Self-applicable Partial Evaluation for Pure Lambda Calculus

Torben Æ. Mogensen
DIKU, University of Copenhagen, Denmark

Abstract

Partial evaluation of an applied lambda calculus was done some years ago in the *lambda-mix* project. When moving to pure lambda calculus, some issues need to be considered, most importantly how we represent programs in pure lambda calculus. We start by presenting a compact representation schema for λ-terms and show how this leads to an exceedingly small and elegant self-interpreter. Partial evaluation is discussed, and it is shown that partial evaluation in the most general sense is uncomputable. There are several ways of restricting partial evaluation. We choose one of these, which requires explicit binding time information. Binding time annotations are discussed, and the representation schema is extended to include annotations. A partial evaluator is then constructed as an extension of the self-interpreter, and self-application is performed to produce possibly the smallest non-trivial compiler generator in the literature. It is shown that binding time analysis can be done by modifying the type-inference based method of *lambda-mix.*

1 Preliminaries

The set of λ-terms, Λ, is defined by the abstract syntax:

$$\Lambda = V \mid \Lambda\,\Lambda \mid \lambda V.\Lambda$$

where V is a countable infinite set of distinct variables. (Possibly subscripted) lower case letters $a, b, x, y, \ldots$ are used for variables, and capital letters $M, N, E, \ldots$ for λ-terms. We will assume familiarity with the rules for reduction in the lambda calculus, and mention these without reference. The shorthand $\lambda x_1 \ldots x_n.M$ abbreviates $\lambda x_1 . \ldots \lambda x_n.M$ and $M_1\, M_2 \ldots M_n$ abbreviates $(\ldots (M_1\, M_2) \ldots M_n)$

Two λ-terms are considered *identical* if they only differ in the names of bound variables (i.e., they are α-convertible), and *equal* if they can be β-reduced to identical λ-terms. We use the symbol $\equiv$ for identity and $=_\beta$ for equality.

When M has a normal form we denote this by NF_M. NF_Λ is the set of λ-terms in normal form. FV_M is the set of free variables in M.

Data is usually represented in pure lambda calculus by λ-terms in normal form (e.g. Church numerals). Normal form λ-terms are indeed "constants" with respect to reduction, which is a natural requirement for data. Normally data values are represented in such a way that the required operations on them are simple to do in pure lambda calculus. Note that the only way to inspect a λ-term from within the lambda calculus itself is to apply it to one or more arguments. Also, note that an application in itself does not involve evaluation, it is a mere syntactic construction. Reduction must be explicitly stated by NF or by use of the $=_\beta$ relation.

A *representation schema* for the lambda calculus is an injective (up to identity) mapping $\lceil\cdot\rceil : \Lambda \to \mathsf{NF}_\Lambda$. That is, $\lceil\cdot\rceil$ will represent any λ-term by a λ-term in normal form. Furthermore, the representations of two λ-terms are identical *iff* the λ-terms are.

2 A representation schema

We will use the representation schema from [Mogensen 1992]. It uses a combination of *higher order abstract syntax* [Pfenning and Elliot 1988] and a well-known way of representing signatures using λ-terms. This is a combination of the usual representations of products and booleans in pure lambda calculus.

Higher order abstract syntax is an abstract syntax representation that extends syntax trees with the abstraction mechanism of the lambda calculus. The idea is to represent scope rules by λ-abstraction. It is no surprise then, that higher order abstract syntax easily captures the scope rules of the lambda calculus. A higher order abstract syntax representation $\lfloor\cdot\rfloor$ of Λ using unary constructors *Var* and *Abs* and a binary construcor *App* is:

$$\begin{array}{lcl} \lfloor x \rfloor & = & \mathit{Var}(x) \\ \lfloor M\,N \rfloor & = & \mathit{App}(\lfloor M \rfloor, \lfloor N \rfloor) \\ \lfloor \lambda x.M \rfloor & = & \mathit{Abs}(\lambda x.\lfloor M \rfloor) \end{array}$$

