemma.

Lemma 3. $\quad \vdash \langle\!|U|\!\rangle^F_\pi \supset \langle U\rangle^F_\pi$

Theorem 2. *For any formula F and two adjacent $\pi, \tau \in \Pi_{\mathbf{F}}(F)$,*

$$\vdash \langle\!|U \equiv V|\!\rangle^{F}_{\pi} \supset \left(\langle\!|W|\!\rangle^{F[U]_\pi}_{\tau} \equiv \langle\!|W|\!\rangle^{F[V]_\pi}_{\tau}\right)$$

Proof. We proceed by induction on the length of π. It is easy to see that, if τ textually precedes π, then the formulas $\langle\!|W|\!\rangle^{F[U]_\pi}_{\tau}$ and $\langle\!|W|\!\rangle^{F[V]_\pi}_{\tau}$ are identical. So we can suppose that π textually precedes τ, that is, there exist ω, μ, and η such that $\pi = \omega.0.\mu$ and $\tau = \omega.1.\eta$. It is easy to see that we can reduce our problem to

$$\vdash \langle\!|U \equiv V|\!\rangle^{G*H}_{0.\mu} \supset \left(\langle\!|W|\!\rangle^{(G*H)[U]_{0.\mu}}_{1.\eta} \equiv \langle\!|W|\!\rangle^{(G*H)[V]_{0.\mu}}_{1.\eta}\right)$$

where $(G * H) = F|_\omega$. The latter is equivalent to

$$\vdash \langle\!|U \equiv V|\!\rangle^{G}_{\mu} \supset \left(\langle\!|W|\!\rangle^{G[U]_\mu * H}_{1.\eta} \equiv \langle\!|W|\!\rangle^{G[V]_\mu * H}_{1.\eta}\right)$$

and then to

$$\vdash \langle\!|U \equiv V|\!\rangle^{G}_{\mu} \supset \left((G[U]_\mu \star \langle\!|W|\!\rangle^{H}_{\eta}) \equiv (G[V]_\mu \star \langle\!|W|\!\rangle^{H}_{\eta})\right)$$

where $\star$ is either $\supset$ or $\vee$, in dependence of $*$. By Lemma 3 and Theorem 1, $\langle\!|U \equiv V|\!\rangle^{G}_{\mu}$ implies $(G[U]_\mu \equiv G[V]_\mu)$, hence the claim is proved. □

Corollary 6. *For any formula F and two adjacent $\pi, \tau \in \Pi_{\mathbf{t}}(F)$,*

$$\vdash \langle\!|s \approx t|\!\rangle^{F}_{\pi} \supset \left(\langle\!|W|\!\rangle^{F[s]_\pi}_{\tau} \equiv \langle\!|W|\!\rangle^{F[t]_\pi}_{\tau}\right)$$

Finally, we introduce the notion of *local substitution*. Let H be a formula such that no quantifier occurs in H in the scope of another quantifier over the same variable. Given a position $\pi \in \Pi_{\mathbf{F}}(H)$, the result of local substitution $H[\sigma]_\pi$ is defined as follows:

$$\begin{aligned}
F[\sigma]_\varepsilon &= F & (F * G)[\sigma]_{0.\tau} &= F[\sigma]_\tau * G \\
(\neg F)[\sigma]_{0.\tau} &= \neg F[\sigma]_\tau & (F * G)[\sigma]_{1.\tau} &= F * G[\sigma]_\tau \\
(\forall x\, F)[\sigma]_{0.\tau} &= (F[x/x\sigma])[\sigma]_\tau & (\forall y\, F)[\sigma]_{0.\tau} &= \forall y\, F[\sigma]_\tau \\
(\exists x\, F)[\sigma]_{0.\tau} &= (F[x/x\sigma])[\sigma]_\tau & (\exists y\, F)[\sigma]_{0.\tau} &= \exists y\, F[\sigma]_\tau
\end{aligned}$$

where $x\sigma \neq x$ and $y\sigma = y$ in the last four equations, i.e. we eliminate the quantifiers over the instantiated variables. Here and below, we will assume that $x\sigma$ is free for x in F and further, σ does not instantiate any variable that occurs in one of the substitutes of σ.

When applied without restrictions, local substitutions may produce illegal instances (e.g. when variables of opposite polarities are instantiated). Also, local substitutions do not preserve local properties in adjacent positions. Consider the formula $F = \forall x\, P(x) \wedge A$ and the substitution $\sigma = [s/x]$ to be applied in F at $\pi = 1.0$, so that $F[\sigma]_\pi = (P(s) \wedge A)$. The atom A has the local property $\forall x\, P(x)$ in F but loses this property in $F[\sigma]_\pi$ — something we would like to avoid.

Therefore, we introduce a more fine-grained operation. As before, let H be a formula such that no quantifier occurs in H in the scope of another quantifier over the same variable, and π be a position in $\Pi_{\mathbf{F}}(H)$.

$$
\begin{array}{rlrl}
(F \supset G)[\sigma]^+_{0.\tau} &= F[\sigma]^-_\tau \supset \bot & (F \supset G)[\sigma]^+_{1.\tau} &= F \supset G[\sigma]^+_\tau \\
(F \vee G)[\sigma]^+_{0.\tau} &= F[\sigma]^+_\tau \vee \bot & (F \vee G)[\sigma]^+_{1.\tau} &= F \vee G[\sigma]^+_\tau \\
(F \wedge G)[\sigma]^+_{0.\tau} &= F[\sigma]^+_\tau \wedge G & (F \wedge G)[\sigma]^+_{1.\tau} &= F \wedge G[\sigma]^+_\tau \\
(\exists x\, F)[\sigma]^+_{0.\tau} &= (F[x/x\sigma])[\sigma]^+_\tau & (F \equiv G)[\sigma]^+_\tau &= F \equiv G \\
(\exists y\, F)[\sigma]^+_{0.\tau} &= \exists y\, F[\sigma]^+_\tau & (\neg F)[\sigma]^+_{0.\tau} &= \neg F[\sigma]^-_\tau \\
(\forall z\, F)[\sigma]^+_{0.\tau} &= \forall z\, F[\sigma]^+_\tau & F[\sigma]^+_\varepsilon &= F
\end{array}
$$

