Preface

This volume contains the proceedings of the second workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS 2008), held in Birmingham, UK on July 29, 2008, as a satellite workshop to the Calculemus 2008 conference, part of the Conferences on Intelligent Computer Mathematics.

Programming languages and mechanized mathematics systems intersect in many ways, and the call for papers singled out:

- *Dedicated input languages for MMS*: covers all aspects of languages intended for the user to deploy or extend the system, both algorithmic and declarative ones. Typical examples are tactic definition languages such as Ltac in Coq, mathematical proof languages as in Mizar or Isar, or specialized programming languages built into CA systems. Of particular interest are the semantics of those languages, especially when current ones are untyped.
- *Mathematical modeling languages used for programming*: covers the relation of logical descriptions vs. algorithmic content. For instance the logic of ACL2 extends a version of Lisp, that of Coq is close to Haskell, and some portions of HOL are similar to ML and Haskell, while Maple tries to do both simultaneously. Such mathematical languages offer rich specification capabilities, which are rarely available in regular programming languages. How can programming benefit from mathematical concepts, without limiting mathematics to the computational worldview?
- *Programming languages with mathematical specifications*: covers advanced "mathematical" concepts in programming languages that improve the expressive power of functional specifications, type systems, module systems etc. Programming languages with dependent types are of particular interest here, as is intentionality vs extensionality.
- *Language elements for program verification*: covers specific means built into a language to facilitate correctness proofs using MMS. For example, logical annotations within programs may be turned into verification conditions to be solved in a proof assistant eventually. How need MMS and PL to be improved to make this work conveniently and in a mathematically appealing way?

From the submissions received the Program Committee chose the following 4 contributions to be presented at the workshop. Each submission was evaluated by three referees. The symposium also included an invited talks by Dr. Conor McBride, University of Strathclyde.

We would like to thank all the people who contributed to the organization of PLMMS 2008, in particular, we are very grateful to:

- The members of the program committee: John Harrison, Hugo Herbelin, James McKinna, Ulf Norell, Bill Page, Christophe Raffalli, Josef Urban, Stephen Watt, and Freek Wiedijk.
- One additional referee, Mikolas Janota.
- Volker Sorge who helped us with local organization.

Makarius Wenzel
Jacques Carette
Workshop Chairs of PLMMS 2008

Theorem Proving for the Lazy Programmer

Conor McBride

In this talk, I shall try to identify behavioural differences between ”propositions” and ”sets” in a propositions-as-types setting and give some sort of analysis of which types might be regarded as propositional, and how we might profitably compute with their elements. More particularly, I shall examine, and (let us not beat about the bush) promote the crucial role of laziness in computing with proofs.

The talk has three parts, addressing three questions:

1. What is proof irrelevance?
   This is largely a terminological question, which I shall address briefly.
2. How do we get (away with) proof irrelevance?
   This is largely a technical question, which I shall address broadly.
3. What can we do with proof irrelevance?
   This is largely a question of design, which I shall try to address imaginatively.

We tend to distinguish propositions from other datatypes as those where we give more priority to the question of inhabitation than to choice of inhabitant. Systems such as Coq have separate sorts of Prop and Set, and they make sure that Set-valued computations can be realized as ordinary programs from which proofs are erased. However, at the level of typechecking, distinctions between proofs are still significant, rather in the manner of an uncomfortable seat. A proof irrelevant system is one which never observes distinctions between proofs. I shall arrange for this to happen by ensuring that computation in proofs is as lazy as possible.

The point, then, is to explore the impact of proof irrelevance on languages which integrate programming with proof. For one thing, we can perhaps reduce the role of terms in proofs, presenting the latter in a more declarative style. For another, we can perhaps reduce the role of proofs in terms by exploiting more automated search if nothing can depend on the choice of proof so constructed. Being a lazy programmer, I hope to enjoy the benefits of delegation.

Computing with Unknowns in Computer Algebra Systems

Yixin Cao[1] and Gabriel Dos Reis[2]

Parasol Lab
Department of Computer Science
Texas A&M University
College Station, Texas, USA.

Abstract

This paper discusses the challenges of adding symbolic computation capabilities to a computer algebra system. Indeed, a simple expressions such as $m \times n$ can be evaluated to a full value when both m and n have known values. However, if one or both of them have no definite value, then most CASs treat them as symbols and a syntactic representation of some kind is returned. Most CASs do this without much regard to types and semantics of the eventual values the symbols stand for. On the other hand, without knowing the semantics of m and n, the expression $m \times n - n \times m$ cannot be given a reliable meaning, nor can $(a + b) \times (a - b)$. This paper presents a light-weight approach to enhance a computer algebra system with the support of typed symbolic computations.

Keywords: AXIOM, Computer Algebra, Symbolic Computation, Type System.

1 Introduction

Computer Algebra and Symbolic Computation are closely related research fields; still they remain focused on different aspects of computing with mathematical objects. Most users of computer algebra systems fail to appreciate the differences, or are puzzled (if not frustrated) about why they should care about the differences. For many, the differences are rather esoteric or academic or both. Yet, the differences reveal fundamental, practical, and philosophical problems underlying the design and implementation of mechanized mathematical systems.

According to Watt [10] Computer Algebra is about *computations using arithmetic from particular algebraic constructions*, whereas Symbolic Computation is about *computation with expression trees, or "terms," representing mathematical objects.* In a nutshell, factoring a polynomial with known coefficient and known exponents, *e.g.* $X^{257} - X^{65} + 17$, falls in the realm of Computer Algebra; whereas factoring symbolic polynomials such as $x^{n^4-6n^3+11n^2-6(n+2m-3)} - 1000000^m$ — where

[1] Email: yixin@cs.tamu.edu
[2] Email: gdr@cs.tamu.edu

n and m stand for arbitrary integer values — is Symbolic Computation. Over the recent years, Stephen Watt and his colleagues have been developing data structures and algorithms for making Computer Algebra more symbolic [11,10,12,13].

The subject matter of this paper is to report on a work in progress of evolving a computer algebra system to support Symbolic Computation in a strongly typed framework. Here, we do not discuss the algorithms of symbolic mathematics *per se*. Rather, the primary focus is on the design and implementation of an interface between users and the highly sophisticated algorithms for symbolic mathematics. That is, we are concerned with *evolving* a system, primarily tuned for algebraic computations, to provide a good support for Symbolic Computation. To make the discussion more concrete, we will use the OpenAxiom [1] system as a vehicle for our ideas. However, we claim that our ideas scale to computer algebra systems built around abstract data types and strong type checking.

One might wonder why this is even an issue. Well, it appears that most general purpose computer algebra systems, such as Maple or Mathematica, are Symbolic Computation engines with support for algebraic computations built on top. What this means is that their primary data structures are abstract syntax trees. The implication is that they can handle more general expression classes at the expense of sound semantics, or efficient algorithms, or specialized data structures. Also, experience has shown that computing with ASTs can be a perilous exercise if not done with extreme caution and care [8]. In summary, in addition of sound semantics concern, special purpose computer algebra systems (*e.g.* Singular or PARI) can choose more specific and more efficient data structures in implementing efficient solutions to problems from the target domain. Consequently, an alternative design choice for a general purpose computer algebra system is to provide expressive enough programming language primitives for extending the base system with domain specific data structures. This is precisely the approach taken by the AXIOM [4] system and its descendants.

The primary contribution of this paper is to spell out the rules used by the OpenAxiom interpreter and to cast the problem in that framework. The second contribution is a proposed solution to the "typed symbolic expression" problem in terms of type inference rules and its implementation in a version of the OpenAxiom interpreter.

2 The OpenAxiom Computation Platform

OpenAxiom [1] is an evolution of the AXIOM Computer Algebra System [4]. It is an open source software for research in computational sciences with bias toward algebraic computations. OpenAxiom retains many aspects of the original AXIOM system. Indeed, it comes equipped with an interpreter and a library compiler. The interpreter is the primary user interface to the large library, containing over 1,200 data types and packages implementing mathematical data structures . The library compiler implements a strongly typed programming language (Spad) for extending the base system. The Spad programming language is roughly an extension of System F with elements of dependent types. It has a two-level type system, with values denoted by objects of *domains of computations*, and domains of computations belonging to *categories* [4]. Note that a Spad object belongs to exactly one domain of computation (generally, the domain used to create the object), while a domain can satisfy several categories simulatenously.

The idiomatic way to extend the OpenAxiom system is to identify the mathematical structures of interest, and to characterize those structures in terms of shared semantics, *e.g.* the *categories* they belong to. For example, the collection of values endowed with a monoid structure can be specified in OpenAxiom as

```
)abbrev category MYMON MyMonoid
MyMonoid(): Category == Type with
  "*": (%,%) -> %
  1  : %
```

This says that `MyMonoid` is the collection, or the category, of domains that have a binary operation named `*`, along with a distinguished constant named `1`. So far, this is just a *specification*. There can be as many implementations of this specification as deemed necessary by the application domains, and implementations are free to choose representations most adequate for meeting semantics and algorithmic complexity requirements. Furthermore, an implementation will satisfy the category `MyMonoid` by explicit assertion: This is accomplished by a domain definition. For example, we can use the additive group implemented by the domain `Integer` and the usual addition over integer values to satisfy the `MyMonoid` category:

```
)abbrev domain INTMON IntMonoid
IntMonoid(): MyMonoid with
    coerce: Integer -> %
  == add
    Rep   == Integer
    x * y == per(rep x + rep y)
    1     == per(0@Integer)
```

The domain `IntMonoid` implements (an extension of) the `MyMonoid` specification by relying on `Integer` for the value representation, as expressed by the line

```
Rep == Integer
```

That instruction in turn automatically activates a pair of morphisms

```
per: Rep -> %
rep: % -> Rep
```

inverse of each of other. From an operational semantic point of view, they are no-ops. From a static semantics point of view, `per` blesses a value of type `Rep` into a value of the current domain, whereas `rep` returns a value of the current domain to the view that it belongs to `Rep`. Please note that the operations `rep` and `per` are functions, not macros as in the Aldor programming language [9]. It is apparent that the implementation of the operation `*` offered by `IntMonoid` is given by the commuting diagram

$$\begin{tikzcd}
\texttt{IntMonoid} \times \texttt{IntMonoid} \arrow[r, "*"] \arrow[d, "\texttt{rep}\times\texttt{rep}"'] & \texttt{IntMonoid} \\
\texttt{Integer} \times \texttt{Integer} \arrow[r, "+"] & \texttt{Integer} \arrow[u, "\texttt{per}"']
\end{tikzcd}$$

Finally, the exported operation

```
coerce: Integer -> %
```

is the publically accessible function to inject values of type `Integer` into the domain `IntMonoid`.

Yet, another implementation of the `MyMonoid` category is a view of lists as a monoid structure when we restrict our attention to list concatenation and the empty list

```
)abbrev domain LISTMON ListMonoid
ListMonoid(T: Type): MyMonoid with
    coerce: List T -> %
  == add
    Rep   == List T
    x * y == per concat(rep x,rep y)$List(T)
    1     == per empty()$List(T)
```

To sum up, OpenAxiom provides strong support for Computer Algebra by offering a powerful two-level type system to write specifications (*e.g.* categories) and implementations (*e.g.* domains). The details of domains are sealed for view from the outside through the use of the private `per` and `rep` morphisms. We refer the reader to the AXIOM book [4] for an in-depth tutorial and more detailed specification of categories and domains in OpenAxiom.

A system intended for interactive mathematical computations should not be overly picky about administrative details such as type annotations on all expressions — as would a strict adherence to System F style programming. Such a system should offer sound assistant where possible, to relieve the user from distractions. Consequently, the OpenAxiom interpreter offers an elaborator of user scripts/commands written in mostly un-annotated subset of Spad into the full (larger) Spad programming language. The elaboration phase is a sophisticated type inference system with bias toward algebraic computation. This phase is what sometimes gives the illusion of OpenAxiom performing symbolic computations, and also source of confusions.

3 The Problem

Many newcomers to the OpenAxiom system in particular, and to the AXIOM family systems in general, tend to expect habits or idioms known from other symbolic computation systems (disguised as computer algebra systems) to carry over. For example, they expect the following

```
x: Integer
x + 1
```

to be given a 'sensible' interpretation – preferably that of a 'value' of type `Integer`. Yet, it does not seem obvious that `x+1` is a *value.* OpenAxiom rejects the above expression with an explanation along the line of

x ***is declared as being in*** ***Integer*** *but has not been given a value.*

The rejection (and explanation) usually only achieves to prompt further incomprehension from users. Incomprehension, probably because 'it just works with all other computer algebra systems', they say. The question then becomes: what does 'it just works' mean? The issue is much more involved than the apparent simplicity suggests. To add to the confusion, the OpenAxiom interpreter seems perfectly capable of handling an expression like

```
integrate(sin(x)^3 + tanh x,x)
```

where there is no declaration for `x` in scope, and there is no possible value for it. How is it that OpenAxiom can handle a ‘symbolic expression’ while it seems completely unable to handle expressions involving *unknowns*, *e.g.* variables with specified typed but with no known value [3]? For most newcomers, this is either mystery or plain bug. Indeed, how is it that being more precise about properties of symbols results in the outright rejection of expressions involving such unknowns? To adequately answer this questions and provide a principled and sound improvement, we need a model for the evaluation semantics of OpenAxiom.

