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 and 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 λ-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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PEPM'12, January 23–24, 2012, Philadelphia, PA, USA.
Copyright © 2012 ACM 978-1-4503-1118-2/12/01. . . $10.00

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, λ-abstraction, `let` and `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 λ-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) λ-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_ω, which they call $F_\omega^\star$, with a rule which amounts to *kind:kind*. While this is not as bad as *type:type*, it is nevertheless quite discomforting. Furthermore, while they do indeed achieve typed self-interpretation, 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

$$T \quad ::= \quad x \mid \lambda x.T \mid \bar{\lambda} x.T \mid T\ T \mid \mathbf{open}\, T\, T \mid \mathbf{vcomp}\, T\, T \mid \mathbf{swap}\, T \mid T : T\,T\,T\,T\,T\,T\,T$$

Figure 1. The Syntax of HOSC-Archon Terms

$$\frac{T_2\ x\ T_1 \Downarrow R \qquad x \notin FV(\lambda^* x.T_1) \cup FV(T_2)}{\mathbf{open}\ (\lambda^* x.T_1)\ T_2 \Downarrow \lambda^* x.R}\ \text{E-OpenLam}$$

$$\frac{T \Downarrow \lambda^1 x.\lambda^2 y.R}{\mathbf{swap}\, T \Downarrow \lambda^2 y.\lambda^1 x.R}\ \text{E-SwapLam}$$

Figure 2. HOSC-Archon Rules for Open and Swap

work of Rhiger [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, include 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 (L_1, L_2), with explicit equivalences between the languages. They achieve reflection because one language is always represented (in the host language) as a first-order datatype, on which intensional analysis may be performed. While quite pragmatic, this is theoretically unsatisfying.

3. HOSC-Archon

Previous work of the second author defined a directly reflective meta-programming language called Archon [12]. We will refer to that language as HOSC-Archon in this paper. The syntax for HOSC-Archon terms is given in Figure 1. In this section, we retain the same syntax as in [12] , though in the next section we will depart from this somewhat. Here, the construct $\bar{\lambda}$ is for call-by-name λ-abstraction, while λ is reserved for call-by-value abstraction. The $\mathbf{vcomp}$ construct is for comparing two variables for equality. The : construct is for intensional case-analysis, called *decomposition*, on the form of an unevaluated term, and includes terms to apply for each of the seven possible forms (one for each syntactic construct) of the scrutinized term to the left of the colon. Colon is a weak form of pattern-matching which, by construction, is always exhaustive.

We will give a full semantics for our revised version of this language in the next section. First, for comparison, we consider the semantics of HOSC-Archon's $\mathbf{open}$ and $\mathbf{swap}$, in Figure 2. Evaluation of a $\mathbf{swap}$-term (the rule E-SwapLam) evaluates the subterm T and then, if it is a consecutively nested λ-abstraction, swaps the order of the λ-bindings. The E-OpenLam shows the situation where $\mathbf{open}$ has an unevaluated λ-abstraction (λ^* indicates either λ or $\bar{\lambda}$), and evaluation applies a term T_2 to the bound variable and the body of that λ-abstraction. It is assumed that variables are renamed before the λ-abstraction is opened, so that the variable x is not free in T_2. After evaluation of that application completes, the result is rebound with λx, thus preventing variables from escaping their scopes. It is this behavior we will relax in the next section. It is possible to define highly intensional meta-programming operations like testing terms for alpha-equivalence, or Mogensen-Scott decoding and encoding functions, in HOSC-Archon [12].

4. Revised Archon: Syntax and Semantics

Figure 3 gives the syntax for terms in our revised version of Archon. Contexts $\mathcal{C}$ are defined for the operational semantics, defined in Figures 4 and 5. As usual for reduction defined with contexts, the clauses defining contexts $\mathcal{C}$ 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 ($\mathbf{n}$) or call-by-value ($\mathbf{v}$). The most important change, of course, is that we have removed $\mathbf{swap}$ and replaced it with $\mathbf{close}$. The motivation is threefold: $\mathbf{swap}$ seems less fundamental than $\mathbf{close}$, $\mathbf{close}$ gives us finer control over scoping, and $\mathbf{open}/\mathbf{close}$ exhibit a more pleasant natural symmetry. $\mathbf{tt}$ and $\mathbf{ff}$ denote the usual Church encodings of booleans $true$ and $false$.