$$
\begin{array}{rlrl}
(F \supset G)[\sigma]^-_{0.\tau} &= F[\sigma]^+_\tau \supset G & (F \supset G)[\sigma]^-_{1.\tau} &= F \supset G[\sigma]^-_\tau \\
(F \vee G)[\sigma]^-_{0.\tau} &= F[\sigma]^-_\tau \vee G & (F \vee G)[\sigma]^-_{1.\tau} &= F \vee G[\sigma]^-_\tau \\
(F \wedge G)[\sigma]^-_{0.\tau} &= F[\sigma]^-_\tau \wedge \top & (F \wedge G)[\sigma]^-_{1.\tau} &= F \wedge G[\sigma]^-_\tau \\
(\forall x\, F)[\sigma]^-_{0.\tau} &= (F[x/x\sigma])[\sigma]^-_\tau & (F \equiv G)[\sigma]^-_\tau &= F \equiv G \\
(\forall y\, F)[\sigma]^-_{0.\tau} &= \forall y\, F[\sigma]^-_\tau & (\neg F)[\sigma]^-_{0.\tau} &= \neg F[\sigma]^+_\tau \\
(\exists z\, F)[\sigma]^-_{0.\tau} &= \exists z\, F[\sigma]^-_\tau & F[\sigma]^-_\varepsilon &= F
\end{array}
$$

where $x\sigma \neq x$ and $y\sigma = y$. For a position $\pi \in \Pi_{\mathbf{t}}(H)$, we define $H[\sigma]^+_\pi = H[\sigma]^+_{\widehat{\pi}}$ and $H[\sigma]^-_\pi = H[\sigma]^-_{\widehat{\pi}}$, where $\widehat{\pi}$ is the longest prefix of π in $\Pi_{\mathbf{F}}(H)$.

These operations keep track of polarity of an occurrence in question and do not instantiate inappropriate variables. Also they eliminate subformulas in certain adjacent positions — exactly those ones which may lose their local properties after instantiation.

Lemma 4. *Let H be a formula such that no quantifier occurs in H in the scope of another quantifier over the same variable. Let π be a position in $\Pi(H)$ and σ, a substitution. Then we have:*

$$\vdash H[\sigma]^+_\pi \supset H \qquad\qquad \vdash H \supset H[\sigma]^-_\pi$$

Theorem 3. *Let H be a formula such that no quantifier occurs in H in the scope of another quantifier over the same variable. Let π be a position in $\Pi(H)$ and σ, a substitution. For any polarity $s \in \{+, -\}$ and any position $\tau \in \Pi_{\mathbf{A}}(H[\sigma]^s_\pi)$, either $(H[\sigma]^s_\pi)|_\tau = \top$ or there exists a position $\tau' \in \Pi_{\mathbf{A}}(H)$ such that the following holds:*

Let μ be the longest common prefix of π and τ'. Let σ' be a substitution such that for any varaible x, if a quantifier over x is eliminated in $H[\sigma]^s_\mu$, then $x\sigma' = x\sigma$, otherwise $x\sigma' = x$. Then $(H[\sigma]^s_\pi)|_\tau = (H|_{\tau'})\sigma'$ and

$$\vdash \langle\!|U|\!\rangle^H_{\tau'} \supset \langle\!|U\sigma'|\!\rangle^{H[\sigma]^s_\pi}_\tau$$

Proof. We can suppose without loss of generality that $\pi \in \Pi_{\mathbf{F}}(H)$ (otherwise $\widehat{\pi}$ should be taken instead of π). We will prove this lemma by induction on

the length of π. In the base case ($\pi = \epsilon$), we take $\tau' = \tau$ and $\sigma' = \iota$, the trivial substitution. Thus the claim is obviously true. Otherwise we consider three typical cases.

Case $H = (F \supset G)$, $\pi = 0.\pi_0$, $s = -$, $H[\sigma]^s_\pi = F[\sigma]^+_{\pi_0} \supset G$, $\tau = 1.\tau_0$. We take $\tau' = \tau$ and $\sigma' = \iota$. Obviously, $(H[\sigma]^-_\pi)|_\tau = G|_{\tau_0} = (H|_{\tau'})\sigma'$. Furthermore, $\langle\!|U|\!\rangle^H_{\tau'} = F \supset \langle\!|U|\!\rangle^G_{\tau_0}$ and $\langle\!|U\sigma'|\!\rangle^{H[\sigma]^s_\pi}_\tau = F[\sigma]^+_{\pi_0} \supset \langle\!|U|\!\rangle^G_{\tau_0}$. By Lemma 4, $\vdash F[\sigma]^+_{\pi_0} \supset F$, and the claim holds. Note that we could not make the final step in the case $s = +$, and therefore we had to define $H[\sigma]^+_\pi = F[\sigma]^-_{\pi_0} \supset \bot$.

Case $H = (F \supset G)$, $\pi = 1.\pi_0$, $s = +$, $H[\sigma]^s_\pi = F \supset G[\sigma]^+_{\pi_0}$, $\tau = 1.\tau_0$. By the induction hypothesis, there exist $\tau'_0 \in \Pi_{\mathbf{A}}(G)$ and a substitution σ' such that $(G[\sigma]^+_{\pi_0})|_{\tau_0} = (G|_{\tau'_0})\sigma'$ and $\vdash \langle\!|U|\!\rangle^G_{\tau'_0} \supset \langle\!|U\sigma'|\!\rangle^{G[\sigma]^+_{\pi_0}}_{\tau_0}$. Then we take $\tau' = 1.\tau'_0$ and obtain $(H[\sigma]^+_\pi)|_\tau = (H|_{\tau'})\sigma'$. Moreover, $\langle\!|U|\!\rangle^H_{\tau'}$ (equal to $F \supset \langle\!|U|\!\rangle^G_{\tau'_0}$) implies $\langle\!|U\sigma'|\!\rangle^{H[\sigma]^+_\pi}_\tau$ (equal to $F \supset \langle\!|U|\!\rangle^{G[\sigma]^+_{\pi_0}}_{\tau_0}$).

Case $H = (\forall x\, F)$, $\pi = 0.\pi_0$, $s = -$, $H[\sigma]^s_\pi = (F[x/x\sigma])[\sigma]^-_{\pi_0}$, $\tau = \tau_0$. Let F' stand for $F[x/x\sigma]$. By the induction hypothesis, there exist some $\tau'_0 \in \Pi_{\mathbf{A}}(F')$ and a substitution σ'_0 such that $(F'[\sigma]^-_{\pi_0})|_{\tau_0} = (F'|_{\tau'_0})\sigma'_0$ and for any V, $\vdash \langle\!|V|\!\rangle^{F'}_{\tau'_0} \supset \langle\!|V\sigma'_0|\!\rangle^{F'[\sigma]^-_{\pi_0}}_{\tau_0}$. Then we take $\tau' = 0.\tau'_0$ and $\sigma' = \sigma'_0 \circ [x/x\sigma]$ (recall that σ'_0 does not instantiate variables from $x\sigma$). We obtain $(H[\sigma]^-_\pi)|_\tau = (F'[\sigma]^-_{\pi_0})|_{\tau_0} = (F'|_{\tau'_0})\sigma'_0 = (F|_{\tau'_0})\sigma' = (H|_{\tau'})\sigma'$. Further, the local image $\langle\!|U|\!\rangle^H_{\tau'}$ (equal to $\forall x\, \langle\!|U|\!\rangle^F_{\tau'_0}$) implies $(\langle\!|U|\!\rangle^F_{\tau'_0})[x/x\sigma]$. The latter formula is equal to $\langle\!|U[x/x\sigma]|\!\rangle^{F'}_{\tau'_0}$ and thus implies $\langle\!|(U[x/x\sigma])\sigma'_0|\!\rangle^{F'[\sigma]^-_{\pi_0}}_{\tau_0}$, that is, $\langle\!|U\sigma'|\!\rangle^{H[\sigma]^-_\pi}_\tau$. □

Informally, Theorem 3 says that any atom in H that "survives" instantiation (i.e. is not replaced with a boolean constant) preserves its local properties, which are instantiated together with the atom.

