Retargeting OpenAxiom to Poly/ML: Towards an Integrated Proof Assistants and Computer Algebra System Framework

Gabriel Dos Reis[1], David Matthews[2], and Yue Li[1]

[1] Texas A&M University,
College Station, USA
[2] Prolingua Ltd,
Edinburgh, Scotland

Abstract. This paper presents an ongoing effort to integrate the AXIOM family of computer algebra systems with Poly/ML-based proof assistants in the same framework. A long-term goal is to make a large set of efficient implementations of algebraic algorithms available to popular proof assistants, and also to bring the power of mechanized formal verification to a family of strongly typed computer algebra systems at a modest cost. Our approach is based on retargeting the code generator of the OpenAxiom compiler to the Poly/ML abstract machine.

Keywords: OpenAxiom, Poly/ML-based proof assistants, runtime systems.

1 Introduction

Computer algebra systems seek efficient implementations of algebraic algorithms. As it happens, they occasionally "cut corners", making assumptions that are not always obvious from official documentations. Proof assistants, on the other hand, focus on mechanized proofs; they are uncompromising on formal correctness at the expense of efficiency. In this paper, we report on an ongoing effort to integrate OpenAxiom [19], a member of AXIOM family of strongly-typed computer algebra systems, with proof assistants (notably Isabelle/HOL [12]) based on the Poly/ML [20] programming system.

A large body of work [3,5,4,11,2,6,10] exists on interfacing computer algebra systems with logical frameworks. Most establish external communication protocols, links, and external exchange formats between proof assistants and computer algebra systems. Many depict those communications in the model of master-slave, where the master is the proof assistant and the slave is the computer algebra system. The work reported in this paper is novel in several aspects and defies assumptions from previous work.

First, we are considering a *strongly typed* computer algebra system. For concreteness, our work is done with OpenAxiom, a member of the AXIOM family. A distinctive feature of AXIOM is that every single computation is performed in a particular environment called *domain of computation*. A domain provides representations for values and operations for manipulating them. In AXIOM, and unlike in most popular computer algebra systems, a value is essentially useless and meaningless without knowledge of the intended domain — i.e. how to interpret that value. Domains, in turn, satisfy various specifications defined by *categories*. For obvious reasons, categories are organized into

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 15–29, 2011.

hierarchies, mirroring development of mathematical concepts and knowledge. In practice, domains and categories are huge data structures, usually with non-trivial cyclical dependencies. For example, the domain `Integer` of integer values satisfy the category `Ring` that specifies a ring structure over the common addition and multiplication operators. On the other hand, the specification of `Ring` involves the domain `Integer` — if anything, to specify the meaning of the expression $n \cdot x$ which denotes the addition of an arbitrary a ring element x to itself n times, and also the natural injection of integers into any ring structure. Furthermore, a simple computation such as `factor(x^2-2*x+1)` has the OpenAxiom system load no less than 19 domains or packages, not counting the hundreds of pre-loaded domains. Clearly, under these circumstances, devising a communication protocol for externalizing enough information to faithfully reconstruct the result of a computation, or check the validity of computational steps can be very inefficient in practice. Day-to-day experiences with dumping OpenAxiom domain and category databases or debug information suggest negative impacts on other support services such as garbage collection.

Second, this project — unlike most existing work — envisions a symbiotic coexistence of OpenAxiom and a Poly/ML-based proof assistant. That is, we envision a tight integration of OpenAxiom and, say, Isabelle/HOL where both systems run in the same address space with Isabelle/HOL calling on OpenAxiom for algebraic computation and OpenAxiom calling on Isabelle/HOL for validating computational steps or performing logical deduction. Achieving this tight integrating requires a formal account of both OpenAxiom program representation (Sec. 3) and the Poly/ML runtime system (Sec. 4), and also a formal correspondence between both (Sec. 5.) The challenge is compounded by the fact that neither the Spad programming language nor the Poly/ML abstract machine have a formal specification. A key contribution of this paper is a step toward formal specification of the Poly/ML abstract machine language and translation algorithms. We anticipate that these formal accounts will provide foundation for further scrutiny and other research projects such as an end-to-end verification of OpenAxiom libraries, or a complete mechanical verification of the Poly/ML system — the runtime system underlying many popular logical framework implementations.

Third, this project is also an excellent opportunity to clarify the semantics of the AXIOM family system. Because of historical implementation artifacts, it is sometimes mistakenly believed that AXIOM is a Lisp system. While it is true that AXIOM systems currently use Lisp systems as base runtime systems, they actually do have type-erasure semantics in the style of ML [7] — except for runtime queries of category satisfaction, but the semantics of those queries are essentially intensional in nature and are compilable with a type-erasure semantics. This project of retargeting OpenAxiom to Poly/ML runtime system clarifies that subtle but crucial aspect of AXIOM systems and offers a validation of that theory.

Finally, this work enables cross-cutting technology reuse. For example, retargeting OpenAxiom to Poly/ML makes available (at virtually no cost) the recent concurrency work [17] done in Poly/ML that has been so beneficial to the Isabelle framework to take advantage of multi-core machines. Furthermore, retargeting the Poly/ML platform opens the possibility of concurrent validation of OpenAxiom computations by several Poly/ML-based proof assistants.

2 The OpenAxiom and Poly/ML Systems

2.1 OpenAxiom

OpenAxiom [19] is an evolution of the AXIOM [13] computer algebra system. It is equipped with a strongly-typed programming language (named Spad) for writing large scale libraries, and a scripting language for interactive uses and for programming in the small. The Spad programming language features a two-level type system to support data abstraction and generic structures and generic algorithms.

