Controlled Self-Applicable On-Line Partial Evaluation, Using Strategies

Mattox Beckman* Sam Kamin†

Department of Computer Science
University of Illinois
Urbana, IL 61801
{beckman,kamin}@cs.uiuc.edu

Abstract

On-line partial evaluators are hardly ever self-applicable, because the complexity of deciding whether to residualize terms causes combinatorial explosion when self-application is attempted. Recently, T. Mogensen found a way to write a self-applicable on-line partial evaluator for λ-calculus. His method is to regard *every* λ-term as having both static and dynamic aspects; then, applications can always be done statically (using the static aspect of the operator). However, the absence of decision-making about residualization has a down side: his partial evaluator knows only how to fully reduce partially evaluated terms. The result is considerable code explosion. We show how this problem can be overcome, in part, by changing the type of the partial evaluator, and giving a new version of the Futamura projections to correspond to that new type. Specifically, we have the partial evaluator take a third argument, called a *strategy*, which "advises" the partial evaluator whether to residualize. Strategies allow the programmer to control the tradeoff between the size of a specialized term and the cost of subsequently applying it. We present a number of strategies and examples of each.

Keywords: Partial evaluation, on-line partial evaluation, lambda calculus

1 Introduction

In the world of partial evaluators, there are *on-line* partial evaluators and then there are *self-applicable* partial evaluators, and there is little overlap between them. On-line partial evaluators contain explicit tests for staticness, and these tests tend to cause combinatorial explosion when the partial evaluator is self-applied.[3, 6]

Recently, Mogensen [4] found an ingenious solution for creating a self-applicable on-line partial evaluator for λ-calculus. The idea is to interpret every λ-expression as a pair of values: an equivalent λ-expression and a function, essentially representing the application of that λ-expression by β-reduction. Applications are evaluated by applying the function part of the operator to the operand. We call this *static* application, since it corresponds to partial evaluation-time β-reduction.

The only problem with Mogensen's solution is that the results of self-applications of his partial evaluator are extremely large λ-expressions, because λ-terms are fully normalized. Mogensen states, "While self-application has been achieved, the results are not perfect. P_{gen} is 136 times larger than P, which is excessive. The large size is mainly due to code duplication cause by uncritical reduction of nonlinear redexes." Some applications should be performed *dynamically*, by creating an application of the first component of the operator to the first component of the operand and doing no further reductions. However, Mogensen's partial evaluator does no dynamic applications, and a straightforward attempt to have it do so is doomed to fail, because of the danger of combinatorial explosion. That is, to have it *decide* at partial evaluation time whether to perform each application statically or dynamically would lead to combinatorial explosion.

In this paper, we present an approach to solving this problem. Our main idea is to give a new version of the Futamura projections. In this version, partial evaluators have *three* arguments. In addition to the traditional two — a binary function and its first argument — there is a third argument, called a *strategy*. The strategy is responsible for deciding whether to perform static or dynamic applications. Since it is not itself part of the partial evaluator, it can make

*Partially supported by NASA grant NAG–1–613
†Partially supported by NSF grant CCR 93–03043

$$
\begin{array}{ll}
(1) & P\,\lceil M\rceil\,\lceil N\rceil \;=\; \lceil M_N\rceil, \\
 & \text{where } M_N\,X \;=_\beta\; M\,N\,X \\
(2) & P\,\lceil P\rceil\,\lceil\lceil M\rceil\rceil \;=\; \lceil M_{gen}\rceil \\
 & \text{where } M_{gen}\,\lceil N\rceil \;=_\beta\; M_N \\
(3) & P\,\lceil P\rceil\,\lceil\lceil P\rceil\rceil \;=\; \lceil P_{gen}\rceil \\
 & \text{where } P_{gen}\,\lceil\lceil M\rceil\rceil \;=_\beta\; M_{gen}
\end{array}
$$

Figure 1: Original Futamura Projections

$$
\begin{array}{ll}
(1) & P\,\lceil M\rceil\,\lceil N\rceil\,\Sigma \;=\; \lceil M_N\rceil, \\
 & \text{where } M_N\,X \;=_\beta\; M\,N\,X \\
(2) & P\,\lceil P\rceil\,\lceil\lceil M\rceil\rceil\,\Sigma \;=\; \lceil M_{gen}\rceil \\
 & \text{where } M_{gen}\,\lceil N\rceil\,\Sigma' \;=_\beta\; \lceil M_N\rceil \\
(3) & P\,\lceil P\rceil\,\lceil\lceil P\rceil\rceil\,\Sigma \;=\; \lceil P_{gen}\rceil \\
 & \text{where } P_{gen}\,\lceil\lceil M\rceil\rceil\,\Sigma' \;=_\beta\; \lceil M_{gen}\rceil
\end{array}
$$

Figure 2: Futamura Projections with Strategies

arbitrarily complex decisions without contributing to combinatorial explosion when the partial evaluator is self-applied.

