Local Specialization in Open-Axiom

Jacob Smith  Yue Li  Yuriy Solodkyy  Gabriel Dos Reis  Jaakko Järvi

September 1, 2009

Abstract

Most algebraic structures in computational mathematics are instances of functors. Functoriality is expressed through parametrized interfaces and datatypes in typeful programming languages. Occasionally, properties of such functor instances depend on aspects of parameters. For instance, the set of integers \( \mathbb{Z}/k\mathbb{Z} \) modulo a positive natural number \( k \) uniformly carries a ring structure for all values of \( k \). When \( k \) is a prime number, \( \mathbb{Z}/k\mathbb{Z} \) also carries a field structure. In the generic programming setting expressing such dependent properties at a single definition point is called local specialization. Several categorial computer algebra systems support local specializations to varying degrees. This paper presents a principled compilation strategy for local specialization supported by the theory of abstract interpretation. We illustrate the correspondence between the concrete and abstract semantics of categories (where local specialization exists) by translating to a concrete internal language, and defining an abstract execution for categories. We implement the framework in the OpenAxiom computer algebra system, and demonstrate the abstract interpretation through real examples from OpenAxiom’s standard algebras.

This report is a technical report accompanying the paper “Supporting Local Specialization in a Categorial Computer Algebra System by Abstract Interpretation”. The purpose of this technical report is to expand on the description of the translation and abstract-interpretation rules presented in the submitted paper. Briefly, the goal of the submitted paper are to provide:

- a specification of the operational semantics of categories and has-predicates in the AXIOM system family;
- to provide a generalization of the set of legal predicates in conditional specifications, and to allow user-defined predicates; and
- to provide a formalization and implementation of an abstract interpretation framework for deriving the static semantics of categories.

1 An Overview of Axiom

OpenAxiom is a library-centric extensible system. Users extend it by writing libraries using the programming language called Spad. We describe enough of the Spad syntax to enable a discussion of Spad categories and domains. We then specify the concrete semantics of Spad categories by defining a translation from Spad to an internal language, followed by the specification of the operational semantics of this internal language.

The general philosophy is based on abstract datatype methodology. A concrete algebraic structure is introduced by its specification (which we call its category) and its implementation (called its domain). As a rule, specifications tend to have many implementations. Consequently, it is typical to define categories indenpendently of their implementations. For example, we can capture the general notion of monoid structure by requiring a binary operation named \( * \) along with a constant named \( 1 \). In Spad, that is expressed as:

\[
\text{Monoid(): Category == Type with}
\]

\[
*: (%,%) \to %
\]

\[
1: %
\]

The datatype \texttt{Integer} obviously satisfies the \texttt{Monoid} specification. So does the list data structure along with the concatenation operation and the empty list. In the OpenAxiom system, datatypes satisfies (to belong to) categories by explicit assertions. And those assertions are made as part of functor definitions. Here is the definition of a functor that provides a monoid view of list data structures.

\footnote{Submitted to VMCAI '10}
ListMonoid(T: Type): Monoid() with
c coerce: List T → %
== add
  import List T
  Rep == List T
c coerce(l: List T) == per l
1:% == per empty()
(x:%) * (y:%) == per concat(rep x, rep y)

This definition says that ListMonoid is a functor taking any datatype T to a Monoid structure (refers to by %) with an operation that implicitly converts a List T to the new datatype. The implementation of that specification is what comes after the == sign. It starts by making locally available operations from List T. The line Rep == List T says that values of the new datatype are internally List Ts. All other lines are just implementations of the operations specified in the interface. The operation per takes a value of type Rep and blesses into a value of the new datatype (%). The operation rep does the inverse. These two operations generate no runtime code. They exist purely as compile time artifacts acting as abstract barrier between a new datatype and its underlying representation.

Below, we describe enough of the syntax of Spad to enable a discussion of Spad categories. We then specify the concrete semantics of Spad categories by defining a translation from Spad to an internal language, followed by the specification of the operational semantics of this internal language.

Program
A Spad program can be understood as a sequence of category and domain definitions, followed by an expression.

CategoryDef
A Spad category definition is a sequence of specifications. A specification can be either a function declaration or category extension. All Spad categories must extend the category Type, which is a root category with no specifications. For example, below, the left-hand category definition declares a Spad category with two signatures: a neutral element 1 and binary operator * of a monoid structure. The right-hand definition extends Monoid with the inverse operation to capture the mathematical notion of group structure.