4 The `Expression` domain

Among the many suggested solutions is the domain constructor `Expression`, part of the standard OpenAxiom distribution. It was observed that OpenAxiom is already doing symbolic computations when it evaluates symbolic expressions such as

```
integrate(sin(x)^3,x)
```

Indeed the OpenAxiom interpreter evaluates that to the symbolic expression

$$\frac{\cos^3 x - 3\cos x}{3}$$

to which it ascribes essentially the type `Expression Integer`. In general, the domain `Expression` τ provides representation for expression trees built from symbols, constants of type τ, usual ring operators, and the usual algebraic and trancendantal operators. However, it also shares most of the defects with the majority of Symbolic Computation systems in that it makes global, system wide assumptions about the meaning of symbols, *e.g.* `*` assumed to be commutative without regard to what the actual semantics of `*` is when computating with values of type τ. Furthermore, the domain constructor `Expression` comes with its own set of defects. First, it requires that its type argument describes an ordered set. However, not all interesting domain of computations — such as `Matrix Integer` — are naturally ordered. Consequently, the type `Expression Matrix Integer` is invalid. Second, user-defined functions are not automatically lifted to the symbolic expression space — they have to be defined explicitly by users. So, that severly limits the usability and scalability of the domain constructor `Expression`.

5 Semantics

The OpenAxiom system essentially consists of a library (the heart of the system) written in the Spad programming language, and an interpreter that elaborates expressions entered at the REPL into Spad expressions, evaluates them and prints the results. Extending the system boils down to defining more categories and domains as discussed in §2. Most users interact with the system through the interpreter.

— Source language syntax —

$$\begin{array}{rlll}
\textit{Library} & l & ::= & d \mid d; l \\
\textit{Definition} & d & ::= & \texttt{Rep ==}\ \tau \mid [x \mid x(s^+)]\ \texttt{:}\ \tau\ \texttt{==}\ e \\
\textit{Type} & \tau & ::= & \texttt{\%} \mid \gamma\,(\boldsymbol{\tau}) \mid \texttt{Domain} \mid \texttt{Category} \\
 & & \mid & \boldsymbol{\tau} \to \tau \mid \texttt{Join}(\boldsymbol{\tau}) \mid \tau\ \texttt{with}\ s^+ \\
\textit{Signature} & s & ::= & x\ \texttt{:}\ \tau \\
\textit{Expression} & e & ::= & c \mid x \mid e(e^+) \mid e; e \mid x\ \texttt{:=}\ e \mid i^*\ \texttt{repeat}\ e \\
 & & \mid & \texttt{if}\ e\ \texttt{then}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{add}\ d^+ \\
\textit{Iterator} & i & ::= & \texttt{for}\ x\ \texttt{in}\ e \mid \texttt{while}\ e \\
\textit{Constructor} & \gamma & ::= & \texttt{Void} \mid \texttt{Boolean} \mid \texttt{Integer} \mid \texttt{Symbol} \mid \ldots \\
\textit{Constant} & c & & \\
\textit{Identifier} & x & &
\end{array}$$

— Meta language syntax —

$$\begin{array}{rlll}
\textit{Typing Context} & \Gamma & ::= & \cdot \mid (x, \tau), \Gamma \\
\textit{Dynamic Environment} & \Delta & ::= & \cdot \mid [x \mapsto v]\Delta \\
\textit{Evaluation Context} & E & ::= & \Gamma \otimes \Delta
\end{array}$$

Figure 1: Abstract syntax of a subset of the Spad language. The notation $Z^?$ represents an optional Z, Z^+ a non-empty finite sequence of Z; the square brackets are used for grouping.

5.1 *The Library Extension Language*

The Spad programming language is an explicitly typed lambda calculus equipped with an eager semantics. The syntax of the Spad subset we consider is summarized in Figure 1. A *library* (ranged over by the metavariable l) is a sequence of definitions. A *definition* (ranged over by the metavariable d) either specifies the internal carrier set of a domain, or introduces a name and a type for a constant, a constructor, or a functional abstraction. A *type* (ranged over by the metavariable τ) is either the current domain, or a constructor instantiation, or the type of all domains, or the type of all specifications, or a function space type, or a conjunction of specification, or a specification composed of list of signatures. A *signature* is a specification of a type for an identifier.

An *expression* (ranged over by the metavariable e) is either a constant, or an identifier, or a function call, a parenthesized semicolon-separated sequence of expressions, assignment to a variable, a loop, or a branching expression, an implementation of a specification. Note that, in concrete syntax, we freely use the layout rule for parenthesized semicolon-separated sequence of expressions. A loop is controlled by a possibly empty collection of iterators. An *iterator* (ranged over by the metavariable i) is either a `for`-iterator where items are drawn one by one from a sequence of values, or a `while`-iterator that tests a predicate. Note that we don't have a notion of ‘command’ or ‘statement’ in Core Spad. Indeed, almost everything is an expression.

5.1.1 Typing Spad Libraries

$$\boxed{d \looparrowright \Gamma \vdash_{\text{decl}} \Gamma}$$

$$\textsc{Type-Const-Def}\quad \frac{\Gamma \vdash_{\text{expr}} e : \tau}{x : \tau\texttt{==}e \looparrowright \Gamma \vdash_{\text{decl}} (x, \tau), \Gamma}$$

$$\textsc{Type-Rep}\quad \frac{}{\texttt{Rep}\ \texttt{==}\,\tau \looparrowright \Gamma \vdash_{\text{decl}} (\texttt{per}, \texttt{Rep} \to \texttt{\%}), (\texttt{rep}, \texttt{\%} \to \texttt{Rep}), \Gamma}$$

$$\textsc{Type-Fun-Def}\quad \frac{(x_1, \tau_1), \cdots, (x_n, \tau_n), (x_0, \boldsymbol{\tau} \to \tau_0), \Gamma \vdash_{\text{expr}} e : \tau_0}{x_0(x_1 : \tau_1, \cdots, x_n : \tau_n)\ \texttt{==}\ e \looparrowright \Gamma \vdash_{\text{decl}} (x_0, \boldsymbol{\tau} \to \tau_0), \Gamma}$$

Figure 2: Typing rules for Core Spad — definitions.

The typing rules for Spad libraries are sketched in Figure 2, Figure 3, and Figure 4. The rules involve tree forms of judgments: typing environment update by declarations $d \looparrowright \Gamma \vdash_{\text{decl}} \Gamma$, expression typing $\Gamma \vdash_{\text{expr}} e : \tau$, and iterator validation $\Gamma \vdash_{\text{iter}} i : \text{OK}$. The rules are and presented in a Natural Semantics style [5]. Type checking uses typing contexts, (ranged over by the metavariable Γ), ordered sequences of identifier-type pairs that associate an identifier with a type.

Every typing context contains types for built-in constants given by a predefined relation

$$\mathcal{T}_{\text{cst}} \subset \mathit{Constant} \times \mathit{Type}.$$

For example, every typing context gives the type `Domain` to the constructors `Void`, `Boolean`, `Integer`, `Symbol`.

The typing of expressions is fairly conventional. An identifier x used as an expression has type τ, when τ is its declared type. We will return to this point in §5.2 and §6, source of the tension between algebraic computation and symbolic computation. Function applications requires that arguments have types that match their corresponding parameter types. Every constituent of a sequence of expressions must be well typed, and the type of whole sequence is the type of the last expression. Assignment to an already declared variable must agree in type with the declared type.

A loop with body e of type τ is well formed when its controlling iterators are well formed. A `while`-iterator is well formed if its condition is of Boolean type. Similarly a `for`-iterator is well formed if the expression where items are drawn from is a list. A one-armed `if`-expression has type `Void`, and its condition must have type `Boolean` and the `then`-branch must be well typed. A complete `if`-expression must have its condition of type `Boolean` and both its branches must have the same

$$\boxed{\Gamma \vdash_{\text{expr}} e : \tau}$$

Type-Const
$$\frac{(c, \tau) \in \mathcal{T}_{\text{cst}}}{\Gamma \vdash_{\text{expr}} c : \tau}$$

Type-Var-Expr
$$\frac{(x, \tau) \in \Gamma}{\Gamma \vdash_{\text{expr}} x : \tau}$$

Type-App
$$\frac{\Gamma \vdash_{\text{expr}} e_i : \tau_i \quad 1 \leq i \leq n \qquad \Gamma \vdash_{\text{expr}} e_0 : \tau_1 \times \cdots \tau_n \to \tau_0}{\Gamma \vdash_{\text{expr}} e_0\,(e_1, \ldots, e_n) : \tau_0}$$

Type-Seq
$$\frac{\Gamma \vdash_{\text{expr}} e_1 : \tau_1 \qquad \Gamma \vdash_{\text{expr}} e_2 : \tau_2}{\Gamma \vdash_{\text{expr}} (e_1; e_2) : \tau_2}$$

Type-Ass
$$\frac{\Gamma \vdash_{\text{expr}} e : \tau \quad (x, \tau) \in \Gamma}{\Gamma \vdash_{\text{expr}} (x{:=}e) : \tau}$$

Type-Iter
$$\frac{\Gamma \vdash_{\text{iter}} i_k : \text{OK} \quad 1 \leq k \leq n \qquad \Gamma \vdash_{\text{expr}} e : \tau}{\Gamma \vdash_{\text{expr}} i_1 \cdots i_n\, \texttt{repeat}\, e : \tau}$$

Liib-If-Partial
$$\frac{\Gamma \vdash_{\text{expr}} e_1 : \texttt{Boolean} \qquad \Gamma \vdash_{\text{expr}} e_2 : \tau}{\Gamma \vdash_{\text{expr}} (\texttt{if}\, e_1\, \texttt{then}\, e_2) : \texttt{Void}}$$

Type-If-Complete
$$\frac{\Gamma \vdash_{\text{expr}} e_1 : \texttt{Boolean} \qquad \Gamma \vdash_{\text{expr}} e_2 : \tau \qquad \Gamma \vdash_{\text{expr}} e_3 : \tau}{\Gamma \vdash_{\text{expr}} (\texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3) : \tau}$$

Figure 3: Typing rules for Core Spad — expressions.

$$\boxed{\Gamma \vdash_{\text{iter}} i : \text{OK}}$$

Type-For
$$\frac{\Gamma \vdash_{\text{expr}} e : \texttt{List}\,(\tau)}{(x, \tau), \Gamma \vdash_{\text{iter}} \texttt{for}\, x\, \texttt{in}\, e : \text{OK}}$$

Type-While
$$\frac{\Gamma \vdash_{\text{expr}} e : \texttt{Boolean}}{\Gamma \vdash_{\text{iter}} \texttt{while}\, e : \text{OK}}$$

Figure 4: Typing rules for Core Spad — iterators.

type.

5.1.2 *Operational Semantics of Spad Libraries*

A big step operational semantics of a subset of Spad libraries was given in [7]. We recall here some key points, even though not all of the intricate details of Spad are essential to appreciate the rules relevant to the problem at hand, and our proposed solution. We have to admit upfront that we do not offer a soundness proof at the moment. We would like to stress that we view this formalism as a necessary step to rationale evaluation of the "typed symbolic computation" puzzle and its proposed solutions.

Eexpressions in Spad evaluate to values

$$\textit{Values} \quad v \quad ::= \quad \mathrm{c} \mid (v, \cdots, v) \mid \langle v, \cdots, v \rangle$$

which are constants, or list of values, or array of values. The constant values include the truth values, integer values, identifiers, and closures.

5.2 *The interpreter*

Eval-Bound-Id

$$\frac{(x, \tau) \in \Gamma \quad \Delta(x) = v}{\{\Gamma \otimes \Delta, x\} \longrightarrow \{\Gamma \otimes \Delta, \langle \tau, v \rangle\}}$$

Eval-Sym-Id

$$\frac{x \notin \operatorname{dom} \Gamma}{\{\Gamma \otimes \Delta, x\} \longrightarrow \{\Gamma \otimes \Delta, \langle \texttt{Symbol}, x \rangle\}}$$

Eval-Lib-Call

$$\frac{\begin{array}{c}\{\Gamma \otimes \Delta, e_0\} \longrightarrow \{\Gamma_0 \otimes \Delta_0, \langle (\tau_1, \cdots, \tau_n) \to \tau, v_0 \rangle\} \\ \{\Gamma_{i-1} \otimes \Delta_{i-1}, e_i\} \longrightarrow \{\Gamma_i \otimes \Delta_i, \langle \tau_i, v_i \rangle\} \\ \{\Gamma_n \otimes \Delta_n, \operatorname{apply}(v_0, v_1, \cdots, v_n)\} \longrightarrow \{\Gamma_{n+1} \otimes \Delta_{n+1}, \langle \tau, v \rangle\}\end{array}}{\{\Gamma \otimes \Delta, e_0(e_1, \cdots, e_n)\} \longrightarrow \{\Gamma_{n+1} \otimes \Delta_{n+1}, \langle \tau, v \rangle\}}$$

Figure 5: Evaluation rules for Core Spad — expressions.