Domain of computation. A central tenet of AXIOM philosophy is that computations occur in a given domain of discourse or *domain of computation*. For instances, the objects $(X^2 - 1)/(X - 1)$ and $X + 1$ are equal in $\mathbf{Q}(X)$, but not as functions from $\mathbf{R}$ to $\mathbf{R}$; see the excellent analysis of this intricate problem by Davenport [9] for further details.

Categories as specifications. Since several domains of computations may implement the same specification — *e.g.* both `Integer` and `String` implement equality comparison — the Spad programming language provides abstraction tools to write specifications as first class objects: *categories*. Here is a specification for domains of computations that implement equality comparison:

```
BasicType: Category == Type with
    =: (%,%) -> Boolean
    ~=: (%,%) -> Boolean
```

Categories can extend other categories, or may be parameterized, or can be composed out of previously defined categories. For example, a semi-group is an extension of `BasicType` with the requirement of an associative operator named `*`:

```
SemiGroup: Category == BasicType with
   *: (%,%) -> %
```

and a left-linear set over a semi-group `S` is a set that is stable by "multiplication" or dilation by values in `S`:

```
LeftLinearSet(S: SemiGroup): Category == Basictype with
    *: (S,%) -> %
```

We similarly define the notion of right-linear. A linear set over a semi-group `S` is a set that is both left-linear and right-linear over `S`:

```
LinearSet(S: SemiGroup): Category ==
   Join(LeftLinearSet S, RightLinearSet S)
```

More complex algebraic structures are specified using similar composition and extension techniques.

2.2 Poly/ML

Poly/ML is a system originally written by the second author while he was in the Computer Laboratory at Cambridge University. Poly/ML was initially developed as an experimental language, Poly, similar to ML but with a different type system. Among

the first users was Larry Paulson who used it to develop the Isabelle theorem prover. More recently David Matthews has continued to develop Poly/ML. The Standard Basis Library has been implemented and the compiler converted to the 1997 Definition of Standard ML (Revised). Poly/ML is available for the most popular architectures and operating systems. There are native code versions for the i386 (32 and 64 bit), Power PC and Sparc architectures. There is a byte-code interpreted version which can be used on unsupported architectures.

3 Spad Program Representation

The Spad programming language is strongly typed. Yet, it allows for runtime instantiation of domains and categories. Consequently, categories and domains are both compile-time and runtime objects. From now on, we will discuss only the representation of category objects. Domains and packages are similarly represented, with some variations to attend to data specific to domains.

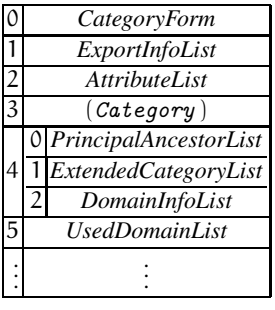

Fig. 1. Layout of category objects

Category object layout. A category object is represented [8] as a large heterogeneous tuple as shown in Fig. 1. Its components have the following meaning:

- slot 0 holds the canonical category form of the expression whose evaluation produces the category object under consideration
- slot 1 holds a list of function signatures exported by the category
- slot 2 holds a list attributes and the condition under which they hold
- slot 3 always contain the form (*Category*). It serves as a runtime type checking tag
- slot 4 contains three parts:
  - a list of principal ancestor category forms
  - a list of directly extended category forms
  - a list of domain explicitly used in that category
- slot 5 holds the list of all domain forms mentioned in the exported signatures.
- each of the slots 6 (and onwards) holds either a runtime information about a specific exported signature, or a pointer to domain object or a category object.

Because several existing OpenAxiom libraries assume the above layout, translation from Spad to the Poly/ML must preserve its observable behavior.

3.1 OIL: OpenAxiom Intermediate Language

Traditionally, AXIOM compilers translate Spad programs to Lisp, then compile the generated Lisp code. The OpenAxiom compiler has been modified to generate an intermediate representation that is independent of Lisp, to enable re-targeting to several backends. The intermediate language, called OIL, is a lambda calculus with constants and shown in Fig. 2.

Module	m	$::= \overrightarrow{d}$
Definition	d	$::= (\texttt{def}\ x\ e)$
expression	e,p	$::= v \mid l \mid [\overrightarrow{e}] \mid d \mid q \mid (f\ \overrightarrow{e}) \mid (\texttt{when}\ (\overrightarrow{w})) \mid (\texttt{bind}\ (\overrightarrow{(x\ e)})\ e)$ $\mid (\texttt{lambda}\ (\overrightarrow{x})\ e) \mid (\texttt{store}\ l\ e) \mid (\texttt{loop}\ (\overrightarrow{i})\ e) \mid (\texttt{seq}\ \overrightarrow{e})$
Location	l	$::= x \mid (\texttt{tref}\ x\ n)$
Branches	w	$::= (p\ e) \mid (\texttt{otherwise}\ e)$
Iterators	i	$::= (\texttt{step}\ x\ e\ e\ e) \mid (\texttt{while}\ p) \mid (\texttt{until}\ p) \mid (\texttt{suchthat}\ x\ e)$
Domain form	t	$::= x \mid (\texttt{D}\ \overrightarrow{t})$
Category form	c	$::= (\texttt{C}\ \overrightarrow{t}) \mid (\texttt{Join}\ \overrightarrow{c}) \mid (\texttt{mkCategory}\ k\ [\overrightarrow{[\sigma,q]}]\ [\overrightarrow{a}]\ [\overrightarrow{t}]\ \texttt{nil})$
Query	q	$::= b \mid (\texttt{hascat}\ t\ c) \mid (\texttt{hassig}\ t\ \sigma) \mid (\texttt{hasatt}\ t\ a)$ $\mid (\texttt{not}\ q) \mid (\texttt{and}\ q\ q) \mid (\texttt{or}\ q\ q)$
Constructor kind	k	$::= \texttt{category} \mid \texttt{domain} \mid \texttt{package}$
Signature	σ	$::= [x,t,\overrightarrow{t}]$
Builtin operator	o	$::= \texttt{eq} \mid \texttt{and} \mid \texttt{or} \mid \texttt{not} \mid \texttt{iadd} \mid \texttt{isub} \mid \ldots$
Function name	f	$::= o \mid \texttt{C} \mid \texttt{D} \mid x$
Literal values	v	$::= \texttt{nil} \mid b \mid n \mid s$
Boolean literal	b	$::= \texttt{false} \mid \texttt{true}$
Integer literal	n	
String literal	s	
Domain constructor	D	
Category constructor	C	
Attribute	a	
Identifier	x	