Export
An export is a specification which can be either a function declaration, or a category extension.

Exports
A conditional specification led by an if-statement with predicate π. In each of its branches there is an Export, meaning a specification is decided at run-time from the evaluation of the predicate π.
**Predicate**  A predicate is used as the condition of a `Exports`. It can be either a atomic predicate such as `true`, a has-predicate, or a logical formulae of atomic predicates.

**Type**  A type is an instantiation of a Spad category or Spad domain, a function type, the name of a Spad category or Spad domain, or a carrier set denoted by `%`. In a category definition, `%` denotes a placeholder for a domain. In a domain definition, `%` denotes the domain itself.

**DomainDef**  A Spad domain definition provides implementations for the specifications by categories. Then implementation part is called `Capsule`, which may define the representation of the object belonging to the domain, and provide definitions for operations declared in the `Exports` of the categories it implements. For example, the category definition

```spad
ListMonoid(T: SetCategory()): Monoid() == add
  import List T -- brings List into scope
  Rep == List T
  construct(l: List T) == per l
  1:% == per empty()
  (x:%) * (y:%) == per concat(rep x, rep y)
```

implements a version of the `Monoid` specification. The neutral element `1` of `Monoid` is implemented by the empty list constructed by `empty()`, the binary operator `*` is implemented by a list concatenation.

**Capsule**  The implementation part of a Spad domain is its capsule. A capsule may specify the representation of a domain, and specifies a sequence of toplevel definitions for operations on the Spad domain objects. Note that a Spad domain specifies the underlying representation of its objects by assigning a type expression to the identifier `Rep`. A representation can occur only in a Spad domain definition.

**Definition**  A definition is the binding of an identifier or a function call expression to a Spad category, Spad domain, or an ordinary function. The Spad language, as understood by the Spad compiler, does not allow ordinary function definitions at toplevel.

**CallForm**  A call form consists of an identifier and a parenthesized sequence of signatures declaring formal parameters. A call form is needed in the definitions of a Spad category, Spad domain, and function.

**Expression**  An expression is either an if-statement, function call, or a constant.

**Identifier**  An identifier is a finite sequence of characters. The set of identifiers in Spad is countably infinite.

## 2 Internal Language

We define the semantics of a Spad program by translation into an internal representation language, aimed to be as simple as possible, yet allowing us to express categories and their specifications therein. Figures 2 and Figure 3 show the syntax and operational semantics, respectively, for this language.

![Figure 2](image-url)  The syntax of the internal language.

The internal language is the untyped lambda calculus with some extensions. We provide support for a list data structure, offering the primitive list operators `cons^2`, `car`, and `cdr`, as well as a predicate for detecting an empty list.

\[\begin{array}{ll}
\text{Terms} & e ::= \lambda x.e \mid e_1 \cdot e_2 \mid \text{cons}(c_1, e_2) \mid \text{car}(e) \mid \text{cdr}(e) \mid \text{isNil?}(e) \\
& \quad \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid \text{let } x = e_1 \text{ in } e_2 \mid R_{\text{id}}(c_1, e_2) \mid R_{\text{id}}(e_1, e_2) \\
& \quad \mid \pi_1(e) \mid \pi_2(e) \mid \text{isNil?}(e) \mid \text{isId?}(e) \mid v \\
\text{Values} & v ::= e \mid x \mid \lambda x.e \mid \text{cons}(v_1, v_2) \mid \text{nil} \mid \text{true} \mid \text{false} \\
\end{array}\]

\(^2\)In the later sections, we use the notation \(\llbracket a_1, \ldots, a_n \rrbracket\) as a shorthand for constructing a list from the elements \(a_1, \ldots, a_n\).
The term \( R_{id}(c_1, c_2) \) represents a Spad domain or category, where the constant \( c_1 \) is the name of the domain or category. The variable \( e_2 \) is a list of representations of the specifications, i.e., the function declarations and category extensions, of the domain or category. The term \( R_{\_\_}(c_1, c_2) \) represents a Spad function definition where \( c_1 \) is a list of the representations of the parameter types of the function, and \( e_2 \) is a lambda abstraction representing the body of the function. To access the elements of the above terms we define the functions \( \pi_1 \) and \( \pi_2 \) that return the first and second element, respectively, when applied to either of the “R-terms”. The constants \( c \) include OpenAxiom’s set of primitive types (e.g., integers and floating-point numbers) and symbols (used, e.g., as names of categories and domains).