The syntax of the language accepted by the interpreter is the same as that of Spad, except that

- type annotations are no longer required in definitions;
- identifiers may be declared without being assigned to;
- identifiers may be used in expressions without declarations or definitions.

To accommodate for these possibilities, the interpreter combines typing contexts Γ with dynamic environments Δ into evaluation contexts $E = \Gamma \otimes \Delta$. This is also justified by the fact that typing is intertwined with evaluation. Evaluation of expressions yield objects

$$\textit{Object} \quad o \quad ::= \quad \langle \tau, v \rangle$$

which are pairs of type and value. For example, the expression `(5::IntMonoid) * (2::IntMonoid)` evaluates to the object $\langle \texttt{IntMonoid}, 7\rangle$. Indeed, the morphism `per` transforms the object $\langle \texttt{Rep}, v\rangle$ to the object $\langle \texttt{\%}, v\rangle$, and the `rep` morphism does the inverse. So much for the object model. We model the interpreter evaluation by reduction rules of the form

$$\{\Gamma \otimes \Delta, e\} \longrightarrow \{\Gamma' \otimes \Delta', o\}.$$

Function evaluation is a standard, applicative order, eager semantics. The most interesting aspect of the evaluation semantics of OpenAxiom, and the one that seems to cause troubles for supporting Symbolic Computation is that of evaluation of a variables.

5.3 Declared and Bound Variables

First, let's consider the non problemantic case of a variable that is declared and bound Eval-Bound-Id. If variable x is declared in the current context as having type τ and, if it has a value v known in the dynamic environment, then it evaluates to the object $\langle \tau, v\rangle$, of type τ. Please, note that the dynamic environment is checked only if the variable is declared in the typing environment. With this typing rule the expression

```
a := 3
-a
```

unsurprisingly evaluates to the object $\langle \texttt{Integer}, -3\rangle$. However it obviously does not handle an expression such as `sin x`, which drags us in the territory of Symbolic Computation. To handle such case, the OpenAxiom interpreter has another typing rule: Eval-Sym-Id. This rule is responsible for the interpreter synthesizing an object with value x, whenever the symbol x is referenced but has no declaration in scope. This rule and the fact that the domain `Expression Integer` has

- an implicit conversion from the `Symbol` domain, and
- a modemap for `sin` that constructs an object of type `Expression Integer` from another object of such type.

The rules Eval-Bound-Id and Eval-Sym-Id are the only ones that OpenAxiom uses for determining the value of an identifier. They seem to suffice for OpenAxiom to give the illusion of doing Symbolic Computation. However, that illusion breaks down very quickly when it comes to computations with unknowns. The next sections present our solution of augmenting OpenAxiom's type system to support such form of symbolic mathematics.

6 Typed Symbolic Computation

6.1 New Inference Rules

To account for unknowns, which are declared variables without assigned values, we introduce a new unary domain constructor named `Symbolic`, and new typing rules as follows. First, the domain `Symbolic T` is intended to model an abstract syntax tree constructed from values and uninterpreted symbols. The intended semantics is that when the free variables in an expression of type `Symbolic` τ are substituted for with values of type τ, then the resulting expression is well typed, and has type τ.

Second, the inference rules used by the interpreter are as follows:

$$\textsc{Eval-Unbound-Id}\quad \frac{\Gamma \vdash_{\text{decl}} x : \tau \quad x \notin \operatorname{dom} \Delta}{\{\Gamma \otimes \Delta, x\} \longrightarrow \langle \texttt{Symbolic}\,\tau, x\rangle}$$

If a variable x has a declared type τ in the current context but has no value in the current environment, then the interpreter synthesizes an object of type `Symbolic` τ, with value the identifier x. In another words, a declared but unbound variable *evaluates to a value synthesized by the interpreter.* Note that it is not the variable x that is of type `Symbolic` τ. Rather, it is the value of the *expression* 'x'. The distinction is very important. In particular, in the library a variable always evaluates to a value of its declared type.

To make the new inference rule useful, we need to address the question of what happens when an expression of type `Symbolic` τ is used as an argument to a function call. We basically have two options:

(i) either the function expects a value that type in the corresponding parameter position, in which case the symbolic object is passed using the existing library checking rules, and the call is evaluated accordint to rule Eval-Lib-Call;

(ii) or the function expects a value of type τ. A design choice may be to reject the call, since there is no way the function call can possibly be evaluated soundly. Doing so, however, would render the feature almost useless as one would have to duplicate all function definitions with which one could possibly compute with symbolic expressions.

Another option is realize that the call would have been otherwise legal was the variable actually bound to some value. In that case, we build a kind of suspension $\langle v_0, \cdots, v_n\rangle$ that remembers the modemap that would have been selected, and the arguments list. This suspension is encapsulated by the implementation of the `Symbolic` domain constructor.

$$\textsc{Eval-Sym-Call}\quad \frac{\begin{array}{c}\{\Gamma \otimes \Delta, e_0\} \longrightarrow \{\Gamma_0 \otimes \Delta_0, \langle (\tau_1, \cdots, \tau_n) \to \tau, v_0\rangle\} \\ \{\Gamma_{i-1} \otimes \Delta_{i-1}, e_i\} \longrightarrow \{\Gamma_i \otimes \Delta_i, \langle \tau_i, v_i\rangle\} \\ \{\Gamma_{k-1} \otimes \Delta_{k-1}, e_k\} \longrightarrow \{\Gamma_k \otimes \Delta_k, \langle \texttt{Symbolic}, \tau, v_k\rangle\}\end{array}}{\{\Gamma \otimes \Delta, e_0\,(e_1, \cdots, e_n)\} \longrightarrow \{\Gamma_n \otimes \Delta_n, \langle \texttt{Symbolic}\,\tau, \langle v_0, \cdots, v_n\rangle\rangle\}}$$

This rule effectively lifts functions defined on 'algebraic data structures' to 'symbolic expression space'. Note however that the lifts do not actually compute anything: They cannot, since the actual values of the identifiers are missing. Rather, they build a typed abstract syntax trees that can be safely evaluated later, when values are known for the unbound variables.

With these two rules, we have been able to add preliminary support for computations with 'unknown quantities' to OpenAxiom.

6.2 Results

Below are some actual session runs based on the modified OpenAxiom to show our results. With our modifications, the polynomial computations are almost the same (some unnecessary output details are omitted here):

```
    (1) -> a:Integer
    (2) -> b:Integer
    (3) -> c:=a*3-b*2
       (3)  - 2b + 3a
                                                    Type: Symbolic Integer
    (4) -> p:=x*2-7
       (4)  2x - 7
                                                  Type: Polynomial Integer
    (5) -> pc:=p c
       (5)  - 4b + 6a - 7
                                                    Type: Symbolic Integer
    (6) -> f(x)==x^3-x^2+1
    (7) -> fb := f b
       (7)  b^3  - b^2  + 1
                                                    Type: Symbolic Integer
    (8) -> a:=1
    (9) -> b:=-3
    (10) -> pc
       (10)  11
                                                     Type: PositiveInteger
    (11) -> c
       (11)  9
                                                     Type: PositiveInteger
    (12) -> fb
       (12)  - 35
                                                             Type: Integer
```

The only obvious benefit here is we define symbolic variable the same as the normal variable. However, when we move to less trivial domains, we can observe more returns:

```
    (1) -> m:Matrix Integer
    (2) -> n:Matrix Integer
    (3) -> a:Integer
    (4) -> b:Integer
    (5) -> (a+b)* (a-b)
       (5)  - b^2  + a^2
                                                    Type: Symbolic Integer
    (6) -> m*n-n*m
       (6)  m n - n m
                                             Type: Symbolic Matrix Integer
    (7) -> (m+n)*(m-n)
       (7)  (m + n) (m - n)
                                             Type: Symbolic Matrix Integer
```

Notice that our solution does not suffer from the common defect of interpreting the symbol `*` as designating a commutative multiplication. Of course, we can construct more complicated examples involving symbolic variables are coefficients of matrices. We did not provide another function `matrix` or `vector` in our domain, and this proposed solution can handle `Matrix`, `Vector`, `Complex`, etc. automatically:

```
(8) -> (a1,a2,a3,a4):Integer
(9) -> (b1,b2,b3,b4):Integer
(10) -> m1:=matrix [[a1,a2],[a3,a4]]
         +a1  a2+
   (10)  |      |
        +a3  a4+
                                                     Type: Matrix Symbolic Integer
(11) -> m2:=matrix [[b1,b2],[0,1]]
        +b1  b2+
   (11)  |      |
        +0   1 +
                                                     Type: Matrix Symbolic Integer
(12) -> m1*m2
        +a1 b1  a1 b2 + a2+
   (12)  |                 |
        +a3 b1  a3 b2 + a4+
                                                     Type: Matrix Symbolic Integer
(13) -> v:=vector [a1,b3]
   (13)  [a1,b3]
                                                     Type: Vector Symbolic Integer
(14) -> v*m1
                          2
   (14)  [a3 b3 + a1 ,a4 b3 + a1 a2]
                                                     Type: Vector Symbolic Integer
```

7 Related Work

The notion of computations with unknowns appears to be a long vexing issue for the AXIOM family computer algebra systems. Davenport and Faure [3] tackled this problem early 1990 and proposed a three-level hierarchy of *unknown domains*: domain `BasicUnknowns` for the variable symbols, `PureUnknownInteger` domain for the expression with `BasicUnknowns`, and the `ConditionalUnknownInteger` domain for expression with several possible values depending the some Boolean conditions. Our approach differs from theirs in that we have opted for an almost transparent solution for the user. Furthermore, we believe that listing explicitly the unbound variables in the type of expressions leads to a non-trivial complexity in terms of use. In the recent years, Stephen Watt [11,10,12,13] and his colleagues have been developing algorithms for efficient symbolic mathematics. We see our work as complementary to theirs, in the sense that we are focusing on the type rules adequate for sound interactive symbolic mathematics sessions. Sexton and Sorge [6] recently proposed algorithms for abstract matrices, *e.g.* matrices with symbolic coefficients.

After initial submission of this work, Bill Page has proposed uses of the `Union` domains as carrier set for symbolic expressions, so that the base type system rules can be applied with few extensions [2]. However, in that scheme, every function (either system supplied or user-defined) needs to be duplicated in that domain to be supported, *e.g.*, without the duplication of `matrix` function, the `matrix [[a,b]]` on two `Symbolic Integer` will not work. Another important direction is to complete the formalization so that a soundness theorem can be proved.

8 Conclusion and Future Work

In this work, we gave a transparent and easy-to-use paradigm for typed symbolic computations in OpenAxiom, in a formal system so that we can assess design choices and competing suggestions. We believe it is very crucial that such an important topic be discussed in a formal framework so that we can gain better understanding of the issues and different suggestions put forward. A pressing future direction is to get the algorithms developed by Stephen Watt in our system.

Acknowledgement

We would like to thank the AXIOM family systems users who brought this issue to our attention, in particular, Bill Page, Tim Daly, Ralf Hemmecke and Martin Rubey and other participants in the discussions on the AXIOM mailing lists we have forgotten to mention.
We thank the anonymous referees for their valuable remarks and suggestions.

References

[1] OpenAxiom: The Open Scientific Computation Platform. http://www.open-axiom.org/.

[2] Simple implementation of symbolic computation in openaxiom. http://axiom-wiki.newsynthesis.org/SandBoxSymbolic.

[3] James Davenport and Christèle Faure. The "unknown" in computer algebra. In *Programmirovanie*, pages 4–10, Jan 1994.

[4] Richard D. Jenks and Robert S. Sutor. *AXIOM: The Scientific Computation System*. Springer, 1992.

[5] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences*, pages 22–39, London, UK, 1987. Springer-Verlag.

[6] Alan Sexton and Volker Sorge. Abstract Matrices in Symbolic Computation. In *ISSAC '06: Proceedings of the 2006 international symposium on Symbolic and algebraic computation*, pages 318–325, New York, NY, USA, 2006. ACM.

[7] Jacob Smith, Gabriel Dos Reis, and Jaakko Järvi. Algorithmic Differentiation in Axiom. In *ISSAC '07: Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, pages 347–354. ACM New York, USA, July 2007.

[8] David R Stoutemyer. Crimes and Misdemeanors in the Computer Algebra Trade. *Notices of the American Mathematical Society*, 37:778–785, 1991.

[9] Stephen Watt. The aldor programming language. http://www.aldor.org/.

[10] Stephen M. Watt. Making computer algebra more symbolic. In *Transgressive Computing 2006: A conference in honor of Jean Della Dora*, pages 43–49, 2006.

[11] Stephen M. Watt. Symbolic computation versus computer algebra. In *Proc. 2006 Conference on the Applications of Computer Algebra*, pages 43–49, Varna, Bulgaria., June 26-29 2006.

[12] Stephen M. Watt. What happened to languages for symbolic mathematical computation? In Jacques Carette and Freek Wiedijk, editors, *PLMMS*, pages 81–90, 2007.

[13] Stephen M. Watt. Functional Decomposition of Symbolic Polynomials. In *Proc. Computer Algebra Systems and Their Applications, (CASA 2008)*, Perugia, Italy, June 30-July 3 2008. to appear.