The standard Futamura projections for the λ-calculus[4] are in Figure 1[1]. The modified projections are in Figure 2. Though they look nearly identical, there is a big difference: With the standard version, the partial evaluator, P, is responsible for making all static-vs.-dynamic application decisions. This expands the size of P, so that if these decisions are at all complicated, self-application becomes impossible. Accordingly, self-application is achieved in practice only when arguments to P are pre-processed so that P's decision-making is very simple. Our version allows P to off-load such decisions onto the strategy argument, allowing them to be made intelligently without compromising the self-applicability of P.

Strategies fit neatly into Mogensen's approach. Since the pair of values associated with a λ-expression contains both an expression and a function, the decision of whether to apply statically or dynamically corresponds to a decision of which part of the pair to use. The strategy argument is invoked whenever the partial evaluator does an application, and it makes that decision. Providing different strategies permits a trade-off between the size of residual terms and their efficiency.

[1] The $\lceil\cdot\rceil$ notation indicates a conversion to higher order abstract syntax; a specific definition is given in section 2. The double encoding seen in the second and third projection occurs because the first argument is itself a partial evaluator and expects its input to be in higher order abstract syntax.

We have not, unfortunately, been able to solve the combinatorial explosion problem in precisely the way that Mogensen suggested. Since Mogensen's approach relies heavily on the use of higher-order abstract syntax, it is impossible to count occurrences of variables, so we cannot determine which redexes are non-linear. However, we have a variety of strategies that give interesting results.

Strategies are a kind of program annotation directing the actions of the partial evaluator. The use of such annotations is far from new — indeed, the earliest self-applicable partial evaluators required explicit static/dynamic annotations [2]. More sophisticated annotations for practical partial evaluation have been studied by several researchers [1, 6, 7], particularly in the context of on-line partial evaluation. The contribution of this paper is not in the particular strategies we present — indeed, our strategies are rather weak, being constrained by the higher-order abstract syntax representation of terms. Rather, it is the way our strategies are incorporated into the partial evaluator so as not to interfere with self-application.

In the next section, we review Mogensen's results from [4]. We then present the idea of strategies, and our strategy-based partial evaluator in section 3. In section 4, we discuss self-application of our partial evaluator. In section 5, we change the representation of λ-expressions slightly, in a way that permits the definition of more strategies. In the final section, we show that the results of binding-time analysis can be incorporated into a strategy-based *off-line* partial evaluator, demonstrating that our approach in a sense subsumes the BTA-based partial evaluators; we do this with yet another change of representation, in which expressions contain annotations which are written by a pre-processor and read by the strategy.

2 Mogensen's result

Mogensen's self-applicable on-line partial evaluator is most easily understood by considering each λ-expression to denote a value that contains both static (function) and dynamic (expression) parts. We call such values "partial evaluation values," or PEV's:

$$PEV = (PEV \to PEV) \times Expr$$

The type $Expr$ contains abstract syntax trees (AST's) representing λ-expressions. Without saying exactly how AST's are represented (for now), we will just postulate the existence of abstract syntax operators App, Abs and Var. App takes two AST's to the AST representing their application, and Var takes a variable to an AST. Abs takes an *abstraction* mapping

λ-terms to AST's, and transforms it into an AST. This contrasts with the usual approach in which *Abs* takes as arguments a variable and an AST and returns an AST. The difference is just the difference between the higher-order abstract syntax used here and the first-order abstract syntax usually used [5].

λ-expressions are translated to PEV's by the following rules. Note that these rules apply only to closed expressions, so the translation of variables is technically "auxiliary." In particular, $[\![x]\!] = x$ is not a PEV but, given this rule, the translations of applications and abstractions to produce PEV's.

$$\begin{array}{lcl}
[\![\cdot]\!] : \Lambda & \rightarrow & PEV \\
 & & \\
[\![x]\!] & \mapsto & x \\
[\![m\ n]\!] & \mapsto & [\![m]\!]_1\ [\![n]\!] \\
[\![\lambda x.e]\!] & \mapsto & \text{let } g\ =\ \lambda x.[\![e]\!] \\
 & & \quad \text{in} < g,\ Abs(\lambda w.(g\ (D(Var(w))))_2) >
\end{array}$$

$$\text{where } D : Expr \rightarrow PEV = \lambda e. < \lambda v.D(App(e, v_2)),\ e >$$

The translation of abstractions merits some explanation. An abstraction $\lambda x.e$ is easily translated to a function from PEV's to PEV's, namely $g = \lambda x.[\![e]\!]$. The problem is to turn that function into a PEV. The function component[2] of the PEV is g itself; the difficult part is constructing the expression component, that is, going from a *function* to an *expression*. This is done by applying g to the PEV representing a variable ($D(Var(w))$) and abstracting that variable ($Abs(\lambda w. \ldots)$). To turn the variable $Var(w)$ into a PEV, we apply D, a function that takes any expression to a PEV. $D(e)$ has e as its expression component and, for its function component, has the function $F : PEV \rightarrow PEV$ that "textually" applies e. That is, given any $PEV < \ldots,\ e' >$, F returns the $PEV < F', App(e, e') >$, where F', when applied to $PEV < \ldots, e'' >$, returns $< \ldots, App(App(e, e'), e'') >$, and so on.