Note that binding of variables is handled meta-circularly by binding of variables. For example the coding of $\lambda x.x\,x$ is:

$$\mathit{Abs}(\lambda x.\mathit{App}(\mathit{Var}(x), \mathit{Var}(x)))$$

Representation of signatures can be done in the following way: for any sort S in the signature, let Sc_i, $i = 1 \ldots n_s$ be the constructors in that sort. Represent $Sc_i(t_1, \ldots, t_m)$ by

$$\lambda x_1 \ldots x_{n_s}.x_i\,\overline{t_1}\;\ldots\;\overline{t_m}$$

where $\overline{t_j}$ is the representation of t_j. As an example, the signature of lists can be represented as:

$$\begin{array}{lcl} \overline{Nil} & \equiv & \lambda xy.x \\ \overline{Cons(A,B)} & \equiv & \lambda xy.y\,\overline{A}\,\overline{B} \end{array}$$

This coding allows a switch on constructor names to be implemented in a simple fashion in the lambda calculus. For example

$$\overline{case\ E\ of\ Nil \Rightarrow F;\ Cons(a,b) \Rightarrow G} \equiv \overline{E}\,\overline{F}\,(\lambda ab.\overline{G})$$

Combining these two ideas we get the following representation schema for the lambda calculus:

$$\begin{array}{lcl} \lceil x \rceil & \equiv & \lambda abc.a\,x \\ \lceil M\,N \rceil & \equiv & \lambda abc.b\,\lceil M \rceil\,\lceil N \rceil \\ \lceil \lambda x.M \rceil & \equiv & \lambda abc.c\,(\lambda x.\lceil M \rceil) \end{array}$$

where a, b, c are variables *not* occurring free in the λ-term on the left-hand side of the equation. Such variables can always be found, e.g. by choosing from the start three variables that do not occur in the entire λ-term. It is clear that the conditions (normal form and injectivity) for representation schemae are fulfilled. Note that

$$\mathsf{FV}_M = \mathsf{FV}_{\lceil M \rceil}$$

As an example, the coding of $\lambda x.x\,x$ is shown below:

$$\lambda abc.c\,(\lambda x.(\lambda abc.b\,(\lambda abc.a\,x)(\lambda abc.a\,x)))$$

It is easy to see that this representation is linear in the size of the represented λ-terms. In fact, the size (measured as the number of variables plus the number of applications plus the number of abstractions) of the representation is roughly 7 times the size of the original term. Operations like testing, decomposition and building of terms are quite efficient using this representation, requiring only a few β-reductions each.

3 Self-interpretation

A *self-interpreter* is a λ-term E, such that

$$E\,\lceil M \rceil\ =_\beta\ M$$

for any λ-term M. That is, E applied to the representation of M is equal to M itself.

The self-interpreter is taken from [Mogensen 1992], which also contains a proof of correctness. We will (for the sake of readability) initially present the self-interpreter using recursive equations and higher order abstract syntax. Then we will use the coding from above to convert it into the pure lambda calculus.

β-reduction of the abstractions in the higher order syntax is used to perform substitution in the interpreter. Thus no environment is needed. The effect is that some β-redices perform substitution, and others simulate reduction in the interpreted program. Apart from this slight subtlety, the interpreter below is remarkably easy to understand.

$$\begin{array}{lcl} E[Var(x)] & = & x \\ E[App(M,N)] & = & E[M]\ E[N] \\ E[Abs(M)] & = & \lambda v.E[(M\,v)] \end{array}$$

We now code the syntax as λ-terms, replace the pattern matching by application (as explained in the previous section) and use a fixed point combinator to eliminate explicit recursion. This yields the complete self-interpreter:

$$\begin{array}{lcl} E & \equiv & Y\,\lambda e.\lambda m.m\ \ (\lambda x.x) \\ & & \qquad\qquad\quad (\lambda mn.(e\,m)\,(e\,n)) \\ & & \qquad\qquad\quad (\lambda m.\lambda v.e\,(m\,v)) \\ \text{where} & & \\ Y & \equiv & \lambda h.(\lambda x.h\,(x\,x))\,(\lambda x.h\,(x\,x)) \end{array}$$