5 Applying local properties

Let us consider a formula of the form $H[F]_\pi$ such that no quantifier occurs in it in the scope of another quantifier over the same variable. Let σ be a substitution. By Theorem 3, there exist a formula H', a position π', and a substitution σ' such that $(H[F]_\pi)[\sigma]^-_\pi \equiv H'[F\sigma']_{\pi'}$ and every local property of F in H is preserved (modulo instantiation) in H'. (While π is not a position of atom in $H[F]_\pi$, we can take an atom $P(\boldsymbol{x})$, where P is a new predicate symbol and $\boldsymbol{x}$ are the free variables of F, and prove $(H[P(\boldsymbol{x})]_\pi)[\sigma]^-_\pi \equiv H'[P(\boldsymbol{x})\sigma']_{\pi'}$. Note that $P(\boldsymbol{x})$ cannot turn into a boolean constant in $(H[P(\boldsymbol{x})]_\pi)[\sigma]^-_\pi$. Then we have $\forall \boldsymbol{x}\,(P(\boldsymbol{x}) \equiv F) \vdash (H[F]_\pi)[\sigma]^-_\pi = H'[F\sigma']_{\pi'}$, by Lemma 1 and Theorem 1. Since P is a new symbol, the premise $\forall \boldsymbol{x}\,(P(\boldsymbol{x}) \equiv F)$ can be discarded.) By Lemma 4, $H[F]_\pi$ implies $H'[F\sigma']_{\pi'}$.

We can prove that $H'[F\sigma']_{\pi'}$ implies $\exists \boldsymbol{x}'\,(F\sigma') \vee H'[\bot]_{\pi'}$, where $\boldsymbol{x}'$ are the free variables of $F\sigma'$. Indeed, $H'[F\sigma']_{\pi'}$ implies $\exists \boldsymbol{x}'\,(F\sigma') \vee H'[F\sigma']_{\pi'}$, which is equivalent to $\forall \boldsymbol{x}'\,(\neg F\sigma') \supset H'[F\sigma']_{\pi'}$, which is equivalent to $\forall \boldsymbol{x}'\,(\neg F\sigma') \supset H'[\bot]_{\pi'}$ by Theorem 1. Therefore, $H[F]_\pi$ implies $\neg H'[\bot]_{\pi'} \supset \exists \boldsymbol{x}'\,(F\sigma')$.

This provides us with a handy tool to test applicability of definitions in a ForTheL text. Consider a section $\mathbb{A}$ and suppose that Γ is the set of sections which logically precede $\mathbb{A}$ in the text. Let G be the formula image of $\mathbb{A}$. Let $P(\boldsymbol{s})$ occur in G in a position μ. Now, suppose that $\mathbb{D} \in \Gamma$ is a definition for the predicate symbol P. Quite naturally, the formula image of $\mathbb{D}$ is of the form $\forall \boldsymbol{x}_1(H_1 \supset \ldots \forall \boldsymbol{x}_k(H_k \supset (P(\boldsymbol{x}_{1,\ldots,k}) \equiv D))\ldots)$. By previous, it suffices to prove $\Gamma \vdash \langle\!|H_1\sigma \supset \ldots H_k\sigma \supset \bot|\!\rangle^G_\mu$, where σ is the substitution $[\boldsymbol{x}_{1,\ldots,k}/\boldsymbol{s}]$, to obtain $\Gamma \vdash \langle\!|P(\boldsymbol{s}) \equiv D\sigma|\!\rangle^G_\mu$. Then G is equivalent to $G[D\sigma]_\mu$, that is, we can *apply* the definition $\mathbb{D}$ to $P(\boldsymbol{s})$. Moreover, all the local properties of terms and subformulas of D in $\mathbb{D}$, instantiated with σ, hold in $D\sigma$ in $G[D\sigma]_\mu$.

In a similar fashion, we define applicability for other forms of ForTheL definitions and signature extensions. Note that the substitution σ and the position of the local instantiation in $|\mathbb{D}|$ are unambiguously determined by the form of $\mathbb{D}$. Using the method described above, we can test any logical predecessor of $\mathbb{A}$ for applicability at a given position in $|\mathbb{A}|$, but then we have to choose an appropriate local instantiation ourselves.

Now, a section $\mathbb{A}$ is *ontologically correct* in view of Γ if and only if every occurrence of a non-logical symbol in $|\mathbb{A}|$ either has an applicable definition or signature extension in Γ or is the principal occurrence in a definition or signature extension $\mathbb{A}$ (which means that $\mathbb{A}$ introduces that very symbol).

A ForTheL text is *ontologically correct* whenever each section in it is ontologically correct in view of its logical predecessors.

6 Conclusion

We have introduced the notion of a locally valid statement for the classical first-order logic and showed how it can be used to reason about the interiors of a formula. In particular, we proved that a locally true equivalence is a sufficient condition for an equivalent transformation of a subformula. The local validity of a statement is expressed with the help of *local images* which can be regarded as a syntactical formalization of the notion of a logical context of the statement occurrence. Since locally equivalent transformations may break local properties of other occurrences, we introduced the notion of directed local validity which is invariant w.r.t. directed locally equivalent transformations. Finally, we defined the operation of local instantiation and showed that this transformation preserves directed local properties. Using this theoretical background, we gave a clear definition of an ontologically correct ForTheL text.

The proposed approach can be regarded as a way to handle partial relations and functions in a mathematical text. Instead of introducing special individual or truth values for undefinedness (as in Kleene's strong logic [6]), ontological correctness requires every term or atom to be well-defined *a priori*, by conformance to the guards of corresponding definitions. Using directed images and deductive techniques preserving local properties, we can guarantee that the text under consideration always stays well-defined. In our opinion, this corresponds well to the usual mathematical practice.

Of course, reasoning inside a formula is not a new idea. To our knowledge, related concepts were first introduced by L.G. Monk in [7] and were further developed in [8]. P.J. Robinson and J. Staples proposed a full-fledged inference system (so called "window inference") [9] which operated on subexpressions taking the surrounding context into account. This inference system has been generalized and extended by J. Grundy [10].