A term is reduced by projecting the expression component of this translation. The partial evaluator creates an application node from its two arguments and normalizes it.

$$\begin{array}{l}
\mathit{normalize}\ e = [\![e]\!]_2 \\
PE\ m\ n = \mathit{normalize}(m\ n)
\end{array}$$

The following reduction sequence illustrates the operation of the partial evaluator.

[2] We will often call the first and second components of a PEV the function and expression components, respectively.

$$\begin{array}{l}
PE\ (\lambda x.((\lambda y.y\ y)\ (\lambda z.z)))\ (\lambda q.q) \\
\rightarrow\ \mathit{normalize}\ ((\lambda x.((\lambda y.y\ y)\ (\lambda z.z)))\ (\lambda q.q)) \\
\rightarrow\ [\![(\lambda x.((\lambda y.y\ y)\ (\lambda z.z)))\ (\lambda q.q)]\!]_2 \\
\rightarrow\ ([\![(\lambda x.((\lambda y.y\ y)\ (\lambda z.z)))]\!]_1\ [\![(\lambda q.q)]\!])_2 \\
\rightarrow\ ((\lambda x.[\![((\lambda y.y\ y)\ (\lambda z.z))]\!])\ [\![(\lambda q.q)]\!])_2 \\
\rightarrow\ ([\![((\lambda y.y\ y)\ (\lambda z.z))]\!])_2 \\
\rightarrow\ ([\![(\lambda y.y\ y)]\!]_1\ [\![(\lambda z.z)]\!])_2 \\
\rightarrow\ ((\lambda y.[\![y\ y]\!])\ [\![(\lambda z.z)]\!])_2 \\
\rightarrow\ ((\lambda y.[\![y]\!]_1\ [\![y]\!])\ [\![(\lambda z.z)]\!])_2 \\
\rightarrow\ ((\lambda y.y_1\ y)\ [\![(\lambda z.z)]\!])_2 \\
\rightarrow\ ([\![(\lambda z.z)]\!]_1\ [\![(\lambda z.z)]\!])_2 \\
\rightarrow\ ((\lambda z.[\![z]\!])\ [\![(\lambda z.z)]\!])_2 \\
\rightarrow\ ((\lambda z.z)\ [\![(\lambda z.z)]\!])_2 \\
\rightarrow\ ([\![(\lambda z.z)]\!])_2 \\
\rightarrow\ Abs(\lambda w.((\lambda z.z)(D(Var(w))))_2) \\
\rightarrow\ Abs(\lambda w.(D(Var(w)))_2) \\
\rightarrow\ Abs(\lambda w.Var(w))
\end{array}$$

Mogensen shows how to render these ideas into λ-calculus. First, we need to represent the abstract syntax of λ-terms within λ-calculus. A λ-term e is translated to an $Expr$ by the function $\lceil \cdot \rceil$, defined as:

$$\lceil e \rceil\ =\ \lambda a \lambda b.\bar{e}, \text{ where } \begin{array}{lcl}
\overline{x} & = & x \\
\overline{m\ n} & = & a\ \overline{m}\ \overline{n} \\
\overline{\lambda x.e} & = & b\ (\lambda x.\overline{e})
\end{array}$$

Then, an $Expr$ is translated to a PEV by applying it to terms A and B:

$$\begin{array}{lcl}
A & = & \lambda p.\lambda p'.p_1\ p' \\
B & = & \lambda g. < g, (\lambda ab.b\ (\lambda w.(g\ (D\ \lambda ab.w))_2\ a\ b)) > \\
D & = & Y\ \lambda d.\lambda m. < m, (\lambda v.d\ (\lambda ab.a\ (m\ a\ b)\ (v_2\ a\ b))) >
\end{array}$$

In other words, $[\![e]\!] = \lceil e \rceil\ A\ B$. In the implementation, it is necessary to convert the arguments of the partial evaluator into higher order abstract syntax first. So, while $[\![\cdot]\!]$ operates on terms of type Λ, the λ-term corresponding to it in our implementation, which we call `P`, operates on $Expr$'s.