\[
\begin{align*}
(\lambda x.e)v & \mapsto e[v/x] & e_1 & \mapsto e'_1 & e & \mapsto e' \\
\text{cons}(e_1, e_2) & \mapsto \text{cons}(e'_1, e_2) & \text{cons}(v, e) & \mapsto \text{cons}(v, e') & e & \mapsto e' \\
car(e) & \mapsto car(e') & e & \mapsto e' & cdr(e) & \mapsto cdr(e') \\
\text{isNil}(e) & \mapsto \text{isNil}(e') & e & \mapsto nil & e & \mapsto v \neq \text{nil} \\
\text{let } x = v \text{ in } e & \mapsto e[v/x] & \text{let } x = e_1 \text{ in } e_2 & \mapsto e'_1 & \text{let } x = e_1 \text{ in } e_2 & \mapsto e'_1 \\
\text{if } \text{true} \text{ then } e_2 \text{ else } e_3 & \mapsto e'_2 & \text{if } \text{false} \text{ then } e_2 \text{ else } e_3 & \mapsto e'_3 & \text{if } e_1 \text{ then } e_2 \text{ else } e_3 & \mapsto e'_1 \\
\pi_i(R_*(c_1, e_2)) & \mapsto e'_i & e & \mapsto R_*(e_1, e_2) & v & \neq R_*(v_1, v_2) \\
\text{is}\_\_?(e) & \mapsto \text{true} & \text{is}\_\_?(v) & \mapsto \text{false}
\end{align*}
\]

**Figure 3:** The small step operational semantics of the internal language. The • symbol stands for \( id \) or \( \rightarrow \). The index \( i \) in the projection rule is either 1 or 2.

### 3 Translation to IL

As the basis for the abstract interpretation discussed in Section 2, we describe the translation of a Spad category into the internal language. Formally, we define a syntax-directed translation function \( T_\bullet[] : \text{SPAD} \rightarrow \text{IL} \), where SPAD is the set of Spad programs and IL the set of programs in the internal language. This function, defined by case for each syntactical form of Spad, is shown in Figure 4.

To translate a category definition to the internal language, we iteratively translate a conditional specification at each program point of the category definition, in addition to the callform of the category. (The callform is an identifier and a parenthesized list of formal parameters of a category.) Each iteration step computes the internal representation of a callform, a function declaration, or recursively translates a category extension following the transitivity of category extension. That is, when a category \( C_1 \) declares that it extends the category \( C_2 \), we replace this specification with the specifications of \( C_2 \), a recursive operation.

The translation of a category definition results in a lambda abstraction in IL introduced by a let-statement which names the lambda abstraction. The lambda abstraction takes an extra parameter \( % \) for the carrier set \( % \) in SPAD. The body of the abstraction is a list of if-statements (whose branches contain “R-terms”) representing a sequence of specifications. The if-statements of the conditional specifications in the body of a category are translated to if-statements in IL; has-predicates are translated to the function \( \in \); category extensions are translated to function calls to other translated categories in IL; the Spad boolean operators and, or, and not are translated to functions of the same name in the IL; function declarations are translated to “R-_,-terms”.

4
let $\oplus = \lambda \text{left}. \text{right}$.

if $\text{isnil? left}$ then right else $\text{cons(car left, fix(concat)(cdr left, right))}$ in

let $\in$ = $\lambda \text{left}. \alpha$.speci

if $\text{isnil? left}$ then $\text{false}$ else $\text{if member_equal?(car speci, value) then true else}$

$\text{fix}(\in)(\text{value, cdr speci})$ in
As an example, we translate the category `ComplexCategory`, defined in Figure 5, which represents the extension of a ring structure by $\sqrt{-1}$. Applying the translation rules at the topmost level yields (showing only the parts of the category that are included in Figure 5):

\[
\text{let } T_r \left[ \text{ComplexCategory} \right] \equiv \lambda T_r \left[ \% \right]. \lambda T_r \left[ R \right]. \\
\text{cons}(T_r \left[ \text{ComplexCategory}(R) \right], \\
T_W \left[ \text{if true then CommutativeRing() else Type()} \right] \uplus \\
T_W \left[ \text{if } R \text{ has IntegralDomain()} \text{ then } \text{exquo} \left( \left( R, R \right) \rightarrow \% \text{ else Type()} \right) \uplus \\
T_W \left[ \text{if true then FullyLinearlyExplicitRingOver(R) else Type()} \right] \uplus \text{Type}(\%)
\]
\]

Where the function $\uplus$ is the concatenation of two lists, see Fig. 6. We note that the callform of the category is used to generate both the name of the function representing the category, the names of the parameters of that function, and an $\mathbf{R}$-term which will become the reflexive assertion that the `ComplexCategory` is the category `ComplexCategory`. Beginning with the translation of the callform to the name of function we get:

\[
T_r \left[ \text{ComplexCategory} \right] \triangleq \text{ComplexCategory}
\]

that is, the identifier of the callform becomes an identifier in IL. The parameters of the callform are translated similarly into identifiers IL.