CRStL: A Declarative Language for the Encoding of Proof Techniques

Dominik Dietrich[1] Ewaryst Schulz[2]

FR 6.2 Informatik
Saarland University
Saarbrücken
Germany

Abstract

We propose the language **CRStL** to formulate mathematical reasoning techniques as proof strategies in the context of the proof assistant ΩMEGA. The language is arranged in two levels, a query language to access mathematical knowledge maintained in development graphs, and a strategy language to annotate the results of these queries with further control information. The two-leveled structure of the language allows the specification of proof techniques in a very declarative way. We give examples to illustrate the use and semantics of **CRStL**.

Keywords: tactic language, query language, proof search, proof planning

1 Introduction

Unlike widely used computer-algebra systems, mathematical assistance systems have not yet achieved considerable recognition and relevance in mathematical practice. One significant shortcoming of the current systems is that they are not fully integrated into or accessible from standard tools that are already routinely employed in practice, like, for instance, standard mathematical text-editors. Integrating formal modeling and reasoning with tools that are routinely employed in specific areas is the key step in promoting the use of formal logic based techniques.

Therefore, in order to foster the use of proof assistance systems, we integrated the theorem prover ΩMEGA [11] into the scientific text-editor $\text{T}_\text{E}\text{X}_{\text{MACS}}$ [13]. The goal is to assist the author inside the editor while preparing a $\text{T}_\text{E}\text{X}_{\text{MACS}}$ document in a publishable format. The vision underlying this research is to enable a document-centric approach to formalizing and verifying mathematics and software, that is, we are aiming at human readable, human writable, and machine checkable documents. In our current implementation theories can be formalized within the text editor and proofs can interactively be constructed by

[1] Email: dietrich@ags.uni-sb.de
[2] Email: schulz@ags.uni-sb.de

the application of basic proof operators, proof strategies, which are similar to tactics, or the introduction of proof sketches. A significant shortcoming in the current version of our system is that proof strategies cannot be specified within the document. Rather, they need to be programmed in the underlying programming language of the proof assistant, which is in the case of ΩMEGA the programming language Lisp. It is clear that for complex and efficient proof strategies a complete programming language is necessary. However, small parts of proofs can often be automated by special purpose proof strategies, which could in principle simply be specified by the user. However, requiring the use of the underlying programming language often prevents the user to take this option because he is unfamiliar with the language. Even for experienced users it is often too time consuming to design a special purpose proof strategy in the underlying programming language.

Consider for example the following simple theorem about binary relations:

$$(R \cup S) \circ T = (T^{-1} \circ R^{-1})^{-1} \cup (T^{-1} \circ S^{-1})^{-1} \tag{1}$$

Suppose further that we have already proved a theorem "Inverse composition"

$$\forall R, S.(R \circ S)^{-1} = S^{-1} \circ R^{-1}, \tag{2}$$

a theorem "Distributivity of $\circ$ over $\cup$":

$$\forall R, S, T.(R \cup S) \circ T = (R \circ T) \cup (S \circ T) \tag{3}$$

and a theorem "Doubleinverse identity":

$$\forall R.R = (R^{-1})^{-1} \tag{4}$$

A possible approach to proof (1) would be to expand all the definitions involved in (1) and then to try to prove the resulting formulas using propositional logic rules, which is already implemented in ΩMEGA as a generic strategy. However, there is a more elegant way. It is easy to see that the previous proved theorems (2), (3) and (4) can directly be used to prove the problem within a few steps. Writing a proof strategy for this specific case in the underlying programming language is laborious; a more lightweight interface would be beneficial.

In this paper we present the declarative language **CRStL** [3] for the specification of proof strategies in ΩMEGA. Similar to $\mathcal{L}_{tac}$ [8] the language is intended to bridge the gap between the predefined proof operators and the programming language of the proof assistant. Our language differs from standard languages for tactics with respect to the following aspects: (1) We consider the theory environment in which the proof construction takes place as a data base and provide an explicit query mechanism to retrieve knowledge items from this environment, in particular proof operators. The retrieved knowledge items can be annotated with additional control information such as matching conditions. Building proof strategies on top of this query mechanism allows the specification of the proof operators to be used within a proof strategy by their properties. This way proof strategies automatically adapt to new contexts when the theory changes. (2) We provide language constructs to select subgoals for subsequent strategy executions by their properties. Thus a proof strategy does not

[3] **C**ontrol**R**ule**St**rategy **L**anguge, pronounce "crystal"

depend on the order in which the subgoals were introduced. (3) In addition to procedural descriptions of proof operators we support also the specification of goal descriptions which will then be used as solution conditions in the corresponding strategy.

The paper is organized as follows: In Sec. 2 we set the context of this work by describing proof construction and knowledge management in the ΩMEGA system. Sec.3 gives an overview of the strategy language **CRStL** which allows the specification of proof strategies in ΩMEGA. We conclude the paper with a discussion in Sec. 4.

2 ΩMEGA's Reasoning Framework

ΩMEGA provides a generic framework for proof construction which consists of (1) the development graph to store the mathematical knowledge structured in theories such as definitions, and axioms (2) the TASKLAYER to maintain a proof attempt and to perform basic proof steps (3) a proof planner to perform a search based on the notion of proof strategies and control rules. In this section we give an overview of these components.

2.1 Knowledge Management in the Development Graph

The knowledge of the ΩMEGA system is organized in axiomatic theories that are built on top of each other by importing knowledge from lower theories via theory morphisms. This organization is based on the notion of *development graphs* (see [4] for details).

Each theory in MAYA's development graph contains standard information like the signature of its mathematical concepts and corresponding axioms, lemmas and theorems. In addition to these notions, the development graph allows the specification of other kinds of knowledge, which are not necessarily affecting the semantics of a theory but which, for instance, provide valuable information to proof procedures. An example is ordering information for the function symbols in the signature of a theory, which can be exploited by simplification procedures. Knowledge can either manually be added to a knowledge kind, or automatically be classified by classification functions. For example, one either specifies a formula to be a definition, or relies on predefined heuristics for definition detection.

Each knowledge item is attached to a specific theory and is visible in all theories that link to this theory. The links can use morphisms to transform the structures from the source theory, therefore the knowledge items are transformed also along these morphisms . For instance, a morphism that renames a function f into a function g transforms an ordering information that $f > h$ (for some function h of the signature) into the ordering information $g > h$. The default behavior for knowledge transformations is simply to transform all terms which occur inside the knowledge item.

2.2 The TASKLAYER

The central component for proof construction in ΩMEGA is the TASKLAYER. It is based on the CORE-calculus [2] that supports proof development at the *assertion level* [10], where proof steps are justified directly by definitions, axioms, theorems or hypotheses (collectively called *assertions*) omitting basic logic inference applications for changing the structure of the formula to match exactly the expected form of the assertion.

Tasks. At the TASKLAYER, the main entity is a task, a multi-conclusion sequent $F_1,\dots,F_j \vdash G_1,\dots,G_k$, expressing a subgoal to be proved. Initially there is only a single task consisting of the conjecture to be proved. A proof attempt is represented by an *agenda*, consisting of a set of tasks, a global substitution which instantiates meta-variables, and contextual information. Tasks are reduced to subtasks by applying proof operators, called *inferences*. If a task has been reduced to an empty set of subtasks, it is called *closed*, otherwise it is called *open*. Tasks and subformulas can be assigned named foci of attention that are maintained within the actual proof.

Inferences. The basic operators for proof construction are so-called *inferences*. Intuitively, an *inference* is a proof step with multiple premises and conclusions augmented by (1) a possibly empty set of hypotheses for each premise, (2) a set of *application conditions* that must be fulfilled upon inference application, (3) a set of *completion functions* that compute the values of premises and conclusions from values of other premises and conclusions. Usually, inferences encode the operational behavior of domain specific assertions. However, they can also encode proof planning methods or calls to external special systems such as a computer-algebra system, an automated deduction system or a numerical calculation package (see [9,3] for more details).

Rather than working with a fixed set of predefined inferences, the set of inferences is continuously extended whenever a new theorem has been proved. The technique to obtain such inferences automatically from assertions follows the introduction and elimination rules of a natural deduction calculus (see [3] for details). For the purpose of this paper it is important to note that one formula can result in several inferences. Consider for instance the domain of set theory and the definition of $\subseteq$:

$$\forall U, V. U \subseteq V \Leftrightarrow (\forall x. x \in U \Rightarrow x \in V) \tag{5}$$

That assertion gives rise to two inferences:

$$\cfrac{\begin{array}{c}[x \in U]\\ \vdots \\ p : x \in V\end{array}}{c : U \subseteq V}\,\mathit{Def}\text{-}\subseteq \qquad \textbf{Appl. Cond.: } \mathit{x\ new\ for\ U\ and\ V} \tag{6}$$

$$\cfrac{p_1 : U \subseteq V \qquad p_2 : x \in U}{c : x \in V}\,\mathit{Def}\text{-}\subseteq \qquad \textbf{Appl. Cond.: } - \tag{7}$$

Reading bottom up, the first inference can be paraphrased as follows: In order to show $U \subseteq V$, we can assume $x \in U$ and have to show $x \in V$ for a fresh variable x. U and V are meta-variables that need to be instantiated during the matching of the inference. However, we can also read (and apply) the inference top down: If we know that $x \in V$ under the assumption $x \in U$ for arbitrary x, then we can conclude $U \subseteq V$.

2.3 Proof Automation

ΩMEGA provides a proof planning framework to encode and apply common patterns of reasoning, called *strategies*. Intuitively a strategy performs a heuristically guided search using a dynamic set of inferences (as well as other strategies), and control rules which determine the planner's behavior at choice points until a specified termination condition succeeds or the search space has been completely traversed. A strategy application results in a nonempty list of solutions encoded in form of agendas or fails. Technically a strategy is a function from an agenda to a list of agendas.

In a nutshell, a strategy execution corresponds to the following loop, which is executed until a specified termination condition is satisfied or the search space is completely traversed: (1) Selection of an agenda A (2) Selection of an open task T from A (3) Selection of the proof operators to be applied to T (4) Instantiation of the proof operators with respect to T under some additionally specified constraints, resulting in a list of so-called *partial argument instantiations* (pais) (5) Selection and application of a subset of the pais, resulting in new agendas. Those new agendas satisfying a solution condition are collected and not further modified. Backtracking occurs if no pais can be produced in step (4). However, there is also the possibility to explicitly backtrack, e.g., if a certain depth has been reached.

Up to now, the only possibility to formulate strategies in ΩMEGA is to use a language based on lisp s-expressions, called POST. This language doesn't provide (1) an access to the structured knowledge stored in the development graph other than symbols, axioms or theorems and (2) a pattern matching facility for term matching and filtering.

3 The Language and its Components

This section introduces the main components of the language **CRStL**. Our goal is to provide a language which permits the user to implement new proof techniques, and thus to extend the systems automation capability, without requiring the knowledge of the systems underlying programming language and its automation API. We are not aiming at a complete programming language, rather at a small, extensible, intuitive, but restricted language for automating small parts of proofs. We want to cover the full spectrum from declarative proof strategies, i.e., specifying what to achieve and what knowledge to be used, to full procedural proof strategies, i.e., specifying what proof operators to apply in which order. Our main requirements are:

- Structured access to the mathematical knowledge stored in the development graph to specify a subset of the proof operators to be used within a proof strategy, similar to SQL or XPath.
- Dynamic binding of variables to parts of the queried structures, especially terms, for later use in constraints or ordering statements.
- Abstraction over the conversion of knowledge to different formats, in particular the conversion of axioms and theorems to inferences.
- Easy access to terms and tasks to express their structural properties, e.g., for specifying constraints to restrict the search space, as well as solution conditions for strategies.
- Introduction of explicit backtracking conditions.
- Restriction of the instantiation possibilities for proof operators.
- Specification of orderings at choice points.
- Support of new user defined predicates and functions.