The code for the complete partial evaluator is given in Figure 3, in the form of the input to our top-level λ-calculus reducer. Backslash represents λ, semicolon represents the end of a term, and colon-equal assigns values into an environment (but only at the top level). Note the λ-calculus coding of pairs $< a, b >$ as functions $\lambda sel.sel\ a\ b$.

In Figure 3, the D term has been rewritten as a self-application of a similar term Q. This has the effect of delaying recursion until all the arguments to D are known, and eliminating the need for the Y combinator. This was necessary since an explicit use of

```
P := \m n. R (\a b. a (m a b) (n a b));
R := \m. m A B F;
A := \m n. m T n;
B := \g. \sel. sel
        g (\a b. b (\w. g (D \a b. w) F a b));
D := Q Q;
Q := \q. \m.
     (\sel. sel (\qq. \v.
                   qq qq (\a b. a (m a b)
                                  (v F a b)))
                (\qq. m)
                     q);
F := (\fst snd. snd);
T := (\fst snd. fst);
```

Figure 3: Mogensen's partial evaluator

Expression	Size	β	ρ
$Ack\ 0\ 3 \to 4$		7	
$Ack\ 1\ 3 \to 5$		19	
$Ack\ 2\ 3 \to 9$		76	
$P\ \lceil Ack \rceil\ \lceil 0 \rceil \to \lceil Ack_0 \rceil$	10	119	
$P\ \lceil Ack \rceil\ \lceil 1 \rceil \to \lceil Ack_1 \rceil$	24	253	
$P\ \lceil Ack \rceil\ \lceil 2 \rceil \to \lceil Ack_2 \rceil$	52	512	
$Ack_0\ 3 \to 4$		3	2.3
$Ack_1\ 3 \to 5$		13	1.5
$Ack_2\ 3 \to 9$		65	1.2
$P\ \lceil P \rceil\ \lceil\lceil Ack \rceil\rceil \to \lceil Ack_{gen} \rceil$	693	8095	
$Ack_{gen}\ \lceil 0 \rceil \to \lceil Ack_0 \rceil$	10	56	2.1
$Ack_{gen}\ \lceil 1 \rceil \to \lceil Ack_1 \rceil$	24	94	2.7
$Ack_{gen}\ \lceil 2 \rceil \to \lceil Ack_2 \rceil$	52	180	2.8
$P\ \lceil P \rceil\ \lceil\lceil P \rceil\rceil \to \lceil P_{gen} \rceil$	17819	268481	

Table 1: Excerpt from Table 2 of [4]

Y will result in a term with no normal form, and will cause non-termination when the partial evaluator is self-applied. This rewriting trick will show up in our partial evaluator as well.

The ability to perform partial evaluation without needing to decide explicitly whether or not a term is static makes self-application feasible. Mogensen is able to self-apply P, according to the Futamura projections in Figure 1. The idea, as usual, is that when M is a 2-argument function and N is its first argument, $P\lceil M \rceil \lceil N \rceil$ returns the representation $\lceil M_N \rceil$ of an expression M_N. For any X, $M_N\ X =_\beta M\ N\ X$, but $M_N\ X$ normalizes more quickly than $M\ N\ X$.

Mogensen uses as his main example this definition of the Ackermann function:

$$\begin{aligned} Ack \ = \ & \lambda mn.m\ (\lambda fm.f\ (m\ f\ (\lambda x.x))) \\ & (\lambda mfx.f\ (m\ f\ x))\ n \end{aligned}$$

Some of his results are summarized in Table 1. The ρ column indicates the speedup, measured in β-reductions, obtained by applying $P\lceil M \rceil \lceil N \rceil$ to an argument X, as compared to calculating $M\,N\,X$ directly. For example, it takes 2.7 times as many β-reductions to produce Ack_1 by calculating $P\ \lceil Ack \rceil\ \lceil 1 \rceil$ as it does to produce it by applying Ack_{gen} to $\lceil 1 \rceil$.

3 Strategies

Mogensen's partial evaluator, being on-line, does not have the benefit of a binding-time analysis to tell it when to statically or dynamically apply. If it had to make such decisions itself, it would suffer combinatorial explosion. So it just β-reduces everything. This simplification makes self-application possible, but also results in very large residual terms, as seen in Table 1. In short, his partial evaluator has no effective control of the partial evaluation process.

We provide this control by giving the partial evaluator a third argument, called a *strategy*. The strategy tells the partial evaluator what to do when one PEV is applied to another. The definition of PEV changes to:

$$\begin{aligned} PEV &= Strategy \to Result \\ Result &= (PEV \to Result) \times Expr \\ Strategy &= PEV \to PEV \to Result \end{aligned}$$

Again, a λ-term is translated into a PEV. When given a strategy — indicating how to handle applications that arise when normalizing this term — the term returns a pair similar to the pairs described in the previous section.

