Towards Typing for Small-Step Direct Reflection

Jacques Carette  
Dept. of Computing and Software  
McMaster University  
Hamilton, Ontario, Canada  
carette@mcmaster.ca

Aaron Stump  
Computer Science Dept.  
The University of Iowa  
Iowa City, Iowa, USA  
astump@acm.org

Abstract

Direct reflection is a form of meta-programming in which program terms can intensionally analyze other program terms. Previous work defined a big-step semantics for a directly reflective language called Archon, with a conservative approach to variable scoping based on operations for opening a lambda-abstraction and swapping the order of nested lambda-abstractions. In this short paper, we give a small-step semantics for a revised version of Archon, based on operations for opening and closing lambda abstractions. We then discuss challenges for designing a static type system for this language, which is our ultimate goal.

Categories and Subject Descriptors  D.3.2 [Programming Languages]: Applicative (functional) languages; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying Reasoning about Programs

General Terms  Languages, Types

Keywords  Meta-Programming, Reflection, Small-Step Semantics, Symbolic Computation

1. Introduction

We are interested in typed, directly reflective, meta-programming languages with binders. By “directly reflective”, we mean that we can not only inspect all terms, but decompose them all as well. In other words, we would like a language in which all well-typed terms are simultaneously extensional and intensional. Furthermore, we would like to do this for as small an extension of the classical \( \lambda \)-calculus as possible.

In previous work [12], the second author defined a directly reflective language called Archon, via a big-step operational semantics. This used a conservative approach to variable scoping based on operations for opening a lambda-abstraction and swapping the order of nested lambda-abstractions. Here, we give a small-step semantics for a revised version of Archon, based on operations for opening and closing lambda abstractions. Since type systems are usually developed based on small-step semantics, this is an important first step.

If we were interested in a combinatory calculus for this task, we would adapt recent work of Jay and Palsberg [7], about which we will say more in the next section. From our point of view, their language is missing a crucial ingredient: binders. We want to be able to do direct reflection, in a hygienic manner, on terms with binders. And, as the authors readily admit, their treatment of typing for the \( E \) combinator is unsatisfactory.

Our work belongs to a rich tradition of investigations on reflection, intensionality, open code, and typed meta-programming, thus we first give a (brief) overview of some of these strands. We present two versions of Archon: first the one from [12], and then a new one which we believe to be more convenient to work with. We then present our work-in-progress on a type system for (revised) Archon.

2. Related Work

Jay and Palsberg [7] achieve something closely related, but for a combinatory calculus. They start from the pure factorisation calculus [6], augmented with some usual combinators from the \( SK \) combinatory calculus, as well as two new combinators, \( B \) and \( E \), respectively for blocking computation and for deciding equality of operators. They then proceed to add syntactic sugar for the identity combinator, \( \lambda \)-abstraction, \( \text{let} \) and \( \text{let rec} \). They furthermore add pattern-matching with path polymorphism [5], but this too can be de-sugared. This is a remarkable piece of work. Unfortunately, it does not achieve our goals: while it is possible to program in their system as if one were in a \( \lambda \)-calculus, introspection can only be done at the level of the underlying combinatory calculus. This is in every way similar to the situation of introspection in Java, whereby one can only examine (and modify) the byte code of a Java class, but not its source code. And, as they mention in section 7.1, the typing of the \( E \) combinator is not entirely satisfactory.

Closer still to achieving part of what we want is the work of Rendel, Ostermann and Hofer [10], who define a typed self-representation of the (pure) \( \lambda \)-calculus. To achieve this, they first leverage a technique from [2] whereby they abstract over a type constructor, and then repeat this at the type level (to introduce kind-polymorphism). This necessitates an extension of system \( F_\omega \) which they call \( F_\omega^L \), with a rule which amounts to kind:kind. While this is not as bad as type:kind, it is nevertheless quite discom-forting. Furthermore, while they do indeed achieve typed self-interpretaion, it is not direct as they only interpret quoted terms (their terms are not self-quoting), nor do they allow reflection.

Another interesting strand concerns intensional logic, and in particular the work of Paul Gilmore on Intensional Type Theory (ITT) [3]. Terms in ITT have two types, an extensional and an intensional type; closed terms in ITT have these two types coincide. We see this as a very valuable insight.

There is a huge amount of work on typed staged languages, which allow a restricted amount of code manipulation, but no reflection, direct or indirect. Most influential on us has been the
work of Higier [11] on a typed language with first-class open and closed code fragments. He cleverly moves the typing context into the types, to allow for a very fine-grained tracking of dependency in an explicitly staged language. Kim, Yi and Calcagno [8] essentially extend this with many more features, including variable-capturing substitution at “higher levels”.