$$\begin{array}{lcl}
\textit{convention } \theta & ::= & \mathbf{v} \mid \mathbf{n} \\
\textit{term } t & ::= & x \mid t\, t' \mid \lambda^\theta x.t \mid \mathbf{open}\, t\, t' \\
 & & \mid \mathbf{close}^\theta x\ t \mid \mathbf{veq}\, t\, t' \mid t : \overrightarrow{t'} \\
\textit{context } \mathcal{C} & ::= & * \mid \mathcal{C}\ t \mid h\ \mathcal{C} \mid (\lambda^{\mathbf{v}} x.t)\ \mathcal{C} \\
\textit{concreteValue } v & ::= & \lambda^\theta x.t \mid h \\
\textit{headValue } h & ::= & x \mid h\ v
\end{array}$$

Figure 3. Syntax for (Revised) Archon Terms

$$\frac{}{\mathcal{C}[(\lambda^{\mathbf{v}} x.t)\ v] \to \mathcal{C}[[v/x]t]}\ \text{C\_BetaV}$$

$$\frac{}{\mathcal{C}[(\lambda^{\mathbf{n}} x.t)\ t'] \to \mathcal{C}[[t'/x]t]}\ \text{C\_BetaN}$$

$$\frac{x' \notin \mathbf{FV}\,(\mathcal{C}[t\, t'])}{\mathcal{C}[\mathbf{open}\,(\lambda^\theta x.t)\ t'] \to \mathcal{C}[((t'\ x')\ [x'/x]t)]}\ \text{C\_Open}$$

$$\frac{}{\mathcal{C}[\mathbf{close}^\theta x\ t] \to \mathcal{C}[\lambda^\theta x.t]}\ \text{C\_Close}$$

$$\frac{x \neq x'}{\mathcal{C}[\mathbf{veq}\, x\, x'] \to \mathcal{C}[\mathbf{ff}]}\ \text{C\_VarDiff}$$

$$\frac{}{\mathcal{C}[\mathbf{veq}\, x\, x] \to \mathcal{C}[\mathbf{tt}]}\ \text{C\_VarSame}$$

Figure 4. Small-Step Semantics for (Revised) Archon, Non-Decomp Rules

The rules of Figures 4 and 5 define a small-step operational semantics for Archon terms. $\overrightarrow{t}$ denotes $t_1 t_2 t_3 t_4 t_5 t_6 t_7$. This semantics bans variable capture during substitution, just like HOSC-Archon, but it now permits variables to escape their scopes. So if we have $t \to^+ t'$, then it can happen that the set $FV(t')$ of free variables of t' is not a subset of $FV(t)$. New free variables may appear during reduction, because unlike in HOSC-Archon, revised Archon does not insist that a variable x which is freed by $\mathbf{open}$ must always be re-bound around a resulting term which might contain x free. One could certainly implement this re-binding discipline on top of $\mathbf{open}$ and $\mathbf{close}$: one can just require all terms to use $\mathbf{open}'$ defined as follows. For simplicity we always re-bind the variable as call-by-name; using decomposition one could re-bind the variable

$$\frac{}{\mathcal{C}[x : \overrightarrow{t}\,] \to \mathcal{C}[t_1\ x]}\ \text{C\_DV}\textsc{ar} \qquad \frac{}{\mathcal{C}[(\lambda^{\mathbf{n}}x.t) : \overrightarrow{t}\,] \to \mathcal{C}[t_2\ \mathbf{ff}\ (\lambda^{\mathbf{n}}x.t)]}\ \text{C\_DC}\textsc{bn} \qquad \frac{}{\mathcal{C}[(t'\ t'') : \overrightarrow{t}\,] \to \mathcal{C}[t_3\ t'\ t'']}\ \text{C\_DA}\textsc{pp}$$

$$\frac{}{\mathcal{C}[(\mathbf{open}\ t'\ t'') : \overrightarrow{t}\,] \to \mathcal{C}[t_4\ t'\ t'']}\ \text{C\_DO}\textsc{pen} \qquad \frac{}{\mathcal{C}[(\mathbf{close}^{\mathbf{v}} x\ t') : \overrightarrow{t}\,] \to \mathcal{C}[t_5\ \mathbf{tt}\ x\ t']}\ \text{C\_DC}\textsc{lose}\text{S}$$