Fig. 2. OpenAxiom Intermediate Language

A module is a collection of top-level definitions. The body of a definition can be a lambda expression, a conditional expression, or a binding of local variables in an expression. We include some builtin operators (*e.g.*for addition on integers, category composition, etc.)

Example. Here is how the intermediate representation of the `BasicType` category from Sec. 2.1 looks:

```
(def BasicType;AL nil)

(def BasicType;
  (lambda ()
    (bind ((g (Join (Type)
```

```
                    (mkCategory domain
                        '(((= ((Boolean) $ $)) true)
                          ((~= ((Boolean) $ $)) true)
                          ((before? ((Boolean) $ $)) true))
                       nil '((Boolean)) nil))))
      (store (tref g 0) '(BasicType))
      g)))

(def BasicType
  (lambda ()
    (when ((not (eq BasicType;AL nil)) BasicType;AL)
          (otherwise (store BasicType;AL (BasicType;))))))
```

The compilation of a category typically produces tree top-level definitions: the definition of a cache holding instantiations of the category, the definition of a "worker" function that actually produces a category object for first-time instantiations, and a "wrapper" function around the worker function. The wrapper function is what the system invokes when a category is instantiated. In this example, *BasicType;AL* is the cache, *BasicType;* is the worker function, and *BasicType* is the wrapper function.

4 Poly/ML Codetrees

The front-end of the Poly/ML compiler performs syntax- and type-checking and produces an intermediate code in the form of a codetree. This untyped representation is machine-independent and, after optimization, is used to produce the machine-dependent code to execute. Our compilation process for Spad generates this code tree. The codetree is an ML data structure and does not have a canonical text representation. For the purposes of explanation the structure can be approximated by the grammar in Fig. 3.

Variables are given numerical names. For instance, a declaration $\mathrm{DECL}(\mathtt{k}, e)$ has a number $\mathtt{k}$ and a codetree term e. The codetree e denotes an expression to be bound to the variable $\mathtt{k}$. This can then be referenced within the rest of the containing block by means of a $\mathrm{LOCAL}(\mathtt{n}, \mathtt{k})$ codetree term. The number $\mathtt{k}$ corresponds to the identifier used in the declaration and $\mathtt{n}$ is the "nesting depth": zero if the reference is within the same function and non-zero if it is within an inner function. Function parameters are

Declaration	$\mathtt{d}$	$::= \mathrm{DECL}(\mathtt{k}, e)$
Code tree	e,$\mathtt{p}$	$::= \mathrm{NIL} \mid \mathtt{c} \mid \mathtt{d} \mid \mathrm{LIT}(\mathtt{n}) \mid \mathrm{ADDRESS}(\mathtt{a}) \mid \mathrm{BUILTIN}(\mathtt{r}) \mid \mathrm{EVAL}(e, [\overrightarrow{e}])$
		$\mid \mathrm{IF}(\mathtt{p}, e, e) \mid \mathrm{BLOCK}([\overrightarrow{e}]) \mid \mathrm{INDIRECT}(\mathtt{n}, e) \mid \mathrm{RECORD}([\overrightarrow{e}])$
		$\mid \mathrm{BEGINLOOP}([\overrightarrow{\mathtt{d}}], e) \mid \mathrm{LOOP}([\overrightarrow{e}]) \mid \mathrm{LAMBDA}(\mathtt{n}, e)$
Coordinates	$\mathtt{c}$	$::= \mathrm{PARM}(\mathtt{n}, \mathtt{k}) \mid \mathrm{LOCAL}(\mathtt{n}, \mathtt{k})$
Builtins	$\mathtt{r}$	$::= \mathtt{amul} \mid \mathtt{aminus} \mid \ldots$
Integer	$\mathtt{n}$, $\mathtt{k}$	
Identifier	$\mathtt{i}$	
Address	$\mathtt{a}$	

Fig. 3. Poly/ML codetree language

accessed using the PARAM(n, k) codetree term where k denotes the k-th parameter of the function declared n levels out to the current containing function. Tuples are created with the RECORD element and fields of a tuple are extracted with INDIRECT. LAMBDA introduces the body of a function. A BLOCK is a sequence of codetree terms and provides an environment for declarations. The result of evaluating a block is the value of the final expression. Generally, every code tree term except the last will be a DECL codetree term and these have scope over the rest of the block. It is possible to have other kinds of terms within a block that may be executed for their side-effects. Loops can be created using recursive functions or through use of BEGINLOOP and LOOP. BEGINLOOP represents the start of a loop and contains a list of DECL entries that represent loop index variables. The value for each DECL is the initial value of the loop variable. The expression part of the BEGINLOOP will almost always be a nested IF-expression with some of the branches containing LOOP-expressions. A LOOP expression causes a jump back to the BEGINLOOP with the loop variables updated with the values in the LOOP expression. The length of the argument list for a LOOP will always match the containing BEGINLOOP. Branches of the IF-expression that do not end with a LOOP result in exiting the LOOP. There is also no explicit code tree for representing function definition in Poly/ML. A function definition is expressed by a DECL codetree term whose second parameter is a lambda expression. Constants can be expressed as either LIT which represents a literal integer or ADDRESS which represents the address of some entity already present in the ML address space. Poly/ML is an incremental compiler and it is usual to compile an expression which makes reference to pre-existing entities.