The translation of λ-terms needs to be modified. As above, $[\![e]\!]$ is the PEV associated with e:

$$\begin{aligned} [\![x]\!] &\mapsto x \\ [\![m\ n]\!] &\mapsto \lambda\Sigma.\ \Sigma\ [\![m]\!]\ [\![n]\!] \\ [\![\lambda x.e]\!] &\mapsto \text{let } g = \lambda x.([\![e]\!]\ \Sigma) \text{ in} \\ & \qquad < g,\ Abs(\lambda w.(g\ (D(Var(w)))\ \Sigma)_2) > \end{aligned}$$
$$\text{where } D\ e\ \Sigma = < Abs(\lambda v.D(App(e, (v\ \Sigma)_2))\ \Sigma),\ e >$$

The partial evaluator is modified so as to take a third argument:

$$\begin{aligned} normalize\ e\ \Sigma &= ([\![e]\!]\ \Sigma)_2 \\ PE\ m\ n\ \Sigma &= normalize\ (m\ n)\ \Sigma \end{aligned}$$

What does a strategy look like? Basically, given PEV π to be applied to PEV π', a strategy can either statically apply π — which means applying the function component of π to π' — or dynamically apply it — meaning create an application node containing the expression components of π and π'. The simplest example is the strategy that uniformly performs static application:

$$\Sigma_{all} \;=\; \lambda\pi.(\pi\ \Sigma_{all})_1$$

That is, given PEV's π and π', Σ_{all} supplies itself to π (this says that the operator should be normalized using the Σ_{all} strategy), which produces a *Result*. It then selects the function component of this *Result* to apply to π'.

When Σ_{all} is used, the results are identical to those of Mogensen. To give a simple example:

$$\begin{aligned} PE\quad & (\lambda x.((\lambda y.y\ y)\ (\lambda z.z)))\ (\lambda q.q)\ \Sigma_{all} \\ =\ & Abs(\lambda z.z) \\ =\ & \lceil \lambda z.z \rceil \end{aligned}$$

Here is the first half of the reduction sequence. It shows how Σ_{all} forces the outer β-reduction to be done — eliminating the $\lambda q.q$ term — and is then used as the strategy for reduction of the resulting term:

$$\begin{aligned} & PE\ (\lambda x.((\lambda y.y\ y)\ (\lambda z.z)))\ (\lambda q.q)\ \Sigma_{all} \\ \rightarrow\ & normalize\ ((\lambda x.((\lambda y.y\ y)\ (\lambda z.z)))\ (\lambda q.q))\ \Sigma_{all} \\ \rightarrow\ & (\llbracket(\lambda x.((\lambda y.y\ y)\ (\lambda z.z)))\ (\lambda q.q)\rrbracket \Sigma_{all})_2 \\ \rightarrow\ & (\Sigma_{all}\ \llbracket(\lambda x.((\lambda y.y\ y)\ (\lambda z.z)))\rrbracket\ \llbracket(\lambda q.q)\rrbracket)_2 \\ \rightarrow\ & ((\llbracket(\lambda x.((\lambda y.y\ y)\ (\lambda z.z)))\rrbracket\ \Sigma_{all})_1\ \llbracket(\lambda q.q)\rrbracket)_2 \\ \rightarrow\ & ((\lambda x.\llbracket((\lambda y.y\ y)\ (\lambda z.z))\rrbracket\ \Sigma_{all})\ \llbracket(\lambda q.q)\rrbracket)_2 \\ \rightarrow\ & (\llbracket((\lambda y.y\ y)\ (\lambda z.z))\rrbracket\ \Sigma_{all})_2 \\ \rightarrow\ & \cdots \\ \rightarrow\ & Abs(\lambda z.Var(z)) \end{aligned}$$

We give examples of the use of our partial evaluator on the *Ack* function in Table 2; these should be compared to Mogensen's results in Table 1. For instance, we see from Table 1 that in Mogensen's partial evaluator, $P\lceil Ack\rceil\lceil 2\rceil$ yields Ack_2, and that $Ack_2\,3$ reduces to 9 in 65 β-reductions. In the top part of Table 2 we see the analogous results from our partial evaluator. Specifically, $P\lceil Ack\rceil\lceil 2\rceil\,\Sigma_{all}$, which we call $Ack_{2,all}$, corresponds directly to Mogensen's Ack_2; the table shows that $Ack_{2,all}\,3$ reduces to 9 in 65 β-reductions.

Another simple strategy is not to β-reduce at all, but to do only dynamic applications:

$$\Sigma_{none} \;=\; \lambda\pi.(D\ ((\pi\ \Sigma_{none})_2)\ \Sigma_{none})_1$$

Given π and π', Σ_{none} (also denoted as Σ_0) first reduces the operator π, using the non-reducing strategy, then takes the expression components of the result and turns it into a PEV by applying D. Recall that D produces a PEV that always does dynamic application (to any number of arguments).