$$\frac{}{\mathcal{C}[(\mathbf{veq}\ t'\ t'') : \overrightarrow{t}\,] \to \mathcal{C}[t_6\ t'\ t'']}\ \text{C\_DV}\textsc{eq} \qquad \frac{}{\mathcal{C}[(\mathbf{close}^{\mathbf{n}} x\ t') : \overrightarrow{t}\,] \to \mathcal{C}[t_5\ \mathbf{ff}\ x\ t']}\ \text{C\_DC}\textsc{lose}\text{D}$$

$$\frac{}{\mathcal{C}[(t' : \overrightarrow{t'}) : \overrightarrow{t}\,] \to \mathcal{C}[t_7\ t'\ t'_1\ t'_2\ t'_3\ t'_4\ t'_5\ t'_6\ t'_7]}\ \text{C\_DD}\textsc{ecomp} \qquad \frac{}{\mathcal{C}[(\lambda^{\mathbf{v}}x.t) : \overrightarrow{t}\,] \to \mathcal{C}[t_2\ \mathbf{tt}\ (\lambda^{\mathbf{v}}x.t)]}\ \text{C\_DC}\textsc{bv}$$

Figure 5. Small-Step Semantics for (Revised) Archon, Decomp Rules

to match its original convention θ.

$$\begin{array}{l}\mathbf{open}' := \\ \quad \lambda^{\mathbf{n}}x.\lambda^{\mathbf{n}}x'.\mathbf{open}\ x\ (\lambda^{\mathbf{n}}y.\lambda^{\mathbf{n}}y'.((\lambda^{\mathbf{v}}x''.\mathbf{close}^{\mathbf{v}}\ y\ x'')\ (x'\ y\ y')))\end{array}$$

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 $\mathbf{close}$.

For another example, Figure 6 shows how the $\mathbf{swap}$ operator of HOSC-Archon can be implemented in revised Archon (we again re-bind the two variables just as $\lambda^{\mathbf{n}}$-abstractions, just for easier readability; we could use decomposition to re-bind with the original convention θ). The term given in the figure for $\mathbf{swap}$ takes in a term x, assumed to be a doubly nested λ-abstraction of the form $\lambda^{\theta}_1 y.\lambda^{\theta}_2 y'.x''$; opens it twice (that is, opens it and then opens its body) to obtain y, y', and x''; and then closes the variables in the reverse order (with a call-by-value β-redex binding variable x''' to force evaluation of the first $\mathbf{close}$-term). This results in the term $\lambda^{\mathbf{n}}y'.\lambda^{\mathbf{n}}y.x''$, which indeed has swapped the order of the bound variables, as desired.

The fact that revised Archon can simulate $\mathbf{swap}$ 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 $|\cdot|$ from HOSC-Archon terms to revised Archon terms, in the obvious way, using the definition of Figure 6 for $\mathbf{swap}$. Then we have the following theorem:

THEOREM 1. *If $t \Downarrow t'$ in HOSC-Archon, then we also have $|t| \to^* |t'|$ 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 $t \Downarrow t'$, then t' 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 $t \to^* t'$ implies $\mathcal{C}[t] \to^* \mathcal{C}[t']$. **End proof.**

5. Types

Our goal is to devise a static type system for revised Archon, which will ensure that $\mathbf{open}$ cannot be called on a term which is not a λ-abstraction; $\mathbf{veq}$ 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 $\mathbf{open}$ must be called on a λ-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 λ-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 (Church-style) terms with a decidable type-checking problem; but for purposes of the examples below, we work with unannotated (Curry-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 $\mathbf{swap}$ be as defined in Figure 6 above. Then $\mathbf{swap}$ itself should be typable, with a type that reflects that its argument should be a doubly-nested λ-abstraction. So the following term should be typable:

$$\mathbf{swap}\ (\lambda^{\mathbf{n}}x.\lambda^{\mathbf{n}}y.x)$$

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 λ-abstraction:

$$\lambda^{\mathbf{n}}x.\mathbf{swap}\ x$$

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

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

$$\mathbf{swap}\ (\lambda^{\mathbf{n}}x.x)$$

This is the most direct reflection of our desire for $\mathbf{swap}$ 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

$$t\ :\ a\ (\lambda^{\mathbf{n}}x.\lambda^{\mathbf{n}}y.\mathbf{open}\ y\ t')\ b\ c\ d\ e\ f$$