As an example of code tree, the following factorial function in ML:

```
fun factorial n =
  if n = 0 then 1
  else n * factorial (n -1);
```

is compiled to the following codetree term:

```
DECL(1,
   LAMBDA(1,
      IF(EVAL(BUILTIN equala, [PARAM(0,1), LIT 0 ]),
         LIT 1,
         BLOCK[DECL(2,
             EVAL(LOCAL(1,1),
                [EVAL(BUILTIN aminus, [PARAM(0,1), LIT 1])])
             ),
             EVAL(BUILTIN amul, [PARAM(0,1), LOCAL(0,2)])
         ]
      )
   )
)
```

The ML function definition is translated to a declaration codetree term which is a DECL expression. The right hand side of the DECL expression is a LAMBDA codetree term representing the body of the function `factorial`. The body contains a conditional, an IF expression whose first argument is the test. If that succeeds the result is

the second argument, the literal value 1, and if it fails the third argument is executed. This consists of a BLOCK. The first entry in the block is a local declaration of a recursive call of `factorial`. The result of this is then multiplied by `n`. Several built-in functions are used. `equala` tests the equality of two integers, and `aminus` does integer subtraction.

5 Generating Poly/ML Codetree from OIL

The task of generating Poly/ML codetree from OIL starts with an OIL module m, a list of top-level declarations. The overall strategy is to translate that cluster of declarations into a Poly/ML codetree term that would eventually evaluate to a record value. Each component of that record is maintained in a one-to-one correspondence with an OIL top-level declaration through an environment of type

$$\text{Env} = [x \mapsto \mathrm{c}].$$

An environment Γ of type Env maps an OIL identifier x to a scope-and-position $\Gamma(x)$, which is either $\texttt{LOCAL}(\mathrm{n}, \mathrm{k})$ for variables or $\texttt{PARAM}(\mathrm{n}, \mathrm{k})$ for function parameters. For top-level declarations, n has value 0. Therefore the most important coordinate information for a top-level declaration is k, which is the slot number for x in the top-level RECORD codetree term.

Notation. In what follows, we use $[\![\bullet]\!]$ to enclose syntactic objects (be they OIL expressions or Poly/ML codetree terms). OIL expression objects are written in *`this font`* whereas Poly/ML codetree terms are typeset with `this other font`. When the meta variable x designates an object from a certain syntactic category, we use the notation $\widehat{\mathrm{x}}$ for a meta variable that holds a sequence of syntactic objects from that same category.

Modules. The entry point to the translation algorithm is the function

$$\mathcal{G} : [\text{Declaration}] \to \text{CodeTree} \times \text{Env}$$

which takes as input a list of OIL top-level declarations and produces a codetree-environment pair. The first component is usually a BLOCK codetree term containing all the codes generated for the top-level declarations and whose last term is a RECORD codetree term that constructs the value representation of the module:

$$\begin{aligned}
&\mathcal{G}(m) = \\
&\quad \textbf{\textit{let}}\ \left\langle \widehat{\mathrm{d}}, \Gamma \right\rangle = \mathcal{D}(m, [], 0, 0) \\
&\qquad \widehat{\mathrm{c}} = \mathcal{C}(\Gamma, 0) \qquad\qquad \text{-- } \textit{reference all toplevel declarations} \\
&\quad \textbf{\textit{in}}\ \left\langle \texttt{BLOCK}(\widehat{\mathrm{d}} + +[\texttt{RECORD}(\widehat{\mathrm{c}})]), \Gamma \right\rangle
\end{aligned}$$

The second component is the resulting environment. The purpose of the the function $\mathcal{D}$

$$\mathcal{D} : [\text{Declaration}] \times \text{Env} \times \text{Integer} \times \text{Integer} \to [\text{CodeTree}] \times \text{Env}$$

is to translate a cluster of declarations in a given initial environment, a scope nesting level, an initial declaration number. It returns a pair of a list of Poly/ML codetree terms for the declarations and an updated environment. The details of code generation for declarations are the subject of the next section. The function $\mathcal{C}$ take an environment Γ, a scope nesting level n, and returns the list of all coordinates in Γ with scope nesting level n.