The language is arranged in two levels, a query language to access the mathematical knowledge, and a strategy language which makes extensive use of these queries and annotates the result of a query with further control information to build a proof strategy. Table 1 summarizes the language **CRStL** in a BNF like notation.

```
<select>         ::= select <selector> from <source> <wherecond>?
                  |  <select> union <select>

<selector>       ::= *
                  |  term
                  |  (name,)* name
<source>         ::= theoryname
                  |  theoryname.knowledge

<infexpr>        ::= use <select> <infcond>? <wherecond>
                  |  <infexpr> union <infexpr>
                  |  <infexpr> intersection <infexpr>
                  |  <infexpr> difference <infexpr>
<infcond>        ::= as <infdirection>
<wherecond>      ::= where <function_call>
<infdirection>   ::= forward | backward | close

<listexpr>       ::= <infexpr> | <stratexpr>+

<stratexpr>      ::= first <listexpr>
                  |  solve <stratexpr>
                  |  repeat <stratexpr> <untilcond>?
                  |  ( <stratexpr> )
                  |  try <stratexpr>
                  |  <stratexpr> then <stratexpr>
                  |  name
                  |  <stratexpr> backtrack-if <cond>
                  |  cases <case>+ end;
                  |  <stratexp> thenselect cases <case>+ end;

<case>           ::= <cond> -> <stratexpr>

<untilcond>      ::= until <cond>

<cond>           ::= <matcher>
                  |  <function_call>

<defstrat>       ::= define-strategy name <parameter>? <stratexpr> end;

<parameter>      ::= parameter (name,)* name;
<matcher>        ::= <matchhead> <matchcond>?
<matchhead>      ::= <sequent> | var
<matchcond>      ::= where  <function_call>
<sequent>        ::= (<termpattern>,)* <termpattern> |- <termpattern>
<namedterm>      ::= <term> | name:<term> | *
<termpattern>    ::= <namedterm> | [<namedterm>] <termqualifier>?
<termqualifier>  ::= + | -
```

Table 1
Syntax of **CRStL**

To get a first feeling for the language consider the following proof strategy "Special Purpose" for the motivating example, shown in Listing 1. "Special Purpose" essentially consists of four parts: a specification of a backtrack event, a repeat loop, a use expression and a select expression. The easiest way to understand the strategy is to read the defining expression from the inside to the outside. The select expression specifies two theorems to be selected from the theorems of the current theory. The use expression performs the conversion of the theorem names to inferences. The result of the query is a set of two inferences, representing the four rewrite rules

$$(R \cup S) \circ T \rightarrow (R \circ T) \cup (S \circ T) \tag{8}$$

$$(R \circ T) \cup (S \circ T) \rightarrow (R \cup S) \circ T \tag{9}$$

$$(R \circ S)^{-1} \rightarrow S^{-1} \circ R^{-1} \tag{10}$$

$$S^{-1} \circ R^{-1} \rightarrow (R \circ S)^{-1} \tag{11}$$

The **solve** construct specifies that the goal of the proof strategy is to solve the task to which it is applied to. This already includes backtracking if none of the rewrite rules is applicable in a situation. Finally, **backtrack-if** enriches the backtracking behavior of the proof strategy such that a backtrack event is invoked whenever the function call `(> stratdepth 3)` evaluates to true. In this expression `stratdepth` is a predefined variable which is bound at runtime to the depth of the search tree by the proof strategy. The **define-strategy** binds the proof strategy to the name "Special Purpose". However, it is also possible to directly invoke a strategy expression.

```
define-strategy "Special Purpose"
  solve
   use
    select "Distributivity of ∘ over ∪",
           "Inverse composition"
    from current.theorems
  backtrack-if
   (> stratdepth 3)
end;
```

CRStL Listing 1: Special purpose strategy for the example

The previous example showed how we can realize a simple depth limited search strategy within our framework. In general the user has to find a trade-off between the restriction of the search space and the simplicity of the search strategy by further control constructs. As the search space is very small in our example, we decided in favor of a very simple proof strategy. Note that in our language function calls, such as the depth limiter from the previous example, build the interface to the underlying programming language and can comprise manipulations of arbitrary lisp objects. Moreover, function calls have access to a number of predefined variables, such as the execution time, the agenda, the theory, or the search depth. We now explain the language constructs in more detail.

3.1 The select *Statement*

The select statement is used to select knowledge items from a specified theory. It consists of three parts, a **selector** part, a **from** part, and a **where** part.

all	all theories reachable from the current context
current	only the local knowledge of the current theory
base	only the underlying logic
top(n)	only the theories reachable from the current theory in n steps
“name”	only the local knowledge of the theory given by name
axioms	the axioms
theorems	already proved theorems
formulas	axioms and proved theorems
definitions	axioms which were classified as definitions
inferences	inferences, user defined and derived from axioms
strategies	user defined strategies
knowledge	stands for any knowledge

Table 2
Available source keywords

The From Part. The **from** part specifies the knowledge source and the theory of the development graph from which the knowledge is retrieved. The theory and the corresponding knowledge source can be accessed by their names. Moreover, we support a number of predefined keywords, e.g., to access the current theory, or the base theory. An overview of the available keywords is shown in Table 2.

From a global perspective one can divide the knowledge into two parts: *direct knowledge*, and *indirect knowledge*. *Direct* knowledge is knowledge which has the form of a proof operator, i.e., an inference or a proof strategy, or can be converted to such. For that purpose knowledge transformation functions need to be specified. For example the function which transforms a name to an axiom simply returns the first axiom which has the specified name. The transformation function from formulas to inferences determines exactly those inferences which were synthesized from the specified formula. An example of a direct knowledge item is an axiom. In this case the conversion of the formula results in all inferences which were obtained from this formula. Thus in case of the definition of subset (5) these are the inferences shown in (6), and (7).

Indirect knowledge is knowledge which cannot be converted to a proof operator, but which can be used at choice points. An example for indirect knowledge is a symbol ordering. It cannot be converted to a proof operator, however it can indirectly be used to restrict the applicability of the proof operators.

Note that new knowledge kinds can easily be added to the development graph and are directly available in the query language.

The Selector Part. The **selector** part works on the knowledge kind specified in the

from part and can be used to further narrow down the set of knowledge items returned by the query. In the simplest case the **selector** part consists of a single $*$. In this case all knowledge items are returned. The **selector** part may also consist of a set of names, in which case only those knowledge items are returned for which there is a name in the specified list of names. Finally there is the possibility to specify a term pattern. In this case only those knowledge items which correspond to a formula that matches the pattern are returned. The term pattern shares the variables with the proof context in which the query is evaluated. Free variables are interpreted as meta-variables to be instantiated by the query. These instantiated variables are passed to the **where** part and can be used there for the specification of additional constraints. We found out that it is convenient for the matching to remove all leading quantifier of formulas corresponding to knowledge items.

The Where Part. The **where** part can be used to specify additional constraints the knowledge items of the query have to satisfy. A variable binding which stems from a pattern matching in the **selector** part is available for evaluation of the expression in the **where** part. Listing 2 shows an example of a select expression consisting of a **selector** part, a **from** part, and a **where** part. The query returns those axioms from the current theory which are equations (after removing the leading quantifiers) $lhs = rhs$ and binds the lefthand side of the equation to the variable *lhs*, and the righthand side of the equation to the variable *rhs*. In the **where** part, it is checked whether *lhs* is greater as *rhs* using the LPO with the symbol ordering stored in the development graph as measure for the terms.

```
select lhs=rhs from current.axioms
  where (greaterlpo lhs rhs
    (select * from current.ordering))
```

CRStL Listing 2: Example of a select expression which binds variables *lhs* and *rhs* for the specification of additional constraints

3.2 *The* use *Statement*

The use statement can be invoked directly after a select statement and performs two tasks: (1) Knowledge items are transformed to proof operators using the installed transformation functions. If no conversion is possible, the user is informed that the query cannot be correctly interpreted. (2) The obtained proof operators are augmented by further control information. For inferences there is the possibility to restrict their application direction using the **as** keyword. **Backward** restricts the applicability of inferences to those where all conclusions are instantiated. **Forward** specifies that no conclusion must be instantiated, and **close** that all premises and all conclusions must be instantiated.

Moreover, the automation API can be accessed via the underlying programming language using a **where** condition. As an example consider the definition of a standard simplification proof strategy shown in Listing 3.

first	selector	applies the first proof operators that succeeds.
solve	iterator	applies the strategy until it fails or a solution was found.
repeat	iterator	applies the strategy until it fails or until the condition evaluates to true.
try	combinator	applies the strategy and doesn't fail if strategy fails.
then	combinator	applies the first strategy and then the second strategy. Fails if the first strategy fails. If second strategy fails then first strategy backtracks internally and the second strategy is invoked again.
backtrack-if	combinator	adds a backtrack event specified in condition to the strategy.
cases	selector	applies the first strategy whose condition evaluates to true or pattern matches. On multiple matches for a single pattern a backtrack point will be defined for each match.
thenselect	combinator	executes a strategy and applies specified strategies to the resulting tasks

Table 3
Strategy Constructors with corresponding classification

```
define-strategy "Simplification"
  repeat
    first
      use select lhs=rhs from current
        where (greaterlpo lhs rhs ordering) as forward
      union
      use select lhs=rhs from current
        where (greaterlpo rhs lhs ordering) as backward
end;
```

CRStL Listing 3: Standard simplification encoded as proof strategy

3.3 Strategy Constructors

The final step in the specification of a proof strategy consists of augmenting the specified list of proof operators by control information needed for their execution. Essentially this consists of a specification of a condition for success and termination. Moreover, there is the possibility to specify a condition for failure, i.e., to directly invoke backtracking. An overview of the strategy constructors is shown in Table 3. The operators can be classified into the three categories **selector**, **iterator** and **combinator**.

Selectors. A list of given proof operators augmented by control information can be instantiated, resulting in a list of pais satisfying a specified set of constraints. **Selector** expressions determine how the pais are produced, and which pai to choose among the set of produced pais. For example, the **first** selector starts to instantiate the proof operators and applies the first which is applicable and satisfies the specified constraints. Another selector is the **cases** selector. It consists of a list of condition action pairs, where the condition is encoded in form of a matcher or a function call. It executes the first action whose condition is satisfied. Similar to the matching in select-expressions bound variables are passed to subsequent expressions.

```
cases
  * |- ~x -> use select "Contradiction" from base.inferences
  default -> use select "same" from base.inferences
end;
```

CRStL Listing 4: The **cases** construct

Listing 4 shows an example of the **cases** selector. The matcher `* |- ~x` encodes the condition that the goal of the current task is a negated formula. In this case, proof by contradiction shall be performed. As default case the inference "same" is applied.

Iterators. An **iterator** encodes a loop together with a default backtracking behavior. So far, we support two iterators, **solve** and **repeat**. **solve** tries to close the task to which it was applied to by repeatedly applying the specified proof operators. If none of them is applicable, it backtracks. It fails if the complete search space is traversed. **repeat** applies the specified proof operators until none of them is applicable anymore. Additionally, a termination condition can be specified using the **until** keyword.

Combinators. Combinators combine strategy expressions which they take as arguments. So far, we support four strategy combinatory, **try**, **then**, **backtrack-if**, and **thenselect**. Whereas the first two have their standard meaning, the latter need further explanation. **Backtrack-if** adds a failure condition to a specified condition and thus explicitly invokes backtracking. **Thenselect** analyses tasks resulting from a strategy application and maps them to new strategy applications. Note that by using conditions the mapping of subsequent proof strategies to tasks does not depend on the order of the tasks.

An example using the **thenselect** combinator is shown in Listing 5. The proof strategy "Induction" tries to perform an induction on the current goal. Successfully applied induction results in a list of subtasks containing step- and base-cases. It then applies a standard simplification to all base cases, and rippling to all step cases.

```
define-strategy "Induction"
  first
    use select axiom from current.axioms
        where (isinductionaxiom axiom)
    as backward
  thenselect cases
    task where (isbasecase task) -> "Simplification"
    default -> "Rippling"
  end;
end;
```

CRStL Listing 5: Induction proof strategy illustrating the **thenselect** construct

As we have already seen in the examples the matching facility is quite powerful. In addition to match top-level formulas, we also allow the matching of arbitrary subformulas.

This is in particular useful because ΩMEGA allows the application of inferences deeply inside formulas. We use the intuitive notation `[term]` that `term` can be a subterm. Moreover, we support the restriction of subformulas to specific polarities (c.f. [14]), where we use `+` to indicate a subformula with positive polarity (intuitively a goal to be shown) and `-` to indicate a subformula with negative polarity (intuitively a fact). The proof strategy shown in Listing 6 takes a formula as parameter and tries to derive a task in which the formula occurs with negative polarity and no dependencies.

```
define−strategy ”Fact”
  parameter formula;
  repeat
    use select ∗ from current.inferences as forward
  until ∗,[formula]− |− ∗
    where (not (proofobligations (pos formula)))
  backtrack−if (greater stratdepth 3)
end;
```

CRStL Listing 6: Proof strategy deriving a specified formula

4 Discussion and Future Work

In this paper we have presented the language **CRStL** for the encoding of proof techniques in form of proof strategies in the proof assistant ΩMEGA. With **CRStL** we aim to bridge the gap between the predefined proof operators and the underlying programming language of ΩMEGA. **CRStL** supports declarative and procedural descriptions of proof strategies, as well as a mixture of them. The main idea was to see the specification process of a proof strategy as a two-staged process, consisting of (1) the selection of proof operators and (2) augmentation of these proof operators with control knowledge.

The selection process is inspired by the query languages SQL, OQL and XPath, which are standard for querying structured knowledge such as relational/object-oriented data bases or XML. Indeed, we see the development graph as an object-oriented hierarchical data base, object-oriented in the sense that the different knowledge kinds correspond to different classes and the theories serve as a hierarchical structuring mechanism.

Searching and retrieving knowledge from mathematical data bases has been studied in the context of Helm [1] and Mizar [12]. Helm focuses on the interaction with different mathematical repositories over the web and Mizar uses the MML Query system[6] also for presentational purposes in MMLQT[5], such as for instance XPath is used in XSLT. Most similar to the strategy constructors, in particular the matching constructs, is the language $\mathcal{L}_{tac}$ [8]. The augmentation of proof operators with control knowledge can also be found in the ELAN system [7].

Future work comprises the refinement of our constructs. For example we plan to add sorting possibilities to the language to express preferences among a set of inferences more naturally. Moreover, so far the language supports only proof strategies working on a single agenda. As ΩMEGA already supports the management of multiple proof attempts in parallel, it is a natural step to extend the language to cope with multiple agendas. It would also be interesting to integrate lemma speculation in **CRStL**.