A common trait of the mentioned approaches is that the local context of an occurrence is represented by a set of formulas which are regarded as local premises for the position in question. Special inference rules are then needed to handle a local context and, what is worse, some "strong" transformations, e.g. replacing $A \vee B$ with $\neg A \supset B$, are required. The notion of local image, as described in this paper, seems to be lighter and less intrusive. In particular, the results of Section 4 are valid in intuitionistic logic, while the local contexts of [7] cannot be adapted for intuitionistic logic in any obvious way.

Moreover, the definition of a local image can be easily extended to a (uni)-modal language: $\langle U \rangle_{0.\pi}^{\Box F} = \Box \langle U \rangle_{\pi}^{F}$ and $\langle U \rangle_{0.\pi}^{\Diamond F} = \Box \langle U \rangle_{\pi}^{F}$, and similarly for directed images. Then the statements of Section 4 (local instantiation aside) can be proved in the modal logic **K**, hence in any normal modal logic.

Acknowledgements. This work is supported by the INTAS project 05-1000008-8144. Some parts were done within the scope of the project M/108-2007 in the framework of the joint French-Ukrainian programme "Egide-Dnipro".

References

1. Trybulec, A., Blair, H.: Computer assisted reasoning with Mizar. In: Proc. 9th International Joint Conference on Artificial Intelligence. (1985) 26–28
2. Barendregt, H.: Towards an interactive mathematical proof language. In Kamareddine, F., ed.: Thirty Five Years of Automating Mathematics, Heriot-Watt University, Edinburgh, Scotland, Kluwer Academic Publishers (2003) 25–36
3. Kamareddine, F., Nederpelt, R.P.: A Refinement of de Bruijn's Formal Language of Mathematics. Journal of Logic, Language and Information **13**(3) (2004) 287–340
4. Lyaletski, A., Paskevich, A., Verchinine, K.: SAD as a mathematical assistant — how should we go from here to there? Journal of Applied Logic **4**(4) (2006) 560–591
5. Lyaletski, A., Paskevich, A., Verchinine, K.: Theorem proving and proof verification in the system SAD. In Asperti, A., Bancerek, G., Trybulec, A., eds.: Mathematical Knowledge Management: Third International Conference, MKM 2004. Volume 3119 of Lecture Notes in Computer Science., Springer (2004) 236–250
6. Kleene, S.C.: Introduction to Metamathematics. Van Nostrand (1952)
7. Monk, L.G.: Inference rules using local contexts. Journal of Automated Reasoning **4**(4) (1988) 445–462
8. Corella, F.: What holds in a context? Journal of Automated Reasoning **10**(2) (1993) 79–93
9. Robinson, P.J., Staples, J.: Formalising the hierarchical structure of practical mathematical reasoning. Journal of Logic and Computation **3**(1) (1993) 47–61
10. Grundy, J.: Transformational hierarchical reasoning. The Computer Journal **39**(4) (1996) 291–302

The Utility of OpenMath

James H. Davenport*

Department of Computer Science, University of Bath, Bath BA2 7AY England
J.H.Davenport@bath.ac.uk,
WWW home page: http://staff.bath.ac.uk/masjhd

Abstract. OpenMath [5] is a standard for representing the *semantics* of mathematical objects. It differs from 'Presentation' MathML [7] in not being directly concerned with the presentation of the object, and from 'Content' MathML in being extensible. How should these extensions be performed so as to maximise the utility (which includes presentation) of OpenMath?

1 What is OpenMath?

"OpenMath is an emerging standard for representing mathematical objects with their semantics, allowing them to be exchanged between computer programs, stored in databases, or published on the worldwide web."[1]. In particular, OpenMath is extensible, unlike MathML 2.0[2] [7]. It achieves this by having an extensible collection of Content Dictionaries. "Content Dictionaries (CDs) are used to assign informal and formal semantics to all symbols used in the OpenMath objects. They define the symbols used to represent concepts arising in a particular area of mathematics" [5, section 1.3].

Notation 1 *By an* OpenMath CD *we will mean any document conforming to the formal syntax of [5].*

The status of an OpenMath content dictionary is one of the following [5, Section 4.2.1]:

- `official`: approved by the *OpenMath* society according to the procedure defined in section 4.5 (of [5]);

* This paper owes much to some questions of Paul Libbrecht, when we were both at the IMA Workshop "The Evolution of Mathematical Communication in the Age of Digital Libraries" — December 8–9, 2006. Thanks are due to the IMA, and particularly Robert Miner, for organising this workshop. Further comments, notably on section 6, are due to him [18] and Christian Gross [14]. Section 7 owes a lot to discussion with Prof. Vorobjov. Drs Naylor and Padegt also made useful suggestions.

[1] http://www.openmath.org/overview/index.html

[2] After this paper was submitted, a draft [8] of MathML 3.0 was produced, which bases content markup on OpenMath content dictionaries, and thus *is* extensible.

– `experimental`: under development, and thus liable to change;
– `private`: used by a private group of *OpenMath* users;
– `obsolete`: an obsolete Content Dictionary kept only for archival purposes[3].

Definition 1. *A Content Dictionary is said to be* public *if it is accessible from* `http://www.openmath/org` *and has one of the two status* `official` *or* `obsolete`*. Similarly, a symbol is said to be* public *if it is in a public CD.*

Note that this definition of *public* refers to the entire symbol, not just the name. Thus

```
<OMS name="sin" cd="transc1"/>
```

is a public symbol, whereas

```
<OMS name="sin" cd="http://www.camalsoft.com/G/transc1"/>
```

is not.

An OpenMath object, all of whose symbols are public, has fixed, permanent, semantics. Even if a CD changes status from `official` to `obsolete`, the semantics do not change (though it is quite likely that new software systems will not be able to interpret it, except in the name of compatibility[4]).

The OpenMath standard explicitly envisages that OpenMath applications can declare and negotiate the CDs (or CD groups) that they understand [5, Section 4.4.2]. In the absence of such negotiation[5], it might seem that the only OpenMath objects which can safely be exchanged are ones all of whose symbols are public (which we can abbreviate to *public* OpenMath objects). If every application had to convert from its semantics to those of the public CDs, there would be great inefficiency involved, especially if the aim was 'cut and paste' from one instance of an application to another instance of the same application (e.g. from mine to yours, or from today's to tomorrow's, or from version x to version ++x or ...). Equally, two different applications may be "sufficiently similar" that each can understand the other's semantics directly.

2 A Pragmatic Interpretation

Definition 2. *A Content Dictionary is said to be* semi-public *if it is accessible from* `http://www.openmath/org` *or from an URI which resolves to a globally accessible URL, and the CD has one of the two status* `official` *or* `obsolete`*. Similarly, a symbol is said to be* semi-public *if it is in a semi-public CD.*

[3] This is the wording of [5]: the present author would be inclined to write "archival and compatibility purposes".