Definitions. Generating Poly/ML code for an OIL definition is straightforward. We allocate LOCAL coordinates for corresponding Poly/ML entity, and generate codes for the initializer in an environment where the name of the entity being defined is bound to its coordinates

$$
\begin{array}{l}
\mathcal{D}([], \Gamma, n, k) = \langle [], \Gamma \rangle \\
\mathcal{D}\big((\texttt{def}\ x\ e) :: \widehat{d}, \Gamma, n, k\big) = \\
\quad \textbf{\textit{let}}\ \ \Gamma_1 = \Gamma\ ++\ [x \mapsto \texttt{LOCAL}(n,k)] \quad -\!-\ \textit{introduce } x \textit{ in the scope of its initializer} \\
\qquad \langle e, \Gamma_2, k_1 \rangle = \mathcal{E}(e, \Gamma_1, n, k+1) \\
\qquad \left\langle \widehat{d}, \Gamma_3 \right\rangle = \mathcal{D}\left(\widehat{d}, \Gamma_1, n, k_1 + 1\right) \\
\quad \textbf{\textit{in}}\ \left\langle \texttt{DECL}(k, e) :: \widehat{d}, \Gamma_3 \right\rangle
\end{array}
$$

That code generation strategy implements unrestricted value recursion, and in particular recursive functions. If we wanted to restrict recursion to functions only, we could delay to the actual binding the expression translation function $\mathcal{E}$. The function $\mathcal{E}$ with functionality

$$\mathcal{E} : \text{Expression} \times \text{Env} \times \text{Integer} \times \text{Integer} \to \text{CodeTree} \times \text{Env} \times \text{Integer}$$

takes an OIL expression, an initial environment, a scope nesting level, a variable position, and produces a triple that consists of the Poly/ML codetree for the OIL expression, an updated environment, and the next available variable position at the same scope. This function $\mathcal{E}$ (which accepts any OIL expressions) is never called directly to generate codetree for declarations. Its details are the subject of the next paragraphs.

Functional abstraction. Given a functional abstraction of arity k at nesting level n, we augment the enclosing environment G with k PARAM coordinates for the parameters, and then generate code for the body with nesting level increased by one:

$$
\begin{array}{l}
\mathcal{E}([\![(\texttt{lambda}\ (x_1\ \ldots x_k)\ e)]\!], \Gamma, n, k') = \\
\quad \textbf{\textit{let}}\ \ \Gamma_1 = \mathcal{P}(\Gamma, [x_1, \ldots, x_k], n+1) \quad -\!-\ \textit{increase the nesting level for the body} \\
\qquad \langle e, \Gamma_2, k'' \rangle = \mathcal{E}(e, \Gamma_1, n+1, 0) \\
\quad \textbf{\textit{in}}\ \langle \texttt{LAMBDA}(k, e), \Gamma, k' \rangle
\end{array}
$$

Finally we return a LAMBDA codetree term, the original environment and the original variable position number. Allocation of coordinates for the function parameters is done via the helper function

$$\mathcal{P} : \text{Gamma} \times [\text{Identifier}] \times \text{Integer} \to \text{Env}$$

defined by:

$$
\begin{array}{l}
\mathcal{P}(\Gamma, [], n, k) = \Gamma \\
\mathcal{P}(\Gamma, x :: \widehat{x}, n, k) = \Gamma\ ++\ [x \mapsto \texttt{PARAM}(n,k)]\ ++\ \mathcal{P}(\Gamma, \widehat{x}, n, k+1)
\end{array}
$$

Note that translation of `lambda`-expression is the only place where we increase the "nesting level" of codetree coordinates.

Local declarations. Code generation for local declarations is quite similar to that of functional abstraction:

$$
\begin{array}{l}
\mathcal{E}(\llbracket(\texttt{bind}((x_1\ e_1)\ldots(x_k\ e_k))\ e)\rrbracket, \Gamma, n, k') = \\
\quad \textbf{\textit{let}}\ \langle e_1, \Gamma_1, k_1'\rangle = \mathcal{E}(e_1, \Gamma, n, k') \\
\qquad \langle e_2, \Gamma_2, k_2'\rangle = \mathcal{E}(e_2, \Gamma_1 + +[x_1 \mapsto \textsc{local}(n, k_1')], n, k_1' + 1) \\
\qquad \ldots \\
\qquad \langle e_k, \Gamma_k, k_k'\rangle = \mathcal{E}(e_k, \Gamma_{k-1} + +[x_{k-1} \mapsto \textsc{local}(n, k_{k-1}')], n, k_{k-1}' + 1) \\
\qquad \langle e, \Gamma', k''\rangle = \mathcal{E}(e_k, \Gamma_k + +[x_k \mapsto \textsc{local}(n, k_k')], n, k_k' + 1) \\
\quad \textbf{\textit{in}}\ \langle \textsc{block}([\textsc{decl}(k_1', e_1), \ldots, \textsc{decl}(k_k', e_k), e]), \Gamma, k''\rangle
\end{array}
$$

Conditional expressions. The simplest conditional expression is the equivalent of an `if-then` expression without an alternative part:

$$
\begin{array}{l}
\mathcal{E}(\llbracket(\texttt{when}\ ((p_1\ e_1)))\rrbracket, \Gamma, n, k) = \\
\quad \textbf{\textit{let}}\ \langle p_1, \Gamma_1, k_1\rangle = \mathcal{E}(p_1, \Gamma, n, k) \\
\qquad \langle e_1, \Gamma_2, k_2\rangle = \mathcal{E}(e_1, \Gamma_1, n, k_1) \\
\quad \textbf{\textit{in}}\ \langle \textsc{if}(p_1, e_1, \textsc{nil}), \Gamma_2, k_2\rangle
\end{array}
$$

As a special-case, for code-generation purpose, we accept `otherwise` as a predicate with the following meaning:

$$
\begin{array}{l}
\mathcal{E}(\llbracket(\texttt{when}\ ((\texttt{otherwise}\ e)))\rrbracket, \Gamma, n, k) = \\
\quad \mathcal{E}(e, \Gamma, n, k)
\end{array}
$$