4 Partial evaluation

We first define what a partial evaluator for pure lambda calculus is. Informally, a partial evaluator should take a representation of a λ-term and the values of some of the parameters, and return a representation of a λ-term that is equivalent to the application of the original term to these parameters. As an equation, we can define that a partial evaluator is a λ-term P, such that there for any λ-terms M and $s_1\ \ldots\ s_n$ exists a λ-term $M_{s_1\,\ldots\,s_n}$, such that

$$P\,\lceil M \rceil\,s_1\ \ldots\ s_n\ =_\beta\ \lceil M_{s_1\,\ldots\,s_n} \rceil$$

and

$$M_{s_1\,\ldots\,s_n}\ =_\beta\ M\,s_1\ \ldots\ s_n$$

It is easy to prove that there can be no P that satisfies the equation in all instances. If this was the case then

$$\lceil M_{s_1\,s_2\,s_3\,s_4} \rceil\ =_\beta\ \lceil M_{s_1} \rceil\,s_2\,s_3\,s_4$$

Now assume that

$$\begin{array}{lcl} s_2 & \equiv & \lambda x.x \\ s_3 & \equiv & \lambda mn.(E\,m)\,(E\,n) \\ s_4 & \equiv & \lambda m.\lambda v.(E\,(m\,v)) \end{array}$$

where E is the self-interpreter. As $E\,\lceil N \rceil\ =_\beta\ N$, we see that

$$\lceil M_{s_1} \rceil\,s_2\,s_3\,s_4\ =_\beta\ M_{s_1}$$

Thus

$$\lceil M_{s_1\,s_2\,s_3\,s_4} \rceil\ =_\beta\ M_{s_1}\ =_\beta\ M\,s_1$$

for all M and s_1, which is clearly not possible.

The basic problem with the definition is that it is supposed to work with any number of static arguments. We could either fix that number (to one), or inform the partial evaluator of the number *before* the static arguments are given. A way to do this is to annotate M with information about which arguments are static. The annotated M, called M^{ann}, can be derived from the representation of M and the number of static arguments. We will look at this process later.

Another problem is that the definition requires P to be total in the sense that applying it to an annotated λ-term will always yield a normal form. This is a nice property, but by a reasoning similar to the above, we can show that this is not always possible. The problem is that no part of the static values can occur in the residual program, as that would entail converting these to their representations, which can be shown to be not generally possible. Our revised definition of partial evaluation is: a partial evaluator is a λ-term P, such that when there for λ-terms M, $s_1\ \ldots\ s_n$ exist a λ-term $M_{s_1\,\ldots\,s_n}$, such that

$$P\,\lceil M^{ann}\rceil\, s_1\,\ldots\,s_n \;=_\beta\; \lceil M_{s_1\ldots s_n}\rceil$$

then

$$M_{s_1\ldots s_n} \;=_\beta\; M\, s_1\,\ldots\,s_n$$

where M^{ann} is M annotated to accept n static arguments.

We can redefine partial evaluation in another way, which would allow totality. The idea is that the static arguments are given by their representations. This allows any part of them to occur in the residual program. The price for this is the extra cost of coding values by their representations. In self-application there will even be two levels of representation, as the representation of the representation of a λ-term is needed. Also, the problem of making a partial evaluator that is total and non-trivial is no mean task.

5 The partial evaluator

We adopt the basic ideas and notation from *lambda-mix*, as described in [Gomard 1989], [Gomard 1990], [Jones *et al.* 1990] and [Gomard and Jones 1991]: the syntax tree is annotated with binding times that describe whether an operation should be performed at partial evaluation time, or remain in the residual program. The operations that are performed (the *static* operations) are shown in normal syntax, and the residual (*dynamic*) operations are underlined. *lambda-mix* used an explicit application operator, but since we just use juxtaposition to indicate application we have no operator to underline. Instead, we underline the space between the function and its argument. An example is

$$\lambda x.\underline{\lambda} y.y\_(x\,y)$$

which indicates that the outermost abstraction will be reduced, but the innermost not. The application of x to y will be performed, but the application of y to the result of this will not. Note that variables are not annotated, it is only if a variable is used as a function that its binding time become apparent.