[4] "Compatibility is the last excuse for not fixing something that you have already admitted to be a bug" [25]. For OpenMath, declaring a CD obsolete and writing a new one with the 'bug' fixed removes even this excuse: see section 6.

[5] Which may well be impossible in a "cut and paste" scenario.

Thus

```
<OMS name="sin" cd="http://www.camalsoft.com/G/transc1"/>
```

appears to be a semi-public symbol, whereas

```
<OMS name="sin" cd="file://C:/camaljpff/G/transc1"/>
```

is not.

We said above that it *appeared* to be a semi-public symbol. That is because the definition is neither effective (we can try to look the symbol up, but who knows if the failure is transient or permanent) nor time-invariant: `camalsoft` may go bankrupt, or its managers may not comply with the OpenMath rules, and delete symbols or change the semantics of them. Hence the concept that can be effective is that of *apparently semi-public*, as applied to a CD or a symbol. However, an apparently semi-public symbol might not have any discernable semantics.

Definition 3. *A symbol is said to be* transitively public *if:*

1. *it is apparently semi-public;*
2. *its semantics can be deduced in terms of public symbols by (possibly many) applications of Formal Mathematical Properties (FMPs) contained in apparently semi-public CDs.*

Again, the definition is not time-invariant, for the same reasons as before. Also, it is not application-independent, since one application might be able to make deductions from FMPs that another could not. *However*, it is the semantics and utility of transitively public symbols that we are concerned with here, since these are ones that applications might reasonably encounter. This is what, effectively, is implied by the `cdbase` in the `OMOBJ` constructs quoted.

3 An example — arctan

One hypothetical example would be the following, for the system Derive[6], whose arctan function differs from the definition in [1]. As pointed out in [9], the two definitions could be related by the followingFMP.

```
<FMP>
  <OMOBJ cdbase="http://www.openmath.org/cd">
    <OMA>
      <OMS name="eq" cd="relation1"/>
      <OMA>
```

[6] As already stated in [9], this is not an issue of some algebra systems, such as Maple, being "right" and others, such as Derive, "wrong": merely that Derive has chosen a different set of branch cut behaviours from OpenMath. Provided the definitions are correct, the choice is one of taste, fortified with the occasional dash of Occam's razor.

```
        <OMS name="arctan" cd="http://www.softwarehouse.com/Derive/transc1"/>
        <OMVAR name="z"/>
      </OMA>
      <OMA>
        <OMS name="conjugate" cd="complex1"/>
        <OMA>
          <OMS name="arctan" cd="transc1"/>
          <OMA>
            <OMS name="conjugate" cd="complex1"/>
            <OMVAR name="z"/>
          </OMA>
        </OMA>
      </OMA>
    </OMA>
  </OMOBJ>
</FMP>
```

With this definition, a "sufficiently intelligent" (in fact it need not be that intelligent in this case) system would be able to understand OpenMath emitted from Derive containing Derive arctangents, encoded as follows:

```
<OMS name="arctan" cd="http://www.softwarehouse.com/Derive/transc1"/>
```

occurrences.

The designer of the Derive→OpenMath phrasebook is then faced with a set of alternatives.

1. Emit in terms of the public OpenMath symbol from `transc1`. This has the advantage that no Derive CD needs to be written, or, more importantly, maintained and kept available. Assuming that Derive can cancel double conjugation, it means that cutting and pasting from one Derive to another is not significantly more expensive. Some-one who is doing Derive $\overset{\text{OpenMath}}{\longrightarrow}$ LaTeX would be distinctly surprised by the results, since the arctan emitted by LaTeX would be (invisibly) one with OpenMath semantics, i.e. complex conjugation might appear in the LaTeX where there was none in the Derive.
2. Emit in terms of the Derive symbol defined above. This has the disadvantage that the CD[7] needs to be written and kept available. If the recipient is another Derive, it would presumably understand this. If the recipient is a "sufficiently clever" other algebra system conforming to OpenMath's semantics of arctan, the correct result will be achieved. If it has Derive's semantics, it will either notice this directly, or cancel the double conjugations. If it has different semantics, it will presumably know what to do.
   The interesting question is what an OpenMath⟶LaTeX phrasebook with no explicit Derive knowledge will do. It is unlikely to have the semantic processing capability to handle the FMP, though in this case it might. However,

[7] And the associated STS [11] file.

a plausible action by such a phrasebook would be to check the STS [11] file, observe that this function was unary from a set to itself (it might notice that the set was **C**, but this is irrelevant) and just print the name as a unary prefix function. Indeed, one could just observe that it was being used as a unary function, as is done in LeActiveMath [18, 24].

3. Ignore the problem, and emit `<OMS name="arctan" cd="transc1"/>`. Alas, this would be a very human reaction. Such a phrasebook would (if it met the other criteria) be entitled to describe itself as OpenMath-compliant, but it would certainly not meet the goal [5, Chapter 5] that "It is expected that the application's phrasebooks for the supported Content Dictionaries will be constructed such that the properties of the symbol expressed in the Content Dictionary are respected as far as possible for the given application domain".
4. Refuse to emit arctans, on the grounds that Derive's is different from OpenMath's. In view of the plausible solutions in the first two choices, this seems unnecessarily "dog-in-the-manger".

We should observe that the mathematically equivalent FMP

```
<FMP>
  <OMOBJ cdbase="http://www.openmath.org/cd">
    <OMA>
      <OMS name="eq" cd="relation1"/>
      <OMA>
        <OMS name="arctan" cd="transc1"/>
        <OMVAR name="z"/>
      </OMA>
      <OMA>
        <OMS name="conjugate" cd="complex1"/>
        <OMA>
          <OMS name="arctan" cd="http://www.softwarehouse.com/Derive/transc1"/>
          <OMA>
            <OMS name="conjugate" cd="complex1"/>
            <OMVAR name="z"/>
          </OMA>
        </OMA>
      </OMA>
    </OMA>
  </OMOBJ>
</FMP>
```

is less useful, as it expresses the 'known' `<OMS name="arctan" cd="transc1"/>` in terms of the 'unknown', rather than the other way round, and therefore requires more logical power to use In particular, the interpreting phrasebook would need to know that the inverse of conjugation is itself conjugation.

Note also that there is no need to define Derive's arctan in terms of the OpenMath one: we could define it directly (see Figure 1) in terms of log, as OpenMath's arctan is in `transc1`.