References

[1] Andrea Asperti, Luca Padovani, Claudio Sacerdoti Coen, Ferruccio Guidi, and Irene Schena. Mathematical knowledge management in helm. *Annals of Mathematics and Artificial Intelligence*, 38(1-3):27–46, 2003.

[2] S. Autexier. The CORE calculus. In R. Nieuwenhuis, editor, *Proceedings of CADE-20*, LNAI 3632, Tallinn, Estonia, july 2005. Springer.

[3] S. Autexier and D. Dietrich. Synthesizing proof planning methods and oants agents from mathematical knowledge. In J. Borwein and B. Farmer, editors, *Proceedings of MKM'06*, volume 4108 of *LNAI*, pages 94–109. Springer, august 2006.

[4] S. Autexier, D. Hutter, Till Mossakowski, and Axel Schairer. The development graph manager MAYA. In Hélène Kirchner and C. e Ringeissen, editors, *Proceedings 9th International Conference on Algebraic Methodology And Software Technology (AMAST'02)*, volume 2422 of *LNCS*. Springer, September 2002.

[5] Grzegorz Bancerek. Information retrieval and rendering with mml query. In *Proceedings of MKM'06*, pages 266–279. Springer-Verlag, 2006.

[6] Grzegorz Bancerek and Piotr Rudnicki. Information retrieval in mml. In *Proceedings of MKM'03*, pages 119–132, London, UK, 2003. Springer-Verlag.

[7] Peter Borovansky, Claude Kirchner, Hne Kirchner, and Christophe Ringeissen. Rewriting with strategies in ELAN: A functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–95, 2001.

[8] David Delahaye. A Proof Dedicated Meta-Language. In *Proceedings of Logical Frameworks and Meta-Languages (LFM), Copenhagen (Denmark)*, volume 70 (2) of *ENTCS*. Elsevier, July 2002.

[9] D. Dietrich. The task-layer of the ΩMEGA system. Diploma thesis, FR 6.2 Informatik, Universität des Saarlandes, Saarbrücken, Germany, 2006.

[10] Xiaorong Huang. *Human Oriented Proof Presentation: A Reconstructive Approach*. Number 112 in DISKI. Infix, Sankt Augustin, Germany, 1996.

[11] Jörg Siekmann, C. Benzmüller, A. Fiedler, Andreas Meier, and Martin Pollet. Proof development with OMEGA: $\sqrt{2}$ is irrational. In Matthias Baaz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 9th International Conference, LPAR 2002*, number 2514 in LNAI, pages 367–387. Springer, 2002.

[12] A. Trybulec and H. Blair. Computer assisted reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th Int. Joint Conference on Artifical Intelligence*. M. Kaufmann, 1985.

[13] Joris van der Hoeven. Gnu TeXmacs: A free, structured, wysiwyg and technical text editor. Number 39-40 in Cahiers GUTenberg, May 2001.

[14] Lincoln Wallen. *Automated proof search in non-classical logics: efficient matrix proof methods for modal and intuitionistic logics*. MIT Press series in artificial intelligence, 1990.

Comparison a human proof with a proof in Isabelle

Yoko Ono[1]

Department of Information Systems
Niigata University of International and Information Studies
Mizukino,Niigata,950-2292, Japan

Hidetsune Kobayashi[2]

Department of Mathematics
Nihon University
Kandasurugadai,Chiyoda-ku,Tokyo,1018308, Japan

Abstract

Human proof and proof of a theorem by Isabelle/HOL are compared. A user friendly proof style is proposed for a proof assistant system. The necessity for the standardization of definitions to some fundamental concepts and the necessity for a database with elementary properties are established.

Keywords: formalization, automated reasoning system, Isabelle, contingency tables, descents

1 Introduction

There are two ways of understanding mathematical objects. One, as shown in the famous book "Anshauliche Geometrie" by Hilbert and Cohn-Vossen[3], is understanding mathematical objects with imagination or intuition by drawing a representative figure. Some simple formulae related to numbers, substitutions and changing the order are expressed symbolically by figures; we can take these as examples of mathematical objects understandable with imagination. Another way is understanding mathematical objects logicallythat is, expressing and understanding them logically. This is the main accepted way, because almost all mathematical objects treated in modern mathematics are invisible and cannot be drawn. They areabstract ideas and only the logical approach is available for us to express and understand them.

To understand a logically expressed theorem and its proof, we might say that we check the details and grasp the main concept and then reconstruct the whole

[1] Email: onoyk@nuis.ac.jp
[2] Email: hkb@formalg.com

proof. Finally we understand. Sometimes we see proof steps through analogy using small examples.

Formalized mathematical objects in a proof assistant (PA) are represented entirely by logical expressions. We cannot apply imagination in PAs at present (although it might be possible). Proofs in a PA have no logical gap, therefore they should be easy to understand; but actually it is quite hard for us. The main reason is that when we are reading a proof we lose our way to the conclusion. This is because, in addition to key or subkey lemmas, many other lemmas are required in a PA proof; the mixture of main proofs and auxiliary proofs make the total logical structure ambiguous. In a PA proof, when we apply a lemma as a rule, we have to show that all of its premises are satisfied. We perceive almost all of them as trivial and so their proofs may be omitted when we write our proofs. But in PA proofs, even the contents are simple; each proof requires some steps. Moreover, in a PA, long steps are required before we reach the kernel part of a proof. Consequently we have a long labyrinth proof.

As a user, we propose a PA proof style having only key (or subkey) lemmas and the outline of a proof as a comment. The PA system should fill the gaps between given lemmas according to the outline given as a comment.

We also request the following. These are insufficient in the current PA, but seem not very hard to resolve. A good PA, as its name implies, ought to have the following functions:

(i) Simple properties should be resolved automatically.

(ii) There should be a database of fundamental concepts. While the MIZAR group has a big database of formalization, what we want is as in a textbook that they should be collected under consistent concepts. Definitions of fundamental concepts should be standardized within a couple of variations, since according to the starting point, choice of some definition should be allowed. Since a PA treats abstract mathematics, it is hard to use definitions of this type; but if we have standard definitions, we can use, for example, matrix as we use it in Maple or in Mathematica.

The target of this paper is to propose an understandable proof style and a human-friendly PA system. For the first proposal, we compared a PA proof written by one of the authors and a proof from a paper. Since we need to compare proofs in a variety of fields, the example we selected in this paper is suitable for two reasons. First, the theorem is easy to see without having a mathematical background, but the proof is not so easy. Secondly, the proof in the paper is not built by taking logical steps from an axiom; instead, the author gives a simple example and by analogy makes us understand the proof.

In section 2, we give an example of a theorem to be compared. We give a brief introduction to the theorem. We also give a PA proof in an easy-to-understand style. In section 3, we present part of a full PA proof classified into blocks. Each block includes a key or sub-key lemma, preparations for the main lemma, and lemmas to sweep out those subgoals derived from the main. In section 4, we collect trivial but necessary lemmas with comments. Using the categories of name, premises, conclusion and comment, we can find a proper lemma from a database. In section

5, we write conclusions.

2 Comparison human proof with PA proof

As a theorem to be compared, we have chosen the Foulkes' enumeration theorem of contingency tables, because contingency table is familiar to us and proof requires only elementary mathematical background.

We give a brief introduction to the contingency table. A contingency table has data showing a relation between two variables. For example, following is a contingency table representing relations between the number of smokers and the number of patients of cancer.

Table 1
Example of contingency table

	smoker	non smoker	total
with cancer	50	20	70
no cancer	10	40	50
total	60	60	120

Generalizing this, we give m-dimensional vector $\boldsymbol{r}$ and n-dimensional vector $\boldsymbol{c}$. An (m, n)-matrix T with natural number entries satisfying $\sum_{j=1}^{m} T_{ij} = c_j$ and $\sum_{i=1}^{n} T_{ij} = r_j$ is a contingency table. In statistical testing, it is necessary to enumerate number of contingency tables satisfying above conditions for fixed $\boldsymbol{c}$ and $\boldsymbol{r}$. In their paper[1], Diaconis and Gangolli gave a survey of contingency table enumeration. Among several enumerating methods, it is said Foulkes' method[2] is easy to understand for us. We quote the theorem and its proof.

Theorem 2.1 *Let $\boldsymbol{r}$ and $\boldsymbol{c}$ be compositions of N. The number of permutations π in S_N with $D(\pi) \subseteq D(\boldsymbol{r})$ and $D(\pi^{-1}) \subseteq D(\boldsymbol{c})$ is $|\sum_{\boldsymbol{rc}}|$.*

Proof. Consider two comparisons, $\boldsymbol{r}$ and $\boldsymbol{c}$ of N. A permutation π can be represented by a permutation matrix

$$\rho(\pi)_{ij} = \begin{cases} 1 \text{ if} \pi(i) = j \\ 0 \text{ otherwise.} \end{cases}$$

Thus, the 1 in the ith row is in position $\pi(i)$ and $\pi(i+1)$¡$\pi(i)$ says the 1 in the $i+1$st row occurs to the left of the one in the ith row. Divide ρ into blocks specified by $\boldsymbol{r}$ and $\boldsymbol{c}$. Then, π has $D(\pi) \subseteq \{r_1, r_1 + r_2, \cdots, r_1 + \cdots + r_{m-1}\}$ if and only if the pattern of ones in each horizontal strip decreases from upper left to lower right. Since $\rho(\pi^{-1}) = \rho(\pi)^t$, $D(\pi^{-1}) \subset \{c_1, c_1 + c_2, \cdots, c_1 + \cdots + c_{n-1}\}$ if and only if the pattern of ones in each vertical strip decreases from upper left to lower right.

With this representation, there is one-to-one correspondence between tables $T \in \sum_{\boldsymbol{rc}}$ and permutations satisfying the constraints: from a permutation matrix with T_{ij} ones in the (i, j) block which also satisfies the monotonicity constraints. There is a unique way to do this: the T_{11} ones in the $(1, 1)$ block must be contiguous along the diagonal, starting at $(1,1)$. The T_{12} ones in the $(1,2)$ block must be contiguous on a diagonal starting at $(T_{11}+1, c_1+1$; that is as far to the left and high up as possible consistent with the monotonicity constraints. The first horizontal block of $\rho(\pi)$ is similarly specified. Now the entries in the $(2,1)$ block and then the second horizontal strip are forced and so on. Continuing, we see that $\rho(\pi)$ is uniquely determined by T. □

The fundamental idea of the theorem is

(i) Let $\sum c_j = N$, we divide (N,N)-permutation matrix P into blocks width and height of which is given by c_j and r_i respectively.

(ii) Let T_{ij} be the number of ones in the (i,j) block of P, then we have an (m,n) matrix T

(iii) Give an algorithm deciding a permutation matrix from T.

(iv) Let f be the permutation f giving P. If f satisfies the monotonicity condition, the P coincides with the permutation matrix determined by the table T.

(v) The number of permutations satisfying the monotonicity condition is equal to the number of contingency tables.

We present PA proof of Foulkes' theorem, following the above idea. `P_matrix` is a square matrix having only one 1 in each row and column and the other entries are 0. A `N_matrix` is a (m,n)-matrix with natural number entries. A `composition_n` is a vector entries of which are natural numbers. Let `c` be (2,2) and `r` be (1,3) then `P_matrix` is divided into `horizontal_strips (vertical_strips)` and blocks as

```
        2       2
    |-------|-------|
 1  | (1,1) | (1,2) |      1-st horizontal_strip
    |-------|-------|
    |       |       |
 3  | (2,1) | (2,2) |      2-nd horizontal_strip
    |-------|-------|
```

`Comp_to_D r c d` is a function `d` such that `d` ∈ `extensional{1..r}` ∧
(∀ `j`∈`{1..r}. if j = 1 then d 1 = c 1 else d j = (` ∑ `k=1..j. (c k)))`
The algorithm giving (i, j) block of a `P_matrix` from a contingency table `T` is:

```
T_to_Perm_block i j T r c P ==
(∀ x ∈ (horizontal_strip i r). ∀ y ∈ (vertical_strip j c).
(if ( ∃ s ∈ {1..(T i j)}. x = ( ∑ k=1..(i - 1). (r k))
+(∑ k=1..(j - 1). (T i k))  + s ∧ y = (∑ l=1..(j - 1). (c l))
+ (∑ l=1..(i - 1). (T l j)) + s)
then (P x y = 1) else (P x y = 0)))
```

In the proof of Foulkes' proof, this algorithm is expressed as form a permutation matrix with T_{ij} ones in the (i, j) block .... There is a unique way to do this: the T_{11} ones in the (1, 1) block must be ... and so on An expert could understand that this proof shows that any permutation matrix P satisfying monotonicity constraints is obtained by a contingency table. Before proving the theorem, the paper gives a simple example and see how the permutation matrix is determined. We observe that this way is understanding with imagination obtained from simple examples.