We can use higher order abstract syntax for annotated expressions also:

$$\begin{array}{lcl}
\lfloor x\rfloor & = & \mathit{Pvar}(x)\\
\lfloor M\,N\rfloor & = & \mathit{Sapp}(\lfloor M\rfloor,\lfloor N\rfloor)\\
\lfloor \lambda x.M\rfloor & = & \mathit{Sabs}(\lambda x.\lfloor M\rfloor)\\
\lfloor M\_N\rfloor & = & \mathit{Dapp}(\lfloor M\rfloor,\lfloor N\rfloor)\\
\lfloor \underline{\lambda} x.M\rfloor & = & \mathit{Dabs}(\lambda x.\lfloor M\rfloor)
\end{array}$$

Note that there is only one variable type, so we call it *Pvar* instead of *Svar* or *Dvar*. Partial evaluation of an annotated λ-term is in *lambda-mix* described by inference rules. Such inference systems are of an operational nature, and makes the reduction strategy explicit, which we want to avoid. We will describe it by a set of equations instead, similar to the way the self-interpreter was described:

$$\begin{array}{lcl}
P[\mathit{Pvar}(x)] & = & x\\
P[\mathit{App}(M,N)] & = & P[M]\;P[N]\\
P[\mathit{Abs}(M)] & = & \lambda v.P[(M\,v)]\\
P[\mathit{Dapp}(M,N)] & = & \mathit{App}(P[M],P[N])\\
P[\mathit{Dabs}(M)] & = & \mathit{Abs}(\lambda v.P[(M\;\mathit{Var}(v))])
\end{array}$$

The static operations are exactly as in the self-interpreter. The dynamic operations build expressions. These expressions are in the abstract syntax for unannotated λ-terms. Note that there is only one rule for variables: the rules for the corresponding abstractions make sure that the variables are bound to the right things. This has, however, the effect that the partial evaluator will only work on closed λ-terms. Free variables are *not* bound to anything, so applying P to a term with free variables can not reduce to a representation of any residual term. We could extend the annotated syntax to distuinguish free variables, but that would spoil our aim of maximal simplicity.

We can easily modify the representation schema for syntax to include annotated syntax:

$$\begin{array}{lcl}
\lceil x\rceil & \equiv & \lambda abcde.a\,x\\
\lceil M\,N\rceil & \equiv & \lambda abcde.b\,\lceil M\rceil\,\lceil N\rceil\\
\lceil \lambda x.M\rceil & \equiv & \lambda abcde.c\,(\lambda x.\lceil M\rceil)\\
\lceil M\_N\rceil & \equiv & \lambda abcde.d\,\lceil M\rceil\,\lceil N\rceil\\
\lceil \underline{\lambda} x.M\rceil & \equiv & \lambda abcde.e\,(\lambda x.\lceil M\rceil)
\end{array}$$

and use this to convert the partial evaluation equation system to a λ-term:

$$\begin{array}{lcll}
P & \equiv & Y\,\lambda p.\lambda m.m & (\lambda x.x)\\
 & & & (\lambda mn.(p\,m)\,(p\,n))\\
 & & & (\lambda m.\lambda v.p\,(m\,v))\\
 & & & (\lambda mn.\lambda abc.b\,(p\,m)\,(p\,n))\\
 & & & (\lambda m.\lambda abc.c\,(\lambda v.p\,(m\,(\lambda abc.a\,v))))\\
\text{where} & & & \\
Y & \equiv & \lambda h.(\lambda x.h\,(x\,x))\,(\lambda x.h\,(x\,x)) &
\end{array}$$

As an example, we take $M \equiv \mathit{Ackermann}$ and $s_1 \equiv 1_{\mathit{Church}}$, where *Ackermann* is Ackermann's function on Church numerals:

$$\begin{array}{lcll}
\mathit{Ackermann} & \equiv & \lambda m.\lambda n.m & (\lambda f.\lambda m.f\,(m\,f\,(\lambda x.x)))\\
 & & & (\lambda mfx.f\,(m\,f\,x))\\
 & & & n\\
\text{and} & & & \\
1_{\mathit{Church}} & \equiv & \lambda fx.f\,x &
\end{array}$$

The annotated version of *Ackermann* is:

$$\begin{array}{lcll}
\mathit{Ackermann}^{ann} & \equiv & \lambda m.\underline{\lambda} n.m & (\lambda f.\underline{\lambda} m.f\_(m\_f\_(\underline{\lambda} x.x)))\\
 & & & (\underline{\lambda} mfx.f\_(m\_f\_x))\_\\
 & & & n
\end{array}$$

and $\mathit{Ackermann}_{1_{\mathit{Church}}}$ is

$$\begin{array}{l}
\mathit{Ackermann}_{1_{\mathit{Church}}} \;\equiv\\
\quad \lambda n.\;((\lambda m.\;\;(\lambda mfx.f\,(m\,f\,x))\\
\qquad\qquad\qquad (m\,(\lambda mfx.f\,(m\,f\,x))(\lambda x.x))))\\
\qquad\quad n
\end{array}$$

Careful examination will show that only copies of the operations that were underlined occur in the residual program.