The effect of Σ_{none} is to form an application node and otherwise do no reduction. For example,

$$\begin{aligned} PE\quad & (\lambda x.((\lambda y.y\ y)\ (\lambda z.z))\ (\lambda q.q))\ \Sigma_{none} \\ =\ & App(Abs(\lambda x.App(Abs(\lambda y.App(y,y)), \\ & \quad Abs(\lambda z.z))), Abs(\lambda q.q)) \\ =\ & \lceil(\lambda x.((\lambda y.y\ y)\ (\lambda z.z))\ (\lambda q.q))\rceil \end{aligned}$$

The first four steps of the reduction sequence are similar to that of Σ_{all}:

$$\begin{aligned} & PE\ (\lambda x.((\lambda y.y\ y)\ (\lambda z.z)))\ (\lambda q.q)\ \Sigma_{none} \\ \rightarrow\ & normalize\ ((\lambda x.((\lambda y.y\ y)\ (\lambda z.z)))\ (\lambda q.q))\ \Sigma_{none} \\ \rightarrow\ & (\llbracket(\lambda x.((\lambda y.y\ y)\ (\lambda z.z)))\ (\lambda q.q)\rrbracket \Sigma_{none})_2 \\ \rightarrow\ & (\Sigma_{none}\ \llbracket(\lambda x.((\lambda y.y\ y)\ (\lambda z.z)))\rrbracket\ \llbracket(\lambda q.q)\rrbracket)_2 \\ \rightarrow\ & ((D((\llbracket(\lambda x.((\lambda y.y\ y)\ (\lambda z.z)))\rrbracket\ \Sigma_{none})_2)\ \Sigma_{none})_1 \\ & \llbracket(\lambda q.q)\rrbracket)_2 \end{aligned}$$

At this point, D forms a residual expression out of its argument and residualizes future applications of that argument.

To again compare our results to Mogensen's directly, if we denote $P\ [Ack]\ [2]\ \Sigma_{none}$ by $Ack_{2,none}$, we see in Table 2 that $Ack_{2,none}$ 3 reduces to 9 after 76 β-reductions, compared to the 65 for $Ack_{2,all}$. Of course, it takes fewer β-reductions to calculate $Ack_{2,none}$ and, perhaps more importantly, $Ack_{2,none}$ is smaller. Thus, we begin to see in the first few lines of Table 2 how strategies allow us to trade speed for size (and partial evaluation time).

We can get intermediate levels of expansion. Strategy $\Sigma_n\ n$ performs the first n β-reductions (for some n) and then no more.

$$\Sigma_n \;=\; \lambda n.n\ (\lambda \mathrm{E}.\lambda\pi.(\pi\ \mathrm{E})_1)\ \Sigma_{none}$$

Here, n is a church numeral which will cause the λE term to be applied n times, followed by Σ_{none}. For example,

$$\begin{aligned} PE\quad & (\lambda x.((\lambda y.y\ y)\ (\lambda z.z))\ (\lambda q.q))\ (\Sigma_n\ 1) \\ =\ & App(Abs(\lambda y.App(y,y)), Abs(\lambda z.z)) \\ =\ & \lceil((\lambda y.y\ y)\ (\lambda z.z))\rceil \end{aligned}$$

In other words, the evaluator did one β-reduction — the outer one — and then stopped.

Table 2 shows results using Σ_n with *Ack*, for n = 1, 4, and 8. We can see that as we go from (Σ_1) up to (Σ_8), both the cost of computing the residual function and its size increase, while the cost of applying that residual function decreases. In particular, the efficiency of these residuals fall between those of $Ack_{2,none}$ and $Ack_{2,all}$.

Another "counting" strategy is the one that abstains from doing top-level reductions, but does reductions within subexpressions:

```
P := \m n s. R (\a b. a (m a b) (n a b)) s F;
R := \m. m A B;
B := \g s. (\ sel. sel
          (\x. g x s)
          (\a b. b (\z. (snd (g (D (\a b. z)) s))
                         a b)));
A := \m n s. s m n;
D := (DQ DQ);
DQ := \q. \v. \s. (\sel. sel
        (\qq. \w. (qq qq)
                    (\a b. a (v a b)
                            (fst (w s) a b)) s)
        (\qq. v)
       q);

fst := \pair. pair (\fst snd. fst);
snd := \pair. pair (\fst snd. snd);
T := \fst snd. fst;
F := \fst snd. snd;
Y := (\h. (\x. h (x x)) (\x. h (x x)));

E_all := Y (\E. \pi. (fst (term E)));
E_none := Y (\E. \pi. fst
               (D (snd (term E)) E));
E_n := \n. n (\E. \pi. fst (term E)) E_none;
E_below-n := \ n. n (\E. \pi. fst
               (D (snd (term E)) E)) E_all;
```