Fig. 1. Definition of an alternative arctan

```
<FMP>
  <OMOBJ cdbase="http://www.openmath.org/cd">
    <OMA>
      <OMS name="eq" cd="relation1"/>
      <OMA>
        <OMS name="arctan" cd="http://www.softwarehouse.com/Derive/transc1"/>
        <OMV name="z"/>
      </OMA>
      <OMA>
        <OMS name="times" cd="arith1"/>
        <OMA>
          <OMS name="divide" cd="arith1"/>
          <OMS name="one" cd="alg1"/>
          <OMA>
            <OMS name="times" cd="arith1"/>
            <OMI> 2 </OMI>
            <OMS name="i" cd="nums1"/>
          </OMA>
        </OMA>
        <OMA>
          <OMS name="ln" cd="transc1"/>
          <OMA>
            <OMS name="divide" cd="arith1"/>
            <OMA>
              <OMS name="plus" cd="arith1"/>
              <OMS name="one" cd="alg1"/>
              <OMA>
                <OMS name="times" cd="arith1"/>
                <OMS name="i" cd="nums1"/>
                <OMV name="z"/>
              </OMA>
            </OMA>
            <OMA>
              <OMS name="minus" cd="arith1"/>
              <OMS name="one" cd="alg1"/>
              <OMA>
                <OMS name="times" cd="arith1"/>
                <OMS name="i" cd="nums1"/>
                <OMV name="z"/>
              </OMA>
            </OMA>
          </OMA>
        </OMA>
      </OMA>
    </OMA>
  </OMOBJ>
</FMP>
```

4 Another example

Let us imagine a theorem prover specialised in results over the natural numbers: let us call it Euclid. Euclid's natural domain of reasoning is the positive integers $1, 2, \ldots$, which it refers to as $\mathbf{N}$. How should Euclid exports results such as "if the successor of a equals the successor of b, then $a = b$", i.e.

$$\forall a, b \in \mathbf{N}\, succ(a) = succ(b) \Rightarrow a = b? \tag{1}$$

Again, the designer of the Euclid→OpenMath phrasebook has various options.

1. Emit in terms of the OpenMath symbol, i.e. encode Euclid's $\mathbf{N}$ as

```
<OMA>
  <OMS name="setdiff" cd="set1"/>
  <OMS name="N" cd="setname1"/>
  <OMA>
    <OMS name="set" cd="set1"/>
    <OMS name="zero" cd="alg1"/>
  </OMA>
</OMA>
```

   This is certainly accurate, but would cause some grief on re-importing into Euclid, since:
   - $\mathbf{N}$ (in the OpenMath sense) has no direct equivalent in Euclid, but has to be encoded as $\mathbf{N} \cup \{0\}$;
   - while expecting an algebra system to cancel double conjugations is reasonable, expecting a proof system to simplify $(\mathbf{N} \setminus \{0\}) \cup \{0\}$ is expecting rather more.
2. Emit in Euclid's own CD, e.g. with a definition as in figure 2. This has advantages as well as disadvantages.
   - Clearly it requires the CD to be written and maintained.
   - An OpenMath→LaTeX converter would probably render this as P. This might look well, but could be confused with

     ```
     <OMS name="P" cd="setname1"/>
     ```

     which is the set of primes[8], normally rendered as $\mathcal{P}$. A configurable OpenMath→LaTeX converter[9] would be able to get this right, and print $\mathbf{P}$.
3. Ignore the difficulty. This is clearly sub-human, rather than merely human, since a theorem-prover that emits incorrect statements could well be argued to be worse than useless.

We return to this issue in section 6.

[8] This is another example of the fact that an OpenMath symbol is the name *and* the CD.

[9] Such as the Notation Selection Tool [21, 22].

Fig. 2. Euclid's definition of **P** in terms of **N**

```
<FMP>
  <OMOBJ cdbase="http://www.openmath.org/cd">
    <OMA>
      <OMS name="eq" cd="relation1"/>
      <OMS name="P" cd="http://www.euclid.gr/CD"/>
      <OMA>
        <OMS name="setdiff" cd="set1"/>
        <OMS name="N" cd="setname1"/>
        <OMA>
          <OMS name="set" cd="set1"/>
          <OMS name="zero" cd="alg1"/>
        </OMA>
      </OMA>
    </OMA>
  </OMOBJ>
</FMP>
```

5 OpenMath and Notation

What use is OpenMath if one can't "see"[10] the results? Probably not much. How does one do it? One solution would be to make OpenMath do it.

> [...] was indicated as an expectation of Robert Miner at the W3C-Math f2f: if you find a CD, you should also have the notations with it ... so that you can present all the symbols in this CD. [18]

However, this begs the question: what is "the notation" [12]. A simple example is that of half-open intervals: the "anglo-saxon" $(0, 1]$ and the "french" $]0, 1]$. More subtly, there is the "anglo-saxon" use of Arcsin to denote a multi-valued function and arcsin to denote the corresponding[11] one-valued function, compared with the "french" notation which is the converse. It should be noted that, in this case, the OpenMath notation is even-handed: one is

```
<OMS name="arctan" cd="transc1"/>
```

the other is

```
<OMS name="arctan" cd="transc3"/>
```

and in both the "anglo-saxon" and "french" cases, one (or one's renderer) has to decide which to capitalise.

[10] Used as shorthand for "convert into a presentation", which may be displayed in various means, e.g. audio [23].

[11] But almost always with the branch cuts locally implicit, and often never stated at all, or changing silently from one printing to the next [1].

To avoid the charge of antigallicanism being levied against the author, let us also point out that there are differences due to subject: $\sqrt{-1}$ is i everywhere except in electrical engineering, where it is j, and so on.

Hence it is impossible for an OpenMath object to know, in a context-free way, how it should be rendered[10]. The best one could hope for is that, associated with an OpenMath CD, there could be a "default rendering" file, which would give *a* rendering for objects using this system, probably by translation into Presentation MathML as in David Carlisle's excellent style sheets [6]. This would have the advantage of allowing technologies such as those described in [16, 23] to process it.

6 Is even-handedness possible?

So far we have tried to be even-handed between various notations: OpenMath makes no choice between $(0, 1]$ and $]0, 1]$, nor says whether the mathematical Arcsin is a single-valued or multi-valued function, i.e. whether it corresponds to the `arcsin` from `transc1` or `transc3`. Even in the case of the branch cuts for arctan, where OpenMath has chosen one definition, it is possible to state the other definition, and do so on an even footing with OpenMath's own definition in `transc1`. Indeed it is possible that, as a result of the great branch cut riots of 2036[12], `transc1` is declared `obsolete`, `transc4` is promulgated with an FMP for arctan as in figure 1, and the authors of the softwarehouse CD change the FMP for arctan to be

```
<FMP>
  <OMOBJ cdbase="http://www.openmath.org/cd">
    <OMA>
      <OMS name="eq" cd="relation1"/>
      <OMS name="arctan" cd="http://www.softwarehouse.com/Derive/transc1"/>
      <OMS name="arctan" cd="transc4"/>
    </OMA>
  </OMOBJ>
</FMP>
```

and probably also mark that CD as `obsolete`. None of this would change the semantics of any OpenMath object.