6 Compilation and self-application

Applying the partial evaluator to an annotated interpreter S^{ann} and a program p, will according to the definition, yield (the representation of) a λ-term r which is β-equivalent to $S\,p$. r is thus p compiled to lambda calculus. If S is the self-interpreter E, and p a representation of a λ-term M, r

should be β-equivalent to M. We actually get α-equivalence between r and M. This shows that the partial evaluator is non-trivial. The annotated self-interpreter is shown below.

$$\begin{array}{lll} E^{ann} & \equiv & Y\,\lambda e.\lambda m.m\ \ (\lambda x.x) \\ & & \qquad\qquad\quad (\lambda mn.(e\,m)\_\,(e\,n)) \\ & & \qquad\qquad\quad (\lambda m.\underline{\lambda} v.e\,(m\,v)) \\ \text{where} & & \\ Y & \equiv & \lambda h.(\lambda x.h\,(x\,x))\,(\lambda x.h\,(x\,x)) \end{array}$$

Note that only the application and the abstraction that simulate application and abstraction in the source program are dynamic. Hence it is no surprise that we get α-equivalence between the source program and the residual program.

If we apply the partial evaluator P to an annotated version P^{ann} of itself and an annotated interpreter S^{ann}, we expect a compiler $\lceil C \rceil =_\beta P_S$. That is, C should take a source program p and return (the representation of) the object program r. If S is E, then the compiler C_{self} should be an identity function on representations, and that is indeed what it turns out to be. Double self-application yields a compiler generator $\lceil G \rceil =_\beta P\,P^{ann}\,P^{ann})$: if G is applied to an interpreter, it returns the corresponding compiler. P^{ann} and C_{self} are shown below. G is shown in firgure1. The fact that G can even be shown in its entirety on half a page leads us to believe that it is indeed the smallest non-trivial automatically generated compiler generator yet in existence.

$$\begin{array}{lll} P^{ann} & \equiv & Y\,\lambda p.\lambda m.m\,(\lambda x.x) \\ & & \qquad\qquad\quad (\lambda mn.(p\,m)\_\,(p\,n)) \\ & & \qquad\qquad\quad (\lambda m.\underline{\lambda} v.p\,(m\,v)) \\ & & \qquad\qquad\quad (\lambda mn.\underline{\lambda} abc.b\_\,(p\,m)\_\,(p\,n)) \\ & & \qquad\qquad\quad (\lambda m.\underline{\lambda} abc.c\_\,(\underline{\lambda} v.p\,(m\,(\underline{\lambda} abc.a\_\,v)))) \\ \text{where} & & \\ Y & \equiv & \lambda h.(\lambda x.h\,(x\,x))\,(\lambda x.h\,(x\,x)) \end{array}$$

Note that the operations on static expressions are annotated as in the self-interpreter. In the operations on dynamic expressions all code-building is residualized.

$$\begin{array}{lll} C_{self} & \equiv & Y\,\lambda s.\lambda m.m\ \ (\lambda x.x) \\ & & \qquad\qquad\quad (\lambda mn.\lambda abc.b\,(s\,m)\,(s\,n)) \\ & & \qquad\qquad\quad (\lambda m.\lambda abc.c\,(\lambda v.s\,(m\,(\lambda abc.a\,v)))) \\ \text{where} & & \\ Y & \equiv & \lambda h.(\lambda x.h\,(x\,x))\,(\lambda x.h\,(x\,x)) \end{array}$$

It is interesting to see that the compiler handles expressions exactly as the partial evaluator handles *dynamic* expressions: the representation is rebuilt with only names of bound variables differing.

The compiler generator copies (rebuilds) the static operations in its input program (e.g. an interpreter) and generates code-building actions for the dynamic operations. So it generates representation of code that generates representation of code. This double representation level makes the code quite large and hard to read. To help a bit, the variables that become the variables for the representation schema in the generated compilers are given the names a', b' and c'.