In PA the main part of the theorem is expresses as

```
lemma T_to_P_surjection01:"[|n matrix m n T; permutation N f;
   permutation_matrix N f P; composition_n N m r; composition_n N n c;
   P_to_Table N m n r c P T; 0 $<$ T im jm; Comp_to_D m r dr;
   Comp_to_D n c dc; descents N f ⊆ dr  {1..m - 1};
   descents N (fp N ) ⊆ dc  {1..n - 1};im ∈ {1..m}; jm ∈ {1..n};
   ∀ x∈ {1..im - 1}. ∀ j∈ {1..n}. T to Perm block x j T r c P;
   ∀ x∈{1..jm - 1}. T to Perm block im x T r c P; s ∈ {1..T im jm};
    P (setsum r {1..im - 1} + setsum (T im) {1..jm - 1} + s)
   (setsum c {1..jm - 1} ( l = 1..im - 1. T l jm) + s) = 1|] ⟹ False
```

This lemma is obtained denying the conclusion of the following final lemma.

```
lemma T to P surjection:"[|n matrix m n T; permutation N f;
   permutation matrix N f P; composition n N m r;
   composition n N n c; P to Table N m n r c P T;
   Comp to D m r dr; Comp to D n c dc;
   descents N f ⊆ dr  {1..(m - 1)};
   descents N (fp N ) ⊆ dc  {1..(n - 1)}|]
   ⟹ ∀ i∈{1..m}. ∀ j∈ {1..n}. T to Perm block i j T r c P"
```

We present key lemma and sub-key lemmas of a proof of `T_to_P_surjection01` in PA. (* ... *) is a comment.

```
(* Outline of the proof is:
   find the least s ∈ {1..T im jm} such that

P (setsum r {1..im - 1} + setsum (T im) {1..jm - 1} + s)
(setsum c {1..jm - 1}+
( l = 1..im - 1. T l jm) + s) = 1,

We have two cases
            (1) the least number is 1
            (2) the least number is not 1

   In the case (1), since we have T im jm is nonzero, there is at
    least one pair (x, y) such that P x y = 1. Using this pair,
    we derive False.
   In the case (2), we have one on the diagonal and if s < T im jm
    then we  see False is derived. *)

apply (cut_tac n = "T im jm" and
       P = " λ z.
     (P (setsum r {Suc 0..im - Suc 0} +
           sesum (T im) {Suc 0..jm - Suc 0} + z)
          (setsum c {Suc 0..jm - Suc 0} +
            (l = Suc 0..im - Suc 0. T l jm) + z) = 0)"
         in less_NMin)

 (* We obtained the least s satisfying less_NMin,
    which we denote as ma  *)

 apply (case_tac "ma = 1")

(*** Beginning with 0 at the corner

                 |      jm       |
           ----- |-------------- | ------
                 |               |
                 |   0-----------|       im
                 |   |           |
           ----- | --|---------- | ------                  ***)
(* There is one on the same column and on the same row
                                 |
                     0-----------|------1-----
                     |           |
                    -----------------------
                     1           |
                     |           |                              *)
apply (frule_tac permutation_matrix_1_exist[of N f P])
       (* got ones on the column and on the row respectively *))
apply ((* There is one in this block.
        And by virtue of descent conditions, we have contradiction.*)

apply ((* There is one in this block.
  And by virtue of monotonicity conditions,we have contradiction. *)
      frule tac i = im and j = jm in P to contingency
      ex 1[of N P m r n c T],assumption+, (erule bexE)+,
        (* location of one is restricted by following two lemmas *)
       frule tac im = im and jm = jm and x = x and y = y in
       T P h range of 1[of m n N r c T P], assumption+,
       frule tac jm = jm and im = im and x = x and y = y in
          T P v range of 1[of m n N r c T P], assumption+)
 apply ((* Now, we apply monotonicity condition.
       Then we have x < i and i < x.
       lemma  perm matrix desc ver is the key for the proof of
       T_to_P_surjection*)
        frule tac j = jm and u = "setsum c {1..jm - 1} +
          ( i = 1..im - 1. T i jm) + 1" and v = y and s = i and t = x in
        perm matrix desc ver[of N f P n c dc], assumption+)
```

```
(* After several steps of cleaning, the case (1) is completed *)
```

3 Grouping PA proof steps

In human proof, as shown in the previous section, we consider proof with some key ideas and the details are resolved in our head. Therefore proof to those details are not written explicitly in the human proof. On the other hand, in PA proof the proofs of the main part and the proof of the subgoals derived by key (or sub-key) lemmas are written together with proofs of main part. Therefore PA proofs become longer and longer, and even for those who have written the PA proof it is hard to read the proof later. Here, we show which lemma is a key or a sub-key, and which lemma is required as an auxiliary lemma for the key or a sub-key. We divide a proof into groups, and show how the groups work.

```
   frule permutation matrix P matrix[of N f P], assumption+,
   frule P to contingency[of N P m r n c T], assumption+,
```

Above two add `P_matrix N P` and `contingency_table m n N r c T` to premises. These two premises are used repeatedly, hence we put them here.
`cut_tac n = "T im jm"` and `P = "λz.`

```
  (P (setsum r {Suc 0..im - Suc 0} +
    setsum (T im) {Suc 0..jm - Suc 0} + z)
   (setsum c {Suc 0..jm - Suc 0} +
   (l = Suc 0..im - Suc 0. T l jm) + z) = 0)" in less NMin)
```

This is a subkey, and shows there exists minimum s satisfying condition `P`. This `cut_tac` gives a subgoal asking the existence of a such that `P` a holds. The following `rule_tac`, three `frules`, `assumption, blast and assumption` + are lemmas used to clean up subgoals given by above `cut_tac`.

```
  rule tac x = s in bexI, fold One nat def, simp del:One nat def
        atLeastAtMost iff
        frule horizontal range[of m n N r c T im jm s], assumption+,
        frule vertical range[of m n N r c T im jm s], assumption+,
        frule P matrix 0 1 str[of N P m r n c im jm
          "setsum r {1..im - 1} + setsum (T im) {1..jm - 1} + s"
          "setsum c {1..jm - 1} + ( k = 1..im - 1. T k jm) + s"],
        assumption+,
        blast, assumption+)

(* Thus, we have sown the existence of the minimum value of s.

  apply (erule bexE, (erule conjE)+)

 (* This is preparatory for the next step.
 We obtained the minimum ma. We consider the case where ma = 1. That
 implies we have 0 at the beginning corner.
```

```
                  |      jm      |
           ------ |------------- | -----------
                  |              |
                  |   0----------|       im
                  |   |          |
           ------ | ------------ | ------       ***)

apply (case tac "ma = 1",
       simp del:One nat def atLeastAtMost iff, thin tac "ma = 1" )
(* There is a one on the same column and on the same row respectively.

                                  |
                      0-----------|------1-----
                      |           |
                      |           |
                    -----------------------
                      |           |
                      1           |

The following block shows this.
The key of this block is permutation matrix 1 exist[of N f P]
and other lemmas are preparation
and resolving the derived subgoals.*)

apply (frule tac i = im and j = jm and s = 1 in
       horizontal range[of m n N r c T], assumption+,
       frule vertical range[of m n N r c T im jm 1], assumption+,
       frule hor strip subset[of N m r im], assumption+,
       frule subsetD[of "horizontal strip im r" "{1..N}"
       "setsum r {1..im - 1} + setsum (T im) {1..jm - 1} + 1" ],
       assumption+,
       frule ver strip subset[of N n c jm], assumption+,
       frule subsetD[of "vertical strip jm c" "{1..N}"
          "setsum c {1..jm - 1} + ( k = 1..im - 1. T k jm) + 1"],
           assumption+,
       frule tac permutation matrix 1 exist[of N f P], assumption+,
        rotate tac -1, frule conj 1, drule conj 2)
```

4 Elementary properties

We present simple but useful lemmas for proving Foulkus' theorem (scripts[4]). Proofs to these lemmas are simple and there are some which can be proved by `simp` only, but the other require several steps to prove. Moreover, if the number of expressions in the premises of a lemma is more than twenty, then sometimes simp wouldn't work quickly.

When we are writing proofs in PA, those properties below should be proved as short as possible, because in human proof such properties are taken trivial and proof is not writen explicitly. Or those simple properties should be proved automatically, even though we do not apply a rule.

In general, it is not easy to collect such properties, because we can find them when we write proofs in PA.

Name	**"(premises ⟹) conclusion"**	**comment**
`add_0_r` :	`"(n::nat) + 0 = n"`	n plus 0
`Suc_pred1` :	`"0<(n::nat)`	n minus plus 1
	`⟹ n - 1 + 1 = n"`	
`add_Suc_right1`:	`"(n::nat) + 1 = Suc n"`	
`Sum_exceed2`:	`"[|(a::nat) < y; y ≤ a + b|]`	interval plus
	`⟹ ∃ s∈{1..b}. y = a + s"`	
`interval_sub2`:	`"[|c ≤ a; b ≤ d|]`	subinterval
	`⟹ {(a::nat)..b} ⊆ {c..d}"`	
`add_less_mono2`:	`"(i::nat) < j`	less `slide`
	`⟹ k + i < k + j"`	
`nset_subTr1`:	`"[|{(a::nat)..b} ⊆ {c..d};`	subinterval `le`
	`a ≤ b|]`	
	`⟹ b ≤ d"`	
`diff_1_le`:	`"[|(1::nat) ≤ h; h < j|]`	le minus 1
	`⟹ h ≤j  1"`	
`diff_1_le_pre`:	`"m < n + (1::nat)`	`le` n + 1 minus 1
	`⟹ m ≤ n"`	
`mem_nset_sub`:	`"i ∈ {(1::nat)..n - 1}`	in long interval
	⟹ i ∈ 1..n"	
`nset_not_sub`:	`"[|j∈{1..Suc n};`	not in short interval
	`j ∉ {1..n}|]`	
	`⟹ j = Suc n"`	
`nset_not_sub1`:	`"[|j∈{a..a + Suc n};`	not in short interval
	`j ∉ {a..a + n}|]`	`slide`
	`⟹ j = a + Suc n"`	

Name	**"(premises $\Longrightarrow$) conclusion"**	**comment**
`part_substitute:`	`"[|(a::nat) = b + c; c = d|]` $\Longrightarrow$ `a = b + d"`	`substitute` second
`part_substitute1:`	`"(b::nat) + c = d` $\Longrightarrow$ `a + b + c = a + d"`	two together
`nat_add_left_cancel1:`	`"(k::nat) = j` $\Longrightarrow$ `(k + m =j + n) = (m = n)"`	`slide` equality
`nat_add_left_cancel2:`	`"((k + m + i) = k + n + (l::nat)) = (m + i = n + l)"`	`assoc slide` equality
`less_exchange00:`	`"[|a < y; a = x|]` $\Longrightarrow$ `x < y"`	`substitute` inequality
`nset_le:`	`"(j::nat)∈{1..n}` $\Longrightarrow$ j ≤n"	`mem` interval le upper bound
`nset_le1:`	`"(j::nat)∈{a..b}` $\Longrightarrow$ `j ≤ b"`	`mem` interval `le` upper bound general
`nset_gt:`	`"(j::nat)∈{1..n}` $\Longrightarrow$ `0 < j"`	j greater than 0
`nset_gt1:`	`"(j::nat)∈{a..b}` $\Longrightarrow$ `a ≤ j"`	`mem` interval ge lower bound
`diff_1_less:`	`"1 ≤(n::nat)` $\Longrightarrow$ `n - 1 < n"`	n - 1 less than n
`minus_1_mem_nset:`	`"[| j ∈ {1..(n::nat)}; j = 1|]` $\Longrightarrow$ `j-1 ∈ {1..n}"`	j - 1 `mem` interval
`In_set_with_cond:`	`"a ∈ {x ∈ A. P x}` $\Longrightarrow$ `a ∈ A ∧ P a"`	`mem` set with condition
`In_set_with_cond_rev:`	`"a ∈ A ∧ P a` $\Longrightarrow$ `a ∈ {x ∈ A. P x}"`	condition `mem` set

Name	**"(premises ⟹) conclusion"**	**comment**
not_in_set_with_cond:	"x ∉ {y. P y} ⟹ ¬ P x"	not mem set with condition
Suc_leI1:	"(a::nat) < b ⟹ a + 1 ≤b"	a + 1 le b
mem_interval_s:	"(a::nat) ≤b ⟹ a∈{a..b}"	lower bound mem interval
le_pred_less:	"(a::nat) + 1 ≤ b ⟹ a < b"	a less b
finite_interval0:	"finite {(a::nat)..b}"	interval is finite set
lessI1:	"(n::nat) < n + 1"	n less n + 1
diff_1_less _bound_interval:	"(j::nat) ∈ {1..n} ⟹ j - 1 ≤n"	mem interval diff 1 le
not_exists_ge:	"[| ¬ (∃ s ∈ {(1::nat) ..m - 1}. x = a + s); a + 1 ≤ x|] ⟹ a + m ≤x"	not mem slide interval ge
Sum_le:	"((k=1..(n::nat). ((s k)::nat)) ≤ (k=1..(n + m). (s k)))"	Sum monotone
Sum_le1:	"n ≤ (m::nat) ⟹ ((k=1..n. ((s k)::nat)) ≤ (k=1..m. (s k)))"	Sum monotone general
Sum_split2:	"n ≤ m ⟹ (i=1..m. r i) = (i=1..n. r i) +(i=(n + 1)..m. r i)"	split summation

Name	**"(premises $\Longrightarrow$) conclusion"**	**comment**

`Sum_hor_vert:`	`"( s=a..(a + m).`
	`( t=b..(b + n).`
	`((M s t)::nat)))`
	`=( t=b..(b + n).`
	`(s=a..(a + m). M s t))"`
		change sum order

5 Conclusion

We need a database with elementary properties, and the database should be designed to allow for a search of a proper lemma not only by name but also by premise, conclusion, and description of a characteristic of the lemma.