Figure 4: Strategy-based partial evaluator

$$\Sigma_{below,n} \;=\; \lambda n.n\ (\lambda \mathrm{E}.\lambda\pi.(D\ (\pi\ \mathrm{E})_2\ \mathrm{E})_1\ \Sigma_{all}$$

This applies D (the dynamic applier) n times, then reverts to Σ_{all}. This strategy might be useful for reducing the arguments of an application without performing the application itself:

$$\begin{aligned} &PE\quad (\lambda x.((\lambda y.y\ y)\ (\lambda z.z))\ (\lambda q.q))\ (\Sigma_{below,n}\ 2)\\ &\quad = App(Abs(\lambda x.Abs(\lambda z.z)), Abs(\lambda q.q))\\ &\quad = \lceil(\lambda x.(\lambda z.z))\ (\lambda q.q)\rceil \end{aligned}$$

On the other hand, $\Sigma_{below,n}\ n$, for any $n > 0$, is no different from Σ_{none} when used with Ack, since the operators and operands in those examples are in normal form, and $\Sigma_{below,n}\ 0$ is always identical to Σ_{all}, so we have not included this strategy in Table 2.

The λ-calculus version of the strategy-based partial evaluator is given in Figure 4. It is very similar to Mogensen's partial evaluator (corresponding to the similarity of the definitions of $[\![\cdot]\!]$ here and in section 2).

Figure 4 also gives the λ-calculus definitions of the strategies we have defined in this section. Note that although we need to use Mogensen's trick in the definition of D to avoid non-termination, we can freely use explicit recursion in the definitions of strategies. This is because strategies are only *executed* by the partial evaluator, and never *processed* as a term to be partially evaluated, even during self-application of the partial evaluator.

4 Self-application with strategies

Our version of the Futamura projections (Figure 2) looks almost exactly like Mogensen's. The difference is that the results of partial evaluation here may not be in completely reduced form. The strategies are used to postpone some β-reductions. The most aggressive strategy — namely, Σ_{all} — will produce the same results that Mogensen obtains, but other strategies will produce expressions that are less reduced.

The crucial feature of self-application with strategies is that the strategies are, so to speak, "off budget." The strategies never appear in residual code[3] , so no matter how complicated the strategy used, the ability to self-apply the partial evaluator is not impaired. For example, a strategy of the form ($\lambda\pi$. **if** <*very complicated, but terminating, condition*> **then** Σ_{none} **else** Σ_{none}) is equivalent to Σ_{none}.

Furthermore, a strategy is used for only a single application of the partial evaluator. For example, suppose the second Futamura projection is used to create a specialized partial evaluator: $P\ \lceil P\rceil\ \lceil\lceil M\rceil\rceil\ \Sigma = \lceil M_{gen}\rceil$. When M_{gen} is applied, it will be supplied with another, perhaps different, strategy Σ'. Thus, M_{gen} might be produced using a non-aggressive strategy like Σ_{none} (so it will be small), but then applied using an aggressive strategy (Σ_{all}). Of course, the latter application will be more costly than if M_{gen} had been produced using Σ_{all}, but that is the entire point: trading space for time.

Table 2 presents a sampling of results of various projections with various strategies. The applications that use Σ_{all} are directly comparable to Mogensen's results. The size of the results of the first projection, when Σ_{all} is the strategy, are of course identical to Mogensen's. The sizes for the other projections are larger, because our partial evaluator is somewhat larger. (Indeed, we have been unable to generate P_{gen} with the Σ_{all} strategy, because it is too large. By using less aggressive strategies we are able to get some approximations (i.e., $P_{gen_{10}}, P_{gen_{30}}$, and $P_{gen_{70}}$) but the resulting program generators have too many residualized expressions, and so partial evaluation actually yields a slowdown.) The bottom section of the table shows the application of our partial evaluator to Mogensen's

[3] This is not completely accurate, since a strategy that performs a residualization will need to supply the actual application node itself, but none of the strategy's "decision making" code will be left in the residual code.