It has been verified that applying G to P^{ann} yields G as the result.

7 Binding time analysis

In the previous section, we just pulled the annotated programs out of a hat. Here we describe how the annotation can be obtained automatically. We, again, use the same basic idea as in *lambda-mix* and base our analysis on type inference. Because of the different languages, our method is in some respects simpler and others more complex than the one described in [Gomard 1990].

It is simpler because we have fewer constructions in the language, i.e. we do not have atomic values, but it is more complex because we have to infer recursive types to get acceptable results.

We construct a type system such that annotated programs are well-typed if and only if the annotation is consistent. Furthermore, the type will reflect the number of static arguments. We start by describing the possible types:

$$\tau \ ::= \ d \ \mid \ \tau_1 \rightarrow \tau_2 \ \mid \ \alpha$$

d is a special type that describes dynamic values. In a dynamic application function and argument must both have this type. $\tau_1 \rightarrow \tau_2$ describes static functions: these must never appear as function or argument in a dynamic application, and whenever they appear as the function in a static application, their argument must have type τ_1, and the result of the application must have type τ_2. α is a free type variable. Values of this type will never appear as a function in an application, nor as argument to a dynamic application. These requirements and the consistency requirement are formalised in the inference rules shown below:

$$\rho \ \vdash \ x \ : \ (\rho\,x)$$

$$\frac{\rho[x \mapsto \tau_1] \ \vdash \ M \ : \ \tau_2}{\rho \ \vdash \ \lambda x.M \ : \ \tau_1 \rightarrow \tau_2}$$

$$\frac{\rho[x \mapsto d] \ \vdash \ M \ : \ d}{\rho \ \vdash \ \underline{\lambda} x.M \ : \ d}$$

$$\frac{\rho \ \vdash \ M \ : \ \tau_1 \rightarrow \tau_2 \quad \rho \ \vdash \ N \ : \ \tau_1}{\rho \ \vdash \ M\,N \ : \ \tau_2}$$

$$\frac{\rho \ \vdash \ M \ : \ d \quad \rho \ \vdash \ N \ : \ d}{\rho \ \vdash \ M\_N \ : \ d}$$

It was proven in [Gomard 1989] that there exist a minimal well-typed annotation for any program in the extended lambda calculus used in *lambda-mix*, in the sense that if some operations or types are *required* to be dynamic, there exist a well-typed annotation for the program where this is the case, and such that any other well-typed annotation would have *more* operators or types dynamic. This is called the *well-typed completion* of the requirements. This result carries immediately over to the system shown above, as it is a subset of the type system used in *lambda-mix*.

There are, however problems: if the type rules indicate that a type must contain itself as a proper part (the *occurs problem*), this is taken to indicate a type error. This is solved by forcing the type to be dynamic, and making the necessary adjustments to the annotation. This was no problem in *lambda-mix*, as the existence of data-structures and an explicit fixed-point combinator made such occurences rare. In the pure lambda-calculus, it has the consequence that the Y combinator cannot be given a static type. The annotations that we showed for the self-interpreter and the partial evaluator in the previous section would not be well-typed in such a system.

Our solution is not to treat the occurs problem as a type error, but instead assign a recursive type to the expression in

Figure 1: The compiler generator G.