The most general form of a conditional expression in OIL is a multiway branch, with an optional default case (the `otherwise`-branch) at the end of list of choices. This form corresponds to a series of nested traditional `if-then-else` expressions:

$$
\begin{array}{l}
\mathcal{E}(\llbracket(\texttt{when}\ (p\ e) :: \widehat{w})\rrbracket, \Gamma, n, k) = \\
\quad \textbf{\textit{let}}\ \langle p, \Gamma_1, k_1\rangle = \mathcal{E}(p, \Gamma, n, k) \\
\qquad \langle e_1, \Gamma_2, k_2\rangle = \mathcal{E}(e, \Gamma_1, n, k_1) \\
\qquad \langle e_2, \Gamma_3, k_3\rangle = \mathcal{E}(\llbracket(\texttt{when}\ \widehat{w})\rrbracket, \Gamma_2, n, k_2) \\
\quad \textbf{\textit{in}}\ \langle \textsc{if}(p_1, e_1, e_2), \Gamma_3, k_3\rangle
\end{array}
$$

Loops. The simplest looping structure in OpenAxiom is the infinite loop represented in OIL as `(loop () e)` where `e` is the expression to be evaluated indefinitely. This basic structure can be controlled by iterators. From a control structure point of view, an iterator is semantically a 4-tuple $\left\langle \widehat{d}, \widehat{e}, p_1, p_2 \right\rangle$ that controls a loop:

1. a sequence $\widehat{d}$ of declarations (and initializations) of variables with lifetime spanning exactly the entire execution of the loop
2. a sequence of expressions $\widehat{e}$ giving values to the variables in the first component for the next attempt at the loop body iteration
3. a filter predicate expression p_1 which controls the evaluation of the body for a particular evaluation of the loop body
4. a continuation predicate p_2 which, if false, terminates the loop

We use the translation function $\mathcal{I}$

$$\mathcal{I} : \text{Control} \times [\text{Iterator}] \times \text{Env} \times \text{Integer} \times \text{Integer} \rightarrow \text{Control} \times \text{Env} \times \text{Integer}$$

as a helper.

$$\mathcal{I}(x, [], \Gamma, n, k) = \langle x, \Gamma, k\rangle$$

$$\mathcal{I}\left(\left\langle \widehat{d}, \widehat{e}, p_1, p_2\right\rangle, [\![(\texttt{while}\ p)]\!] :: \widehat{i}, \Gamma, n, k\right) =$$
$$\textit{let}\ \langle p, \Gamma, k'\rangle = \mathcal{E}(p, \Gamma, n, k)$$
$$p_2' = \text{EVAL}(\text{BUILTIN and}, [p_2, p])$$
$$\textit{in}\ \mathcal{I}\left(\left\langle \widehat{d}, \widehat{e}, p_1, p_2'\right\rangle, \widehat{i}, \Gamma, n, k'\right)$$

$$\mathcal{I}\left(\left\langle \widehat{d}, \widehat{e}, p_1, p_2\right\rangle, [\![(\texttt{until}\ p)]\!] :: i, \Gamma, n, k\right) =$$
$$\textit{let}\ \langle p, \Gamma, k'\rangle = \mathcal{E}(p, \Gamma, n, k)$$
$$p_2' = \text{EVAL}(\text{BUILTIN and}, [p_2, \text{EVAL}\,(\text{BUILTIN not}, [p])])$$
$$\textit{in}\ \mathcal{I}\left(\left\langle \widehat{d}, \widehat{e}, p_1, p_2'\right\rangle, \widehat{i}, \Gamma, n, k'\right)$$

$$\mathcal{I}\left(\left\langle \widehat{d}, \widehat{e}, p_1, p_2\right\rangle, [\![(\texttt{suchthat}\ p)]\!] :: \widehat{i}, \Gamma, n, k\right) =$$
$$\textit{let}\ \langle p, \Gamma, k'\rangle = \mathcal{E}(p, \Gamma, n, k)$$
$$p_1' = \text{EVAL}(\text{BUILTIN and}, [p_1, p])$$
$$\textit{in}\ \mathcal{I}\left(\left\langle \widehat{d}, \widehat{e}, p_1', p_2\right\rangle, \widehat{i}, \Gamma, n, k'\right)$$

$$\mathcal{I}\left(\left\langle \widehat{d}, \widehat{e}, p_1, p_2\right\rangle, [\![(\texttt{step}\ x\ e_1\ e_2\ e_3)]\!] :: \widehat{i}, \Gamma, n, k\right) =$$
$$\textit{let}\ \langle e_1, \Gamma_1, k_1\rangle = \mathcal{E}(e_1, \Gamma, n, k+1) \quad -- \text{k } \textit{is for the loop variable}$$
$$\langle e_2, \Gamma_2, k_2\rangle = \mathcal{E}(e_1, \Gamma_1, n, k_1)$$
$$\langle e_3, \Gamma_3, k_3\rangle = \mathcal{E}(e_2, \Gamma_2, n, k_2+1) \ -- k_2 \textit{ is used for holding the value of } e_2$$
$$p = \text{EVAL}(\text{BUILTIN int\_lss}, [x, \text{LOCAL}\,(n, k_2)])$$
$$e = \text{EVAL}(\text{BUILTIN aplus}, [\text{LOCAL}\,(n, k), \text{LOCAL}\,(n, k_3+1)])$$
$$\Gamma' = \Gamma_3 + +[x \mapsto \text{LOCAL}\,(n, k)]$$
$$\widehat{d}' = [\text{DECL}\,(k, e_1), \text{DECL}\,(k_2, e_2), \text{DECL}\,(k_3+1, e_3)]$$
$$\widehat{e}' = [e, \text{LOCAL}\,(n, k_2), \text{LOCAL}\,(n, k_3+1)]$$
$$x = \left\langle \widehat{d} + +\widehat{d}', \widehat{e} + +\widehat{e}', \text{NIL}, p\right\rangle$$
$$\textit{in}\ \mathcal{I}(x, i, \Gamma', k_3+2)$$