We propose that definitions of fundamental mathematical objects should be standardized and that a database for definitions should be open to users. We propose a proof format that provides only key or sub-key lemmas and between them, the PA should fill by automated reasoning.

To demonstrate realizing this type of PA, we divided proofs into groups, observed preparatory lemmas that are required to derive premises of the key (or sub-key), and after applying the key (or sub-key), observed lemmas required to sweep out the sub-goals given by the key (or sub-key). Proofs for such auxiliary lemmas are not very difficult, and backward inference seems to work rather efficiently.

Many theorems in mathematics require long proof in PA; for example, the lemma `T_to_P_surjection01` required about 500 lines. However, Diaconis and Gangolli used only twenty lines for the proof of Foulkes' theorem. Therefore, we require a short proof of the PA.

References

[1] Diaconis, P. and Gangolli, A., *Rectangular Arrays with Fixed Margins*, Discrete Probability and Algorithms (D. Aldous et al., eds.)(1994), 15–41, Springer, New York.

[2] Foulkes, H., *Eulerian numbers, Newcomb's problem and representations of symmetric groups*, Discrete Math. 30 (1980), 3–99.

[3] Hillbert, D. and Cohn-Vossen, S., "Anshauliche Geometrie," Springer, Berlin, 1932.

[4] Schank,R. C. and Riesbeck, C. K. "Inside Computer Understanding: Five Programs Plus Miniatures," Lawrence Erlbaum, 1981.

Ideas over terms generalization in Coq

Vincent Siles[1,2]

LIX/INRIA/Ecole Polytechnique
Palaiseau, France

Abstract

Coq is a tool that allows writing formal proofs and check their correctness in its underlying logic framework, the *Calculus of Inductive Constructions*. Coq's type system handles dependent types, so we should be able to write dependently typed programs with it. However writing complex programs with their full specification can be a rather difficult exercise. In this position paper, I discuss to what extent we can use the mechanism already implemented in Coq to make the work of the programmer easier, especially with the use of Coq's unification algorithm.

Keywords: type inference, generalization, dependently typed programming

1 Introduction

Coq's underlying logic framework, the *Calculus of Inductive Constructions* [1] is a strong basis for using dependent types. It is a powerful way to write regular programs along with a specification. However, this exercise can become arduous if you want a precise specification or if you are using complex proofs. In this position paper, I would like to show to what extend we can use the mechanism already implemented in Coq to ease the work of the programmer, in particular with the use of Coq's unification algorithm to compute missing terms.

The main idea is to be able to write ML-like program in the more expressive type system of Coq, with the same support for type inference and polymorphism that we can find in *OCaml* or *MLF* [2], where the computer can, to some extent, infer some type information omitted by the programmer.

2 Using Coq's unification to compute omitted terms

Implicit syntax and unification

In CIC, every binder is *explicit*. All of the implicit syntax mechanism of Coq is just syntactic sugar to be more user-friendly, which mainly add holes (also called *ex-*

[1] Thanks to my supervisors Hugo Herbelin and Bruno Barras
[2] Email: vincent.siles@lix.polytechnique.fr

istential variables or *evars*) in the input term, expecting that the unification will fill them. In other languages like the variant of Calculus of Implicit Constructions [4] with decidable type-checking [5], the implicit syntax mechanism is in the core language, but the user still has to point out which argument of his function can be left implicit.

My claim is that some of those implicit arguments can be automatically guessed by the computer, and so the user can omit them entirely from his code. This however, requires higher-order unification which is a complex problem with the rather unfortunate property of being undecidable. Hence we tried, using an experimental test version of Coq (based on the version 8.1), to see to which extent Coq's algorithm can "guess" this information.

Type inference

As mentioned above, all the binders are explicit. The consequence is that type inference will not add binders to the terms entered by the user, it will only try to compute missing information from implicit syntax hints. We have implemented the ideas of this paper in a test version of Coq to mimic the behaviour of ML-languages. We will now have a close look to some basic example, to understand how Coq deals with this problem.

```
Coq < Definition id x :=x .
Error: Cannot infer a type for x
```

If we check the internal representation of the term above, we will see that Coq added an evar for the type of x and failed to fill it. But it is clear here that any type could have been used to fill this hole, because this hole is not defined yet, and has no constraint on it (except being a valid type, a constraint we will omit to mention henceforth). As a first try, we could just add enough binders in front of a term with holes to bind those *constraint-free holes* and obtain the following behaviour:

```
Coq < Definition id x :=x .
id is defined
Coq < Print id.
id = fun (T : Type) (x : T) => x
     : forall T : Type, T -> T
```

Now we will consider a new example : the application function.

```
Coq < Definition app f x := f x.
Error: Cannot infer a term for an internal placeholder
```

This time, the problem is more subtle and is a consequence of the property of subtyping in Coq's type system. The system will manage to add some evars and will have (approximately) the following constraints[3] :

$$[\,] \vdash f :?a \rightarrow ?b \quad [f :?a \rightarrow ?b] \vdash x :?c \quad [\,] \vdash ?c \leq ?a$$

We only require x's type to be a subtype of f's domain. So the previous approach is broken because it would build the term

[3] $?c \leq ?a$ stands for "$?c$ is a subtype of $?a$"

`Definition app A B (f:A->B) x := f x` which has still the constraint over `x`'s type to be solved. This is were the subtyping system comes handy. Thanks to the subtyping rules[4], we can prove that if you are a subtype of a variable, then you are equal to this variable. So, we can generalize $?a$ and $?b$ into type variables A and B, so that the remaining constraint is changed to $[] \vdash ?c = A$ and can easily be solved. Our `app` function can now be computed with its full type information:

```
Coq < Definition app f x := f x.
app is defined
Coq < Print app.
app =
fun (T T0 : Type) (f : T -> T0) (x : T) => f x
     : forall T T0 : Type, (T -> T0) -> T -> T0
```

From this example, we learned that it is necessary to collapse all the constraint-free variables that are related by subtyping relations. This can be easily done if we look at the evars to be the nodes of a graph where the edges are the subtyping constraints: every connected graph stands for one variable we can bind in front of the term.

Recursive functions

With non-dependent types, the case of recursive functions works fine. Since we have no dependencies to deal with, we can add the binders *before* the fixpoint definition to close the term without interfering with the recursive definition. This will not be the case with dependent types, as we will see later.

```
Coq < Fixpoint length l :=
Coq < match l with
Coq < | nil => 0
Coq < | cons _ l' => 1+ (length l')   end.
Error: Cannot infer a term for an internal placeholder

 (* The test version should have produced the following term *)
Coq < Definition length A  := fix length (l:list A) :=
Coq <  match l with
Coq <  | nil => 0
Coq <  | cons _ l' => 1 + (length l')   end.
length is defined
```

Generalizing into ML

Most ML type inference algorithm have a *gen* rule (e.g. the Hindley-Milner Algorithm, or MLF's *gen* rule [2]), which is related to the use of implicit products in the langage. But Coq does not have such rule or product. Here we saw that we can postpone this step to be the very last and mimic the *gen* rule just by adding binders for every constraint-free evars (with respect to the subtype relation). Thanks to this, we can now type directly in Coq our favorite *OCaml* programs, with very little syntactic changes, even for recursive functions.

[4] http://coq.inria.fr/V8.1pl3/refman/Reference-Manual006.html#toc26

3 Coq: a dependently typed programming language ?

The next step is to extend the previous algorithm to dependent types. We will consider two examples of programs that we would like to be type-checkable in a version of Coq with implicit generalization .

```
Coq < Print vector.
Inductive vector (A : Type) : nat -> Type :=
    Vnil : vector A 0
  | Vcons : A -> forall n : nat, vector A n -> vector A (S n)
```

Length

```
 Fixpoint length v := match v with
  | Vnil => 0
  | Vcons _ _ w => 1 + (length w)   end.
```

The first function is a dumb length function which computes the length of a vector as if it were a list: we do not care about the length information in the type. With the approach of the previous section, we can successfully fill the holes and type check the term, but a major change occurs: adding binders in front of the term is not enough, now we have to modify the structure of the term itself to take into account the dependency of v over its length.

```
Coq < Definition length A n := fix length (v:vector A n) {struct v} :=
Coq <   match v with
Coq <    | Vnil => 0
Coq <    | Vcons _ _ w =>1+(length w)   end.
Toplevel input, characters 127-128
>    | Vcons _ _ w =>1+(length w)
>                              ^
Error:
In environment
n : nat
A : Type
length : forall (v : vector A n) , ?3
v : vector A n
a : A
n0 : nat
w : vector A n0
The term "w" has type "vector A n0" while it is expected to have type
 "vector A n"
```

This first definition is the case of a global generalization, and will be ill-formed because of the recursive call, since the length of w is not n but n-1.

```
Coq < Definition length A := fix length n (v:vector A n) {struct v} :=
Coq <   match v with
Coq <     | Vnil => 0
Coq <     | Vcons _ _ w =>1+(length _ w)   end.
length is defined
```

The second one is a local generalization, which will preserve the link between v and its size.
So we have to be careful were we put the binders, and we have to also change the recursive call (you can notice that we add an _ in the recursive call) to take into account the new argument. This is the most dangerous step because if we change the structure of a term, we may break several constraints or results already computed by Coq, so we have to redo the inference step from the beginning, which may be quite costly.

With this enhancement of the algorithm, however, the length function can be defined as we desired. Note also that the length, which is not used in the computation, has been inferred.

Append

```
 Fixpoint append v w {struct v} := match v with
  | Vnil => w
  | Vcons a _ v' => Vcons a _ (append v' w)   end.
```

The second function takes two vectors and puts the second at the end of the first one to build a longer vector. Again this time, the length of the vectors are not used in the computation, only to type check the recursive call. This time, however, the right length is not available in the context of the evar as it was in the *length* function; it has to be computed from the length of v', w and the plus function. This is the kind of problem that will break our algorithm, because Coq v8.1's unification algorithm only knows how to solve equations of the form $?x\ x_1 \dots x_n = x_i$, while here one needs to solve equations of the form $?x\ O = a$ and $?x\ (S\ n) = b$: it will not be able to reconstruct the *plus* function for natural numbers from scratch.

There is also a hidden problem in the typing of the match, in its return type. If we do not help Coq here, it will infer the return type to be the type of w instead of being of type *vector* p where p is the sum of the length of v and w (this problem has more to do with typing dependent match and is out of our scope here), hence failing to infer the right return type, leading to an ill-typed term.

In this case, the only solution we could think of has been to provide the return type of the function and the return argument of the match by hand, so that Coq will manage to infer the right return type and also the use of the plus function in the recursive call.

```
Coq < Fixpoint append (A:Type) n (v:vector A _)
                               m (w:vector A _) : vector A (n+m) :=
Coq < match v  in (vector _ p) return (vector A (p+m)) with
Coq < | Vnil => w
Coq < | Vcons a n’ v’ => Vcons _ a _ (append _ n’ v’ _ w)   end.
append is recursively defined
```

But in so doing we have given almost all of the typing information of the term, precisely what we wanted to avoid in the first place.

4 Conclusion and Future work

The non-dependent part of this attempt at an algorithm has been partially implemented in an experimental extension of Coq (everything but the recursive function support, due to the complexity of Coq's fixpoint handling) and is working well, but since the dependent part is really not the good way to solve this problem, this implementation has been dropped.

The two main problems of type inference we spotted are quite different, but they are both important open problems concerning programming with dependent types:

- Inferring terms built from the ones in the context: we could do simple arithmetic but something more robust like the *Program tactic* [6] may be a better one.
- Dependent matching: inferring the return clause of a match, or getting all the information we can from the pattern matching is currently under discussion among the Coq Development Team. *Epigram* [7] has a rather interesting solution but it still requires some work from the user. We still hope to infer some of those hints automatically.

References

[1] Coq Development Team, *The Coq Proof Assistant Reference Manual*, URL:http://coq.inria.fr/V8.1pl3/refman/index.html.

[2] D. Rémy, D. LeBotlan, B. Yakobowski, *MLF*, URL:http://pauillac.inria.fr/ remy/mlf/.

[3] Ulf Norell, *Towards a practical programming language based on dependent type theory*, Phd thesis, Chalmers University of Technology, 2007.

[4] A. Miquel, *Le Calcul des Constructions implicite: syntaxe et smantique*, Phd thesis, Université Paris 7, 2001.

[5] B. Barras,B. Bernardo, *The Implicit Calculus of Constructions as a Programming Language with Dependent Types*, FoSSaCS (2008), 365–379.

[6] Matthieu Sozeau, *Russell and the tactic Program*, URL:http://mattam.org/research/russell.en.html.

[7] H. Goguen, C. McBride, J. McKinna, *Eliminating Dependent Pattern Matching*, Essays Dedicated to Joseph A. Goguen (2006), 521–540.