$$
\begin{array}{lll}
G & \equiv & Y\,\lambda g.\lambda m.m\ \ (\lambda x.x) \\
 & & \qquad\qquad\quad (\lambda mn.\lambda abc.b\,(g\,m)\,(g\,n)) \\
 & & \qquad\qquad\quad (\lambda m.\lambda abc.c\,(\lambda v.g\,(m\,(\lambda abc.a\,v)))) \\
 & & \qquad\qquad\quad (\lambda mn. \\
 & & \qquad\qquad\quad\quad \lambda abc.c\,\lambda a'. \\
 & & \qquad\qquad\quad\quad\quad \lambda abc.c\,\lambda b'. \\
 & & \qquad\qquad\quad\quad\quad\quad \lambda abc.c\,\lambda c'. \\
 & & \qquad\qquad\quad\quad\quad\quad\quad \lambda abc.b\ (\lambda abc.b\,(\lambda abc.a\,b')\,(g\,m))\ (g\,n)) \\
 & & \qquad\qquad\quad (\lambda m. \\
 & & \qquad\qquad\quad\quad \lambda abc.c\,\lambda a'. \\
 & & \qquad\qquad\quad\quad\quad \lambda abc.c\,\lambda b'. \\
 & & \qquad\qquad\quad\quad\quad\quad \lambda abc.c\,\lambda c'. \\
 & & \qquad\qquad\quad\quad\quad\quad\quad \lambda abc. \\
 & & \qquad\qquad\quad\quad\quad\quad\quad\quad b\,(\lambda abc.a\,c') \\
 & & \qquad\qquad\quad\quad\quad\quad\quad\quad\ \ (\lambda abc.c\,(\lambda v. \\
 & & \qquad\qquad\quad\quad\quad\quad\quad\quad\quad\quad g\,(m \\
 & & \qquad\qquad\quad\quad\quad\quad\quad\quad\quad\quad\quad \lambda abc.c\,(\lambda a'. \\
 & & \qquad\qquad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \lambda abc.c\,\lambda b'. \\
 & & \qquad\qquad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \lambda abc.c\,\lambda c'. \\
 & & \qquad\qquad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \lambda abc.b\ (\lambda abc.a\,a')\ (\lambda abc.a\,v)))))) \\
\text{where} & & \\
Y & \equiv & \lambda h.(\lambda x.h\,(x\,x))\,(\lambda x.h\,(x\,x))
\end{array}
$$

question. We thus extend our type system to include types of the form

$$\tau \ ::= \ \mu\alpha.\tau$$

we do not need to change the inference rules if we use the equivalence:

$$\mu\alpha.\tau \ = \ \tau[\alpha := \mu\alpha.\tau]$$

Intuitively, recursive types are considered equivalent to the infinite types obtained by unfolding the recursion. It is easy to see that the recursive types allow any two types not containing d to be unified. Hence any λ-term can be given a completely static type and thus also a completely static annotation. Once we introduce d into the types, there can be conflicts between d and function types, making some annotations not well-typed. Note, however, that the annotation where all operations are dynamic is always well-typed with the type d. We conjecture that there exists a minimal (least dynamic) well-typed completion for any binding time requirements for any λ-term, just as in *lambda-mix*.

Binding time analysis consists of finding a well-typed completion of an initially completely static annotation and a type of the form

$$\tau_1 \to \ldots \to \tau_n \to d$$

This type indicates that there are n static arguments with types $\tau_1 \ldots \tau_n$, and the result is dynamic. The τ_i can be type variables, which the binding time completion will bind to the necessary types. The completion might require some of the τ_i to unify with dynamic. This means that some of the static arguments appears in contexts where they are required to be dynamic. Since we cannot convert a value to its representation inside the lambda calculus, this means that we will not be able to do partial evaluation. Note that a static value can have the type $d \to d$, without a conflict. An example is if the value is $\lambda x.x$, and it is applied to a dynamic value. The function is applied and reduced to its argument, so no conversion is necessary. In general, there is no problem if the static argument can be given a type τ_i containing free type variables, and only these are replaced by (types containing) d by the binding type completion. If τ_i in the initial binding time specification is sufficiently specific, the completion will fail if a static value is used in a dynamic context. Note that this is not in conflict with the conjecture that a minimal completion always exists. In the general case, a completion can make the type of the term more dynamic, which might force some arguments to be dynamic.

When doing binding time analysis of the self-interpreter E, we want to restrict ourselves to the case where the first argument is the representation of a λ-term. Thus we specify the type

$$(\mu\alpha.(\beta \to \beta) \to (\alpha \to \alpha \to \beta) \to ((\beta \to \alpha) \to \beta) \to \beta) \ \to \ d$$

The recursive type is the type of representations, and the d specifies that there are no further static arguments. There is in fact only one well-typed completion of E with this type. This is the one that was shown in the previous section. The free type variable β is unified with d during completion, but this just means that parts of the static argument are applied to dynamic arguments. If we had not specified the type of the static argument, there would be 5 different completions. Two of these have more static types than the one shown, one having also a more static annotation with only the application (and not the abstraction) dynamic. The types are similar to the type of representations, but have only some of the occurrences of β replaced by d. The requirement that all occurences are the same, disallows this completion when the type is specified. The other two completions have less static types and annotations. The types of these have d's in places where the type of representations have function types.