The translation of an OIL loop expression to a Poly/ML codetree term first translates the iterators to control terms, then uses the resulting environment to translate the body of the loop::

$$\mathcal{E}\left([\![(\texttt{loop}\ \widehat{i}\ e)]\!], \Gamma, n, k\right) =$$
$$\textit{let}\ \left\langle\left\langle \widehat{d}, \widehat{e}, p_1, p_2\right\rangle, \Gamma', k'\right\rangle = \mathcal{I}\left(\langle [], \text{NIL}, \text{NIL}, \text{NIL}\rangle, \widehat{i}, \Gamma, n, k\right)$$
$$\langle e, \Gamma'', k''\rangle = \mathcal{E}(e, \Gamma', n, k')$$
$$\textit{in}\ \left\langle \text{BEGINLOOP}\left(\widehat{d}, \text{IF}\,(p_2, \text{BLOCK}\,[\text{IF}\,(p_1, e, \text{NIL}), \text{LOOP}\,\widehat{e}], \text{NIL})\right), \Gamma'', k''\right\rangle$$

Sequence of expressions. A sequence of OIL expressions corresponds to a BLOCK codetree term.

$$\mathcal{E}([\![(\texttt{seq}\ e_1 \ldots e_{n'})]\!], \Gamma, n, k) =$$
$$\textit{let}\ \langle e_1\Gamma_1, k_1\rangle = \mathcal{E}(e_1, \Gamma, n, k)$$

$$
\begin{aligned}
&\ldots\\
&\langle e_{n'}, \Gamma_{n'}, k_{n'}\rangle = \mathcal{E}(e_{n'}, \Gamma_{n'-1}, n, k_{n'-1})\\
\textbf{\textit{in}}\ &\langle \mathrm{BLOCK}\,[e_1, \ldots, e_{n'}], \Gamma_{n'}, k_{n'}\rangle
\end{aligned}
$$

Function calls. An OIL function call expression generally corresponds to an `EVAL` codetree term. The operation may be a builtin operation, a "variable", or a global function (corresponding to a constructor instantiation), or a special runtime function (*e.g.* for `mkCategory` or `Join`) Arguments are evaluated from left to right.

6 Implementation

The code generation component is implemented as an OpenAxiom library. First we compile a Spad program to the OIL intermediate representation. That representation is then translated to Poly/ML codetree terms, written into a file. Note that, the codetree format written to disk is slightly different from the pretty printed version of the Poly/ML compiler. This is done so to make parsing easier, and hide redundant information. One caveat is that the current rule is implemented for category definition without default implementation. Currently the function `Join` only merges specifications from different category objects.

7 Example

Below is the translation of the OIL representation of the category `BasicType`:

```
DECL(1,
 LAMBDA(0,
 BLOCK[
  DECL(2,
   EVAL(ADDRESS Join,
     [
     RECORD[
        EVAL(
          LAMBDA(0,
            BLOCK[
              DECL(2,
                EVAL(ADDRESS Join,
                [
                  RECORD[
                     EVAL(ADDRESS mkCategory, [ADDRESS ?]),
                     LIT 0
                  ]
                ])
              ),
              DECL(3,
                EVAL(ADDRESS setName,
                  [
                    RECORD[LOCAL(0,2), ADDRESS ?]
```

```
                ]
              )
            ),
            LOCAL(0, 3)
          ]
        ),
        [LIT 0]
      ),
      RECORD[EVAL(ADDRESS mkCategory, [ADDRESS ?]), LIT 0]
    ]
    ]
  )
 ),

 DECL(3,
   EVAL(
     ADDRESS setName ,
     RECORD[LOCAL(0,2), ADDRESS ?]
   )
  ),

 LOCAL(0,3)
 ]
)
)
```

8 Related Work

IR based code generations in CAS. The Aldor [21] language compiler defines a first order abstract machine (FOAM) which is an intermediate language for representing Aldor code at a lower level [22]. FOAM is platform independent. However, we have never been able to generate a working FOAM out of AXIOM systems. One of the FOAM's goal is to define data structures for program transformation, optimization at FOAM level, as well as for generating C and Common Lisp code for Aldor programs. Maple [16] uses *inert* expressions to represent a Maple program. The internal representation supports code generation for other languages such as C, Java and Fortran, as well as optimization functionalities provided by Maple's CodeGeneration package. The inert forms closely reflect the Maple internal DAG data structure representation [18].

Interfacing CAS and deduction systems. Various methods for interfacing CAS and deduction systems have been extensively discussed in the literature. In the work of Ballarin and Homann [3], Maple was used as a term rewriting system for enhancing the expression simplifier of Isabelle. Interfaces are provided in Isabelle's ML environment for starting and exiting a Maple session, sending expressions to Maple for evaluation, and receiving results from Maple. The communication at lower level is through a Unix pipe between Maple and Isabelle processes. High-level syntax translation rules are defined to achieve translations between expressions in Isabelle syntax and their equivalents in Maple's syntax. The work of Calmet and Homann [5] as well as the work of

Barendregt and Cohen [4] suggest the use of a separate language such as OpenMath [1] to implement communication protocols between different computation and deduction systems. Harrison and Théry combined HOL and Maple to verify results computed by Maple [11]. The communication is based on the idea of a "software bus". Translation between HOL and Maple terms is implemented as a third party between the two systems. The work of Adams and Dunstan [2] integrated Maple with the automated theorem prover PVS to validate computations in a real analysis library. Maple is extended with external C functions through Maple's foreign function interface. The C functions manage the communications with PVS and high-level syntax translations between different syntax. Recently, Delahaye and Mayero demonstrated a speed-up of the field tactics in Coq with Maple's field computation [10]. Algebraic expressions over a field in Coq are translated to the expressions in Maple's syntax, and sent to Maple for computation. The results are sent back to Coq and translated back into Coq's syntax.