However, the problem raised in section 4 is not so easily resolved: the question of whether $\mathbf{N}$ contains zero can, and indeed has [13], generate much debate. Many books, especially in German, suppose that $\mathbf{N}$ does *not* contain zero, e.g. the following.

[12] Caused by the requirement to move the branch cut in Network Time Protocol [20] and associated data formats. Rioters marched under the slogan "give us our two thousand one hundred and forty seven million, four hundred and eighty three thousand, six hundred and forty eight seconds back".

> Natürliche Zahlen sind die Zahlen, mit denen wir zählen: 1, 2, 3, 4, 5,.... Auf der Zahlengeraden bilden sie eine Abfolge von Punkten im Abstand 1, von 1 aus nach rechts gehend. Die Menge aller natürlichen Zahlen wird mit $\mathbf{N}$ bezeichnet. Weiters verwenden wir die Bezeichnung $\mathbf{N}_0 = \{0\} \cup \mathbf{N}$ für die natürlichen Zahlen zusammen mit der Zahl 0. [2, N]

Other sources are less definitive.

> Die natürlichen Zahlen sind die beim Zählen verwendeten Zahlen 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, usw. Oft wird auch die 0 (Null) zu den natürlichen Zahlen gerechnet. [3, Natürliche Zahl].

Indeed, the question is apparently as context-dependent as the rendering of $\sqrt{-1}$, but the impact of getting it wrong is much more misleading.

> Even German school books differ here. It depends on whom you ask. If you ask someone from number theory, he'd usually say that N is without 0. But if you ask someone from set theory, he'd say that N is with 0. It's just what is more convenient (i.e. shorter) for their usual work. [14]

It is clear that we have two different concepts, and several notations, as shown in Table 1.

Table 1. Natürliche Zahl

Concept	English	German (number)	German (set)	OpenMath
$0, 1, 2 \ldots$	$\mathbf{N}$	$\mathbf{N}_0$	$\mathbf{N}$	`name="N" cd="setname1"`
$1, 2, 3 \ldots$	$\mathbf{N}^+$ or $\mathbf{N}^*$	$\mathbf{N}$	?	`??`

What should replace `??`. Following our earlier policies, that different concepts (like one-valued/multi-valued arcsin) have different OpenMath, it clearly has to be a new symbol. With hindsight, the German number-theory notation might have been the best to inspire OpenMath, but we cannot change the semantics of `<OMS name="N" cd="setname1"/>`. We could introduce a new `N` in a different CD, and declare `setname1` obsolete, but that would probably be worse than the Branch Cut riots.

Hence we need another symbol. This could be in `setname1`, or in some other CD. If in `setname1`, it would need another name: if in another CD, it could also be called N, but this would probably cause more chaos. So, let us propose that we add

```
<OMS name="Nstar" cd="setname1"/>
```

to OpenMath. We then have a choice: we can define it in terms of the standard $\mathbf{N}$, as we suggested in figure 2, or we can define it in a free-standing way, by saying that it is 1 and its successors: formally

```
<OMOBJ cdbase="http://www.openmath.org/cd">
 <OMBIND>
    <OMS name="forall" cd="quant1"/>
    <OMBVAR>
       <OMV name="n"/>
    </OMBVAR>
    <OMA>
      <OMS name="implies" cd="logic1"/>
      <OMA>
        <OMS name="in" cd="set1"/>
        <OMV name="n"/>
        <OMS name="Nstar" cd="setname1"/>
      </OMA>
      <OMA>
        <OMS name="or" cd="logic1"/>
        <OMA>
          <OMS name="eq" cd="relation1"/>
          <OMV name="n"/>
          <OMS name="one" cd="alg1"/>
        </OMA>
        <OMA>
          <OMS name="in" cd="set1"/>
          <OMA>
            <OMS name="minus" cd="arith1"/>
            <OMV name="n"/>
            <OMS name="one" cd="alg1"/>
          </OMA>
          <OMS name="Nstar" cd="setname1"/>
        </OMA>
      </OMA>
    </OMA>
  </OMBIND>
</OMOBJ>
```

(it being assumed here, as in the case of the existing definition of $\mathbf{N}$, that this definition is minimal, i.e. Peano's axioms).

Provided we have *at least* the second definition (having both is not excluded), we are being as even-handed as possible: both concepts exist in OpenMath, as in the case of single-valued/multi-valued arcsin. Admittedly, the *default* rendering *might* be of $0\ldots$ as $\mathbf{N}$, and $1\ldots$ as $\mathbf{N}star$ or $\mathbf{N}^*$, but this is merely another reason for renderers to be configurable.

7 Semantics drives Notation?

So far, this paper has argued that semantics is all that matters, and that notation should follow. This is essentially the OpenMath premise (and the author's). But

life has a habit of not being so simple: take 'O'. Every student is taught that $O(f(n))$ is really a set, and that when we write "$g(n) = O(f(n))$", we really mean "$g(n) \in O(f(n))$". Almost all[13] textbooks then use '=', having apparently placated the god of Bourbaki[14]. However, actual uses of O as a set are rare: the author has never[15] seen "$O(f) \cap O(g)$", and, while a textbook might[16] write "$O(n^2) \subset O(n^3)$", this would only be for pedagogy of the O-notation. So 'O' abuses notation, but OpenMath is, or ought to be, of sterner stuff. It certainly *would* be an abuse of `<OMS name="eq" cd="relation1"/>` to use it here, as the relation it implies is none of reflexive, symmetric and transitive[17].

The set-theoretic view is the one taken by OpenMath CD[18] `asymp1`, except that only limiting behaviour at $+\infty$ is considered[19], and there is some type confusion in it: it claims to represent these as sets of functions $\mathbf{R} \to \mathbf{R}$, but in fact the expressions are assuming $\mathbf{N} \to \mathbf{R}$.

Hence it is possible to write $\lambda n.n^2 \in O(n^3)$ in OpenMath. This poses two problems for renderers:

a) how to kill the λ;
b) how to print '=' rather than '$\in$'.

The first problem is common across much of mathematics: note that $\lambda m.m^2 \in O(n^3)$ is equally valid, but one cannot say $m^2 = O(n^3)$. The second problem could be solved in several ways.

1. By resolutely using $\in$, as [17].
2. By attributing to each appropriate use of `<OMS name="in" cd="set1"/>` its print representation (at the moment there seems to be no standard way of doing this, though).
3. By fixing the rendering of `<OMS name="in" cd="set1"/>` to print it as '=', either:
   (a) for all symbols in `asymp1` (thus getting it "wrong") for symbols such as `<OMS name="softO" cd="asymp2"/>`;
   (b) or for all usages of the (STS or other) type "function in set", thus printing $\sin = \mathbf{R}^{\mathbf{R}}$.