Similarly, binding time analysis of the partial evaluator P starts with the type

$$(\mu\alpha.(\beta \to \beta) \to (\alpha \to \alpha \to \beta) \to ((\beta \to \alpha) \to \beta) \to (\alpha \to \alpha \to \beta) \to ((\beta \to \alpha) \to \beta) \to \beta) \to d$$

and gives the sole well-typed completion of P. The recursive type is now the type of representations of annotated expression. If the type is not specified a total of 1505 completions are possible.

A completion algorithm has been implemented in Prolog. It implements the recursive types as cyclic data-structures to make unification of types possible by Prologs unification (without occurs check). The input and output use explicit recursion with μ for reading and printing types. These are converted to and from the cyclic data structures. The annotation is implemented by having binding-time fields in the λ-term syntax. These are initially free variables, but become bound when one of the inference rules are used. If a conflict occurs, backtracking will try another inference rule giving another annotation. By arranging the rules such that the static rules are used before the corresponding dynamic rules, the first answer will be the minimal completion. This is not the most efficient way of obtaining the minimal completion, but it suffices for terms as small as those shown here. See [Henglein 1992] for an efficient completion algorithm.

8 Conclusion

We have presented an extremely small self-applicable partial evaluator for pure lambda calculus. Apart from the binding time analysis everything is done inside the lambda calculus itself. This could be done, but it would not be a trivial task, as unification and backtracking would have to be implemented in lambda calculus.

We have only briefly touched the topic of termination of partial evaluation. The presented system is not guaranteed to terminate even when well-typed annotations exist. For example, if the term contains a fixed-point combinator

$$Y \equiv \lambda h.(\lambda x.h\,(x\,x))\,(\lambda x.h\,(x\,x))$$

applied to a dynamic argument, the minimal completion is

$$Y^{ann} \equiv \lambda h.(\lambda x.h\_(x\,x))\,(\lambda x.h\_(x\,x))$$

With this annotation the recursion would be unfolded infinitely during partial evaluation. If we disallowed recursive types, partial evaluation of well-annotated terms would be guaranteed to terminate. However, *any* use of a fixed-point combinator would be annotated as dynamic, forcing *e.g.* the program argument to E to be dynamic. This is clearly not satisfactory. A possible solution would be to restrict the use of recursive types to cases where termination is guaranteed. It is not obvious how to do this.

References

[Dybvig 1987] Dybvig, R.K. 1987. The Scheme Programming Language, Prentice-Hall.

[Gomard 1989] Gomard, C. 1989. Higher Order Partial Evaluation - HOPE for the Lambda Calculus, *Masters Thesis*, DIKU, University of Copenhagen, Denmark.

[Gomard 1990] Gomard, C. 1990. Partial Type Inference for Untyped Functional Programs, *Proceedings of the ACM Conference on Lisp and Functional Programming 1990*, ACM Press: 282-287.

[Gomard and Jones 1991] Gomard, C. and Jones, N.D. 1991. A Partial Evaluator for the Untyped Lambda Calculus, *Journal of Functional Programming*, Volume 1, Part 1,: 21-69.

[Henglein 1992] Henglein, F. 1992, Dynamic Typing, *To appear in Proc. European Symp. on Programming (ESOP) 1992.*

[Jones *et al.* 1990] Jones, N.D., Gomard, C.K., Bondorf, A., Danvy, O., Mogensen, T. 1990. A Self-Applicable Partial Evaluator for the Lambda Calculus. *Proceedings of the 1990 International Conference on Computer Languages*, IEEE Computer Society Press: 49-58.

[Mogensen 1992] Mogensen, T. Efficient Self-interpretation in Lambda Calculus, to appear in *Journal of Functional Programming.*

[Pfenning and Elliot 1988] Pfenning, F. and Elliot, C. 1988. Higher-Order Abstract Syntax, *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, ACM Press: 199-208.

[Reynolds 1985] Reynolds, J.C. 1985. Three Approaches to Type Structure, *Lecture Notes in Computer Science* Volume 185, Springer-Verlag: 97-138.