Atkey, Lindley and Yallop [1] take a different tack: rather than deal with self-representation, especially of embedded languages, they really work with language pairs (L₁, L₂), with explicit equivalences between the languages. They achieve reflection because one language is always represented (in the host language) as a first-sentence between the languages. They achieve reflection because one

4. Revised Archon: Syntax and Semantics

Figure 3 gives the syntax for terms in our revised version of Archon. Contexts C are defined for the operational semantics, defined in Figures 4 and 5. As usual for reduction defined with contexts, the clauses defining contexts C show where reduction may take place in a term; so we may reduce in an argument to a call-by-value λ-abstraction, but not a call-by-name one. We allow symbolic computation in both HOSC-Archon and revised Archon, so the notion of values v includes (via headValue) applications of variables x to values. We use (calling) convention markers θ to indicate whether λ-abstractions are call-by-name (θ) or call-by-value (v). The most important change, of course, is that we have removed swap and replaced it with close. The motivation is threefold: swap seems less fundamental than close, close gives us finer control over scoping, and open/close exhibit a more pleasant natural symmetry. 

tf and ff denote the usual Church encodings of booleans true and false.
to match its original convention $\theta$.  

\[
\text{open'} := \lambda^n x. \lambda^n x'. \text{open } x (\lambda^n y. \lambda^n y'. ((\lambda^n z'. \text{close}^n y z'') (x' y''))) 
\]

This term takes in a term $x$ to open and a function $x'$ to apply to the bound variable and body of $x$. It opens $x$, using a term which will receive the bound variable of $x$ as $y$ and the body as $y'$. It then calls the original function $x'$ on $y$ and $y'$, obtaining the result as $x''$. It then re-binds $y$ around that result $x''$ using close.

For another example, Figure 6 shows how the swap operator of HOSC-Archon can be implemented in revised Archon (we again re-bind the two variables just as $\lambda^n$-abstractions, just for easier readability; we could use decomposition to re-bind with the original convention). The term given in the figure for swap takes in a term $x$ to open and a function $x'$ to apply to the bound variable and body of $x$. It opens $x$, using a term which will receive the bound variable of $x$ as $y$ and the body as $y'$. It then calls the original function $x'$ on $y$ and $y'$, obtaining the result as $x''$. It then re-binds $y$ around that result $x''$ using close.

The fact that revised Archon can simulate swap from HOSC-Archon shows that revised Archon is at least as expressive as HOSC-Archon. To make this more precise, suppose we have defined a translation $\cdot :: \cdot$ from HOSC-Archon terms to revised Archon terms, in the obvious way, using the definition of Figure 6 for swap. Then we have the following theorem:

**Theorem 1.** If $t \downarrow \tau$ in HOSC-Archon, then we also have $\mid t \mid \rightarrow \tau' [\tau']$ in revised Archon.

**Proof.** The proof is by straightforward induction on the structure of the derivation of the HOSC-Archon evaluation judgment. It makes use of the fact that if $t \downarrow \tau$, then $\tau'$ is an HOSC-Archon value, which translates to a revised Archon value. It also makes use of a standard derived congruence lemma for revised Archon, stating that $t \rightarrow \tau' \rightarrow \tau'$ implies $\mid t \mid \rightarrow \tau' [\tau']$. End proof.

#### 5. Types

Our goal is to devise a static type system for revised Archon, which will ensure that open $\cdot$ cannot be called on a term which is not a $\lambda$-abstraction; veq $\cdot$ can only be called on terms which are variables; and where the free variables of terms can be tracked by the type system. Note that we really do mean that open must be called on a $\lambda$-abstraction, i.e. its first argument will not be evaluated, implying that staging properties, although implicit, are nevertheless very important. Tracking of free variables can be useful if one wished to enforce statically some additional policy about free variables. For example, we might want to require that in a top-level definition, the defining term is closed; or we might want to disallow evaluation of terms with free variables unless they are statically guaranteed to be $\lambda$-abstractions.

As perhaps should not be surprising given the complexity of the type systems in related works, it turns out to be quite subtle to design a liberal but sound type system to meet the above goals. Here, we highlight challenges and sketch ideas in that direction, starting with some simple examples which such a type system should allow or reject. Note that eventually, one would like to have a system of annotated (Curry-style) terms with a decidable type-checking problem; but for purposes of the examples below, we work with unannotated (Church-style) terms, as this allows us to avoid attempting to define the syntax for types at this point.

#### 5.1 Simple Examples

**Basic swap example (accept).** Let swap be as defined in Figure 6 above. Then swap itself should be typable, with a type that reflects its argument should be a doubly-nested $\lambda$-abstraction. So the following term should be typable:

\[
\text{swap} (\lambda^n x. \lambda^n y. x) 
\]

This term simply swaps variables $x$ and $y$. The type assigned to this term should reflect the fact that the term is closed.

**Indirect swap (accept).** The term below should be typable where the type of $x$ expresses that it is a doubly-nested $\lambda$-abstraction:

\[
\lambda^n x. \text{swap } x 
\]

Furthermore, typing should probably express that the sets of free variables of the input and output of this $\lambda$-abstraction are the same.