Our approach differs from all the approaches mentioned above in several aspects. The existing approaches rely mainly on defining: (1) process communication protocols between CAS and deduction systems and (2) high-level program syntax translations. This results in the CAS and the deduction system each having each their own address space. Our work generates the same lower level run time instructions from programs with different syntax. OpenAxiom programs and ML programs will share the same address space. Translation rules in our work are defined between intermediate representations instead of high-level syntax. Furthermore, improvements to the Poly/ML system (such as recent addition of efficient and scalable concurrency primitives) are directly available to both OpenAxiom and Poly/ML-based engines. We consider this aspect to be of a significant benefit. It will also help enhance and improve recent implicit parallelization capabilities [14,15] added to OpenAxiom.

9 Conclusion and Future Work

This paper is a report on a work-in-progress. Obviously, much remains to be done for runtime domain instantiations. In the long-term, we would like to obtain a complete formal specification of the Spad programming language, including its intermediate representation. Ideally, we would like a formally checked translation. But, that is a much harder task that will span several years. In the near future, after the complete re-targeting of OpenAxiom, we plan to investigate various connections between several Poly/ML based logical systems (Isabelle in particular) and the OpenAxiom platform as completion of our initial goal.

Acknowledgements. This work was partially supported by NSF grant CCF-1035058.

References

1. Abbott, J., Díaz, A., Sutor, R.S.: A report on openmath: a protocol for the exchange of mathematical information. SIGSAM Bull. 30, 21–24 (1996)
2. Adams, A., Dunstan, M., Gottliebsen, H., Kelsey, T., Martin, U., Owre, S.: Computer algebra meets automated theorem proving: Integrating maple and pvs. In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, pp. 27–42. Springer, Heidelberg (2001)

3. Ballarin, C., Homann, K., Calmet, J.: Theorems and algorithms: an interface between isabelle and maple. In: Proceedings of the 1995 international symposium on Symbolic and algebraic computation, ISSAC 1995, pp. 150–157. ACM, New York (1995)
4. Barendregt, H., Cohen, A.M.: Electronic communication of mathematics and the interaction of computer algebra systems and proof assistants. J. Symb. Comput. 32, 3–22 (2001)
5. Calmet, J., Homann, K.: Classification of communication and cooperation mechanisms for logical and symbolic computation systems. In: Frontiers of Combining Systems (FroCos), pp. 221–234 (1996)
6. Carette, J., Farmer, W.M., Wajs, J.: Trustable communication between mathematics systems. In: Proc. of Calculemus 2003, Aracne, pp. 58–68 (2003)
7. Crary, K., Weirich, S., Morrisett, G.: Intensional Polymorphism in Type-erasure Semantics. J. Funct. Program. 12, 567–600 (2002)
8. Davenport, J.H.: A New Algebra System. Technical report, IBM Research (1984)
9. Davenport, J.H.: Equality in Computer Algebra and Beyond. J. Symb. Comput. 34, 259–270 (2002)
10. Delahaye, D., Mayero, M.: Dealing with algebraic expressions over a field in coq using maple. J. Symb. Comput. 39, 569–592 (2005)
11. Harrison, J., Théry, L.: A skeptic's approach to combining hol and maple. J. Autom. Reason. 21, 279–294 (1998)
12. Isabelle (2011), http://www.cl.cam.ac.uk/research/hvg/Isabelle/index.html
13. Jenks, R.D., Sutor, R.S.: AXIOM: The Scientific Computation System. Springer, Heidelberg (1992)
14. Li, Y., Reis, G.D.: A Quantitative Study of Reductions in Algebraic Libraries. In: PASCO 2010: Proceedings of the 4th International Workshop on Parallel and Symbolic Computation, pp. 98–104. ACM, New York (2010)
15. Li, Y., Reis, G.D.: An Automatic Parallelization Framework for Algebraic Computation Systems. In: ISSAC 2011: Proceedings of the International Symposium on Symbolic and Algebraic Computation. ACM, New York (2011)
16. Maple. Maplesoft Inc., Canada (2011), http://www.maplesoft.com
17. Matthews, D.C.J., Wenzel, M.: Efficient Parallel Programming in Poly/ML and Isabelle/ML. In: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming, DAMP 2010, pp. 53–62. ACM, New York (2010)
18. Monagan, M.B., Geddes, K.O., Heal, K.M., Labahn, G., Vorkoetter, S.M., McCarron, J., De Marco, P.: Maple Advanced Programming Guide, Maplesoft, Canada (2007)
19. OpenAxiom (2011), http://www.open-axiom.org
20. Poly/ML (2011), http://www.polyml.org
21. Watt, S.M.: Aldor Programming Language (December 2009), http://www.aldor.org
22. Watt, S.M., Broadbery, P.A., Iglio, P., Morrison, S.C., Steinbach, J.M.: Foam: A first order abstract machine version 0.35. Technical report, IBM Thomas J. Watson Research Center (2001)