[13] [17] is an honourable exception.
[14] "the abuses of language without which any mathematical text threatens to become pedantic and even unreadable".
[15] Not even in the one context where it *would* be useful: $\Theta(f) = O(f) \cap \Omega(f)$, which is stated in words as [10, Theorem 3.1].
[16] [10, p. 41] write $\Theta(n) \subset O(n)$.
[17] Curiously enough, the FMPs currently only state transitivity: this probably ought to be fixed.
[18] Currently `experimental`.
[19] The CD author presumably considered that the level of abstraction needed for a more general definition was unwarranted. The current author would agree, especially as the context of O is generally only implicit in the wider context of the paper.

4. (the current author's favourite) By adding a symbol[20] `<OMS name="Landauin" cd="asymp1"/>`, which would, by default, print as '=', but have the semantics of '∈'.

How is this last to be achieved? One possibility would be to say that it is the same as '∈':

```
<FMP>
<OMOBJ cdbase="http://www.openmath.org/cd">
  <OMA>
    <OMS cd = "relation1" name="eq"/>
    <OMS cd = "set1" name="in"/>
    <OMS cd = "asymp1" name="Landauin"/>
  </OMA>
</OMOBJ>
</FMP>
```

but this runs the risk of saying that any '∈' can become `Landauin`. A better way might be

```
<FMP>
<OMOBJ cdbase="http://www.openmath.org/cd">
  <OMA>
    <OMS cd = "logic1" name="implies"/>
    <OMA>
      <OMS cd = "asymp1" name="Landauin"/>
      <OMV name="A"/>
      <OMV name="B"/>
    </OMA>
    <OMA>
      <OMS cd = "set1" name="in"/>
      <OMV name="A"/>
      <OMV name="B"/>
    </OMA>
  </OMA>
</OMOBJ>
</FMP>
```

8 Conclusions

OpenMath *can* represent a variety of concepts, not just those "chosen by the designers". Alternative choices of branch cuts, single-valued/multi-valued functions, starting point for the natural numbers etc. are all supportable. Whether these are rendered in a manner appropriate to the user clearly depends on the

[20] It might be more appropriate to call it `Bachmannin`, since [4] is apparently the source of *O*. [15]

user, which means that OpenMath renderers *need* to be configurable, and at a variety of levels [19, section 4.2].

Even the Bourbaki school believe that notation exists to be abused, as well as used: OpenMath exists purely to be used, and does not exist to be abused. However, in some cases such as 'O', it may need to make slight adjustments to permit conventional notation, such as inserting symbols like `<OMS cd = "asymp1" name="Landauin"/>`, which are mathematically redundant.

8.1 Detailed suggestions

1. Add `<OMS cd = "asymp1" name="Landauin"/>`.
2. Add reflexive and symmetric properties to `<OMS cd = "relation1" name="eq"/>`.
3. Add `<OMS name="Nstar" cd="setname1"/>`, possibly to `setname1` or possibly to another CD.
4. Add a standard means of giving printing attributes (as required in 2 on page 104).

References

1. M. Abramowitz and I. Stegun. Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. *US Government Printing Office*, 1964.
2. Anonymous. Natürliche Zahlen. `http://www.mathe-online.at/mathint/lexikon/n.html`, 2006.
3. Anonymous. Wikipedia, Deutsch. `http://de.wikipedia.org`, 2007.
4. P. Bachmann. Die analytische Zahlentheorie. *Teubner*, 1894.
5. S. Buswell, O. Caprotti, D.P. Carlisle, M.C. Dewar, M. Gaëtano, and M. Kohlhase. The OpenMath Standard 2.0. `http://www.openmath.org`, 2004.
6. D.P. Carlisle. Openmath, MathML and XSLT. *ACM SIGSAM Bulletin 2*, 34:6–11, 2000.
7. World-Wide Web Consortium. Mathematical Markup Language (MathML) Version 2.0 (Second Edition). *W3C Recommendation 21 October 2003*, 2003. `http://www.w3.org/TR/MathML2/`.
8. World-Wide Web Consortium. Mathematical Markup Language (MathML) Version 3.0. *W3C Working Draft*, 2007. `http://www.w3.org/TR/2007/WD-MathML3-20070427`.
9. R.M. Corless, J.H. Davenport, D.J. Jeffrey, and S.M. Watt. According to Abramowitz and Stegun. *SIGSAM Bulletin 2*, 34:58–65, 2000.
10. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. Introduction to Algorithms, 2nd. ed.. *M.I.T. Press*, 2001.
11. J.H. Davenport. A Small OpenMath Type System. *ACM SIGSAM Bulletin 2*, 34:16–21, 2000.
12. J.H. Davenport. Nauseating Notation. `http://staff.bath.ac.uk/masjhd/Drafts/Notation.pdf`, 2007.
13. E.W. Dijkstra. Why numbering should start at zero. `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html`, 1982.
14. C. Gross. Re: Utility of OpenMath. E-mail `64191.89.49.160.232.1172849601.squirrel@webmail.uni-augsburg.de`, 2007.

15. D.E. Knuth. Big Omicron and big Omega and big Theta. *ACM SIGACT News 2*, 8:18–24, 1974.
16. A. Lazrek. Multilingual Mathematical e-Document Processing. `http://www.ima.umn.edu/2006-2007/SW12.8-9.06/activities/Lazrek-Azzeddine/MathArabIMAe.pdf`, 2006.
17. A. Levitin. Introduction to the design and analysis of algorithms. *Pearson Addison-Wesley*, 2007.
18. P. Libbrecht. E-mails. `45B8875E.7000204@activemath.org`, 2007.
19. S. Manzoor, P. Libbrecht, C. Ullrich, and E. Melis. Authoring Presentation for OPENMATH. In M. Kohlhase, editor, *Proceedings MKM 2005*, pages 33–48, 2005.
20. D.L. Mills. Network Time Protocol, Version 3. `http://rfc.net/rfc1305.html`, 1992.
21. E. Smirnova and S.M. Watt. Interfaces for Mathematical Communication. `http://www.ima.umn.edu/2006-2007/SW12.8-9.06/activities/Smirnova-Elena/SmirnovaWatt.pdf`, 2006.
22. E. Smirnova and S.M. Watt. Notation Selection in Mathematical Computing Environments. In *Proceedings Transgressive Computing 2006*, pages 339–355, 2006.
23. N. Soiffer. Accessible Mathematics. `http://www.ima.umn.edu/2006-2007/SW12.8-9.06/activities/Soiffer-Neil/index.htm`, 2006.
24. Various. LeActiveMath. `http://www.activemath.org`, 2007.
25. D.J. Wheeler. Private Communication to J.H. Davenport. *June 1982*, 1982.