**Scoping and swap (reject).** The following example should be disallowed, even if the $\lambda$-abstraction is given a type like $T \Rightarrow T \Rightarrow T$:

\[
\text{swap} (\lambda^n x. x) 
\]

This is the most direct reflection of our desire for swap to be an *intensional* operation.

**Decomp and open (accept).** Typing for decomposition should use some kind of type refinement, so that in each branch of a decomposition, typing can take into account that the scrutinee term has a known form. Thus

\[
t : a (\lambda^n x. \lambda^n y. \text{open } y t) b c d e f 
\]

should be typable, for typable scrutinee $t$, a suitable term $t'$ to apply to the bound variable and body of $t$, and suitable other decomposition branches $a$ through $f$.

**Variables ranging over variables (accept).** The following term should be typable, with a type expressing that if the arguments supplied for $x$ and $y$ are variables, then the result of applying the
\(\text{swap} := \lambda^n \cdot \text{open } x \lambda^n y. \lambda^n \cdot \text{open } x' \lambda^n y'. \lambda^n \cdot \text{close } x'' \cdot \text{close } n (\lambda^n \cdot \text{close } y) (\lambda^n \cdot \text{close } y')\)

Figure 6. Definition of swap Using Open and Close in Revised Archon

\(\lambda\)-abstraction is a boolean:

\(\lambda^n x. \lambda^n y. \text{veq } x y\)

Note that this requires the type system to be able to express the idea that a variable (like \(x\)) ranges over free variables, since if a term of a different form is supplied for \(x\), the application of this \(\lambda\)-abstraction will have a stuck term (as \(\text{veq } t t'\) is stuck unless both \(t\) and \(t'\) are variables).

Application, variables and swap (reject). It is entirely possible that a free variable has a type such that the left term below is well-typed, while the right term is not.

\[ f \ x \ y \ \text{swap } f \]

While \(f\) represents a function of 2 arguments, that does not imply that is is a function of 2 arguments.

5.2 Ideas on Typing

Shapes and types. One idea that seems promising is to incorporate both shapes and types into the type system. A shape is a type-like expression which expresses more about the intensional form of a term. An example shape is \(T_1 \Rightarrow T_2 \Rightarrow T_1\). This shape expresses (among other things) that the term in question is an application; that property is usually not expressible in a type system. Here, we expect ideas in an emerging line of research on “small-step typing” to help, since there, terms are rewritten in a small-step fashion to their types, passing through shapes as intermediary forms [4, 9, 13].

Tracking free variables. Since an open term fundamentally depends on the names of the free variables that it contains, if we wish to enforce any policy which depends on the presence or absence of (certain) free variables, we need to track this. For example, internalizing capture-avoiding substitution requires this feature. Binders Unbound [14] gives other examples of the utility of this feature.

Denotations of types. Since types are specifications, it can be useful to define a semantics for types in a denotational style, as a guide for a decidable type system. Such a semantics determines what types are supposed to mean. A basic example is the following for function types \(T \rightarrow T'\), from reducibility for normalization of \(\lambda\)-calculus:

\[ t \in [T \rightarrow T'] \Leftrightarrow \forall t' \in [T' \rightarrow T'] \] \(t \ t' \in [T']\)

This type thus expresses an extensional view of terms: a term \(t\) is in the meaning of the type \(T \rightarrow T'\) iff for every input \(t'\) in the meaning of \(T\), the application \(t \ t'\) is in the interpretation of \(T'\). For revised Archon, we anticipate needing types embodying this extensional viewpoint, but also ones with a more intensional character.

For the terms \(\lambda^n x. x\) and \(\lambda^n x. \lambda^n y. (x y)\) are indistinguishable extensionally when \(x\) is taken to range over functions; but we must distinguish them somehow in order to allow the “indirect swap” example above, while ruling out the “scoping and swap” example.

5.3 Other semantic differences

Open terms differ significantly from closed terms. For example, \(x + 1\) and \(\lambda^n x. x + 1\) may at first seem quite similar, since they can be inter-derived via \(\text{close } n (x + 1)\), and open \(\lambda^n x. x + 1\) \((\lambda^n x. \lambda^n b. b)\). Nevertheless, we assert that \(x + 1\) represents the “add 1 concept”, while \(\lambda^n x. x + 1\) represents the action of adding 1. Another example is that we can easily add a constant which represents the “halts” concept (as applied to terms), but we would be hard-pressed to instantiate it.

6. Conclusion

We believe that revised Archon has the “right” operational semantics for a useful core calculus for (typed) meta-programming which incorporates many useful features: binders, direct reflection, and symbolic computation. Another significant advantage of direct reflection is that persistent code is no longer an issue, unlike in most other calculi. Our ongoing work makes us quite optimistic that by combining a shape system, type refinement with free variable tracking will culminate in a static “type” system for revised Archon.

Acknowledgements: We wish to thank Marc Bender, Barry Jay, and the anonymous PEPM ‘12 reviewers for insightful comments.

References