We note that we do not translate the types of the formal parameters: the type information is embedded within the specification lists that are passed as arguments to the function. For example, an argument passed to parameter $R$ would be a specification list, containing run-time evidence of all the types of the domain represented by $R$. Similarly, the callform is also translated into a specification (from the second line of the translation):

\[
T_X \left[ \text{ComplexCategory}(R) \right] \triangleq R_{id}(T_r \left[ \text{ComplexCategory} \right], [T_X \left[ R \right]]) \\
\triangleq R_{id}(\text{ComplexCategory}, [\text{car}(T_r \left[ R \right])]) \\
\triangleq R_{id}(\text{ComplexCategory}, [\text{car}(R)])
\]

Conditional specifications in categories are translated into if-statements in IL; furthermore, the value `true` from Spad becomes the values `true` in IL. Category extension declarations are translated into function calls of the same name. The function `Type` returns the specification list $[R_{id}(\text{Type}, [])]$.

\[
T_W \left[ \text{if true then CommutativeRing() else True} \right] \triangleq \text{if } T_r \left[ \text{true} \right] \text{ then CommutativeRing() else Type()} \\
\triangleq \text{if true then CommutativeRing() else Type()} \\
\triangleq \text{if true then CommutativeRing(\%) else Type(\%)}
\]

We now turn our attention to the translation of a has-predicate. Intuitively, a has-predicate $X$ has $C$ is asking if a domain $X$ has the type $C$. We translate this into a question asking if the specification of $C$ is within the list of specifications of $X$:

\[
T_r \left[ R \text{ has IntegralDomain()} \right] \triangleq T_X \left[ \text{IntegralDomain()} \right] \in T_r \left[ R \right] \\
\triangleq R_{id}(\text{IntegralDomain()} \in R).
\]

User-defined predicates are translated into a call to the function `lookup` (which we do not show the implementation of). For instance, the right-hand-side of the conjunction in the predicate of the final conditional specification is translated as:

\[
T_r \left[ \text{irreducible?(monomial(1, 2)SR+1)SR} \right] \\
\triangleq \text{lookup}(T_r \left[ \text{irreducible?} \right], [\text{car}(T_r \left[ \text{monomial(1, 2)SR+1} \right]), T_r \left[ R \right] \text{(})T_r \left[ \text{monomial(1, 2)SR+1} \right] \text{)}
\]

Where the function call `monomial(1, 2)SR+1` is recursively translated to calls to the function `lookup`. We note that we must evaluate the call to `monomial(1, 2)SR+1` twice: once to determine the type of the result and once to determine the value of the result. Since this presentation of the Spad language is purely functional, these computations can be folded together, memoized, etc.
The final result of the translation is:

\[
\text{let } \text{ComplexCategory} = \lambda \hat{\mathcal{R}}. \text{cons}(\text{R}_d(\text{ComplexCategory}, [\text{car}(\hat{R}))),
\begin{cases}
\text{if true then CommutativeRing}(\%) \text{ else Type}((\%)), \\
\begin{cases}
\text{R}_d(\text{IntegralDomain}, []) \in \hat{R} \text{ then } \\
\text{R}_d(\text{exquo}, [\text{R}_{...}([\text{car}(\%), \text{car}(\%)), \text{car}(\hat{R})], \text{nil}]) \text{ else Type}((\%))
\end{cases}, \\
\text{if true then FullyLinearlyExplicitRingOver}(\%, \hat{R}) \text{ else Type}((\%)), \\
\begin{cases}
\text{if and}(\text{R}_d(\text{Field}, []) \in \hat{R}, \\
\text{lookup}(\text{irreducible?}, [\text{car}(\text{lookup}(\text{monomial}, ...)(...)), \hat{R}]))(\text{lookup}(\text{monomial}, [...], \hat{R})(...)) \\
\text{then Field}((\%)) \text{ else Type}((\%))
\end{cases}
\end{cases})
\]