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

$$\mathbf{swap} := \lambda^{\mathbf{n}} x.\mathbf{open}\, x\, \lambda^{\mathbf{n}} y.\lambda^{\mathbf{n}} x'.\mathbf{open}\, x'\, \lambda^{\mathbf{n}} y'.\lambda^{\mathbf{n}} x''.((\lambda^{\mathbf{v}} x'''.\mathbf{close}^{\mathbf{n}}\, y'\, x''')\, (\mathbf{close}^{\mathbf{n}}\, y\, x''))$$

Figure 6. Definition of $\mathbf{swap}$ Using Open and Close in Revised Archon

λ-abstraction is a boolean:

$$\lambda^{\mathbf{n}} x.\lambda^{\mathbf{n}} y.\mathbf{veq}\, x\, y$$

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 λ-abstraction will have a stuck term (as $\mathbf{veq}\, t\, t'$ 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.

$$f\, x\, y \qquad \mathbf{swap}\, f$$

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 $(T_1 \Rightarrow T_2)\ T_1$. 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 $T \to T'$, from reducibility for normalization of λ-calculi:

$$t \in [\![T \to T']\!] \Leftrightarrow \forall t' \in [\![T]\!].\ t\ t' \in [\![T']\!]$$

This type thus expresses an extensional view of terms: a term t is in the meaning of the type $T \to T'$ 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 $\lambda^{\mathbf{n}} x.x$ and $\lambda^{\mathbf{n}} x.\lambda^{\mathbf{n}} y.(x\ y)$ 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 $\lambda^{\mathbf{v}} x.x + 1$ may at first seem quite similar[1], since they can be inter-derived via $\mathbf{close}^{\mathbf{v}} x\ (x + 1)$, and $\mathbf{open}\ (\lambda^{\mathbf{v}} x.x + 1)\ (\lambda^{\mathbf{n}} v.\lambda^{\mathbf{n}} b.b)$. Nevertheless, we assert that $x + 1$ represents the "add 1 concept", while $\lambda^{\mathbf{v}} x.x + 1$ 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.

[1] in an obvious extension of revised Archon

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

[1] Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 37–48. ACM, 2009.

[2] Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.

[3] Paul C. Gilmore. An intensional type theory: Motivation and cut-elimination. *J. Symb. Log.*, 66(1):383–400, 2001.

[4] M. Hills and G. Rosu. A Rewriting Logic Semantics Approach to Modular Program Analysis. In C. Lynch, editor, *Proceedings of RTA 2010, Edinburgh, Scotland, UK*, pages 151–160, 2010.

[5] Barry. Jay. *Pattern calculus : computing with functions and structures*. Springer, Berlin ; London :, 2009.

[6] Barry Jay and Thomas Given-Wilson. A combinatory account of internal structure. Accepted to *Journal of Symbolic Logic*.

[7] Barry Jay and Jens Palsberg. Typed self-interpretation by pattern matching. In *Proceeding of ICFP 2011*, pages 247–258. ACM, 2011.

[8] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. In *POPL 2006*, pages 257–268. ACM, 2006.

[9] G. Kuan, D. MacQueen, and R. Findler. A rewriting semantics for type inference. In *Proceedings of the 16th European conference on Programming (ESOP)*, pages 426–440. Springer-Verlag, 2007.

[10] Tillmann Rendel, Klaus Ostermann, and Christian Hofer. Typed self-representation. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 293–303. ACM, 2009.

[11] Morten Rhiger. First-class Open and Closed Code Fragments. In Marko C. J. D. van Eekelen, editor, *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming (TFP)*, volume 6, pages 127–144, 2005.

[12] Aaron Stump. Directly Reflective Meta-Programming. *Higher Order and Symbolic Computation*, 22(2):115–144, 2009.

[13] Aaron Stump, Garrin Kimmell, and Roba El Haj Omar. Type Preservation as a Confluence Problem. In Manfred Schmidt-Schauß, editor, *Proceedings of RTA 2011*, volume 10 of *LIPIcs*, pages 345–360, 2011.

[14] Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. Binders unbound. In *Proceeding of ICFP '11*, pages 333–345, New York, NY, USA, 2011. ACM. doi: 10.1145/2034773.2034818.