Expression	*Size*	β	ρ
$Ack\ 2\ 3 \rightarrow 9$	21	76	
$P\ [Ack]\ [2]\ \Sigma_{all} \rightarrow Ack_{2,all}$	52	744	
$P\ [Ack]\ [2]\ \Sigma_0 \rightarrow Ack_{2,0}$	35	492	
$P\ [Ack]\ [2]\ \Sigma_1 \rightarrow Ack_{2,1}$	32	466	
$P\ [Ack]\ [2]\ \Sigma_4 \rightarrow Ack_{2,4}$	35	523	
$P\ [Ack]\ [2]\ \Sigma_8 \rightarrow Ack_{2,8}$	49	791	
$Ack_{2,0}\ 3 \rightarrow 9$	21	76	1.0
$Ack_{2,1}\ 3 \rightarrow 9$	21	75	1.013
$Ack_{2,4}\ 3 \rightarrow 9$	21	73	1.041
$Ack_{2,8}\ 3 \rightarrow 9$	21	69	1.101
$Ack_{2,all}\ 3 \rightarrow 9$	21	65	1.169
$P\ [P]\ [[Ack]]\ \Sigma_0 \rightarrow Ack_{gen,0}$	506	3120	
$P\ [P]\ [[Ack]]\ \Sigma_8 \rightarrow Ack_{gen,8}$	339	4746	
$P\ [P]\ [[Ack]]\ \Sigma_{64} \rightarrow Ack_{gen,64}$	5532	76427	
$P\ [P]\ [[Ack]]\ \Sigma_{96} \rightarrow Ack_{gen,96}$	3848	51104	
$P\ [P]\ [[Ack]]\ \Sigma_{all} \rightarrow Ack_{gen,all}$	3842	51072	
$Ack_{gen,0}\ [2] \rightarrow Ack_2$	52	758	0.981
$Ack_{gen,8}\ [2] \rightarrow Ack_2$	52	751	0.991
$Ack_{gen,64}\ [2] \rightarrow Ack_2$	52	621	1.198
$Ack_{gen,96}\ [2] \rightarrow Ack_2$	52	510	1.459
$Ack_{gen,all}\ [2] \rightarrow Ack_2$	52	510	1.459
$P\ [P]\ [[P]]\ \Sigma_{10} \rightarrow Pgen_{10}$	644	9184	
$P\ [P]\ [[P]]\ \Sigma_{50} \rightarrow Pgen_{50}$	47672	643247	
$P\ [P]\ [[P]]\ \Sigma_{70} \rightarrow Pgen_{70}$	170154	2297039	
$P\ [P]\ [[P]]\ \Sigma_{all} \rightarrow Pgen_{all}$	?	?	
$Pgen_{10}\ [[Ack]]\ \Sigma_{all} \rightarrow Ack_{gen,all}$	3842	52022	0.982
$Pgen_{50}\ [[Ack]]\ \Sigma_{all} \rightarrow Ack_{gen,all}$	3842	51992	0.982
$Pgen_{70}\ [[Ack]]\ \Sigma_{all} \rightarrow Ack_{gen,all}$	3842	51520	0.991
$P\ [mP]\ [[mP]]\ \Sigma_{all} \rightarrow mP_{gen,all}$	18083	408860	
$P\ [mP]\ [[mP]]\ \Sigma_{10} \rightarrow mP_{gen,10}$	528	7487	
$P\ [mP]\ [[mP]]\ \Sigma_{50} \rightarrow mP_{gen,50}$	8188	110161	
$P\ [mP]\ [[mP]]\ \Sigma_{100} \rightarrow mP_{gen,100}$	23893	321046	
$P\ [mP]\ [[mP]]\ \Sigma_{500} \rightarrow mP_{gen,500}$	19976	431987	
$mP_{gen,all}\ [[Ack]] \rightarrow Ack_{gen}$	693	2422	3.342
$mP_{gen,10}\ [[Ack]] \rightarrow Ack_{gen}$	693	8124	0.996
$mP_{gen,50}\ [[Ack]] \rightarrow Ack_{gen}$	693	8090	1.001
$mP_{gen,100}\ [[Ack]] \rightarrow Ack_{gen}$	693	7746	1.045
$mP_{gen,500}\ [[Ack]] \rightarrow Ack_{gen}$	693	2503	3.234

Table 2: Results of the strategy-based partial evaluator

(denoted mP in the chart); we have included these numbers because they can be compared even more directly to those in Table 1.

The results are rather difficult to read, because there are several degrees of freedom. What is of greatest interest is the trade-off between the size of residual code and the cost of applying it. For the third projection, note that while more aggressive expansions yield versions of P_{gen} that take fewer β-reductions to produce Ack_{gen}, the code size grows dramatically.

5 More strategies

The strategies of the previous section are oblivious to the properties of the expressions being evaluated. Obvious strategies like "expand if the argument is small," "expand calls to function f, but no others," and "expand if there is only one occurrence of the bound variable in the body of the λ-expression" cannot be written. In this section, we make a simple change in the abstract syntax of λ-terms which will allow the first of these strategies to be written. To write the second requires a further change of representation which we postpone to the next section. To write the third strategy — the one suggested by Mogensen — seems to be impossible when terms are represented in higher-order abstract syntax; switching to first-order abstract syntax is a possibility that we are currently exploring (see the conclusions).

The representation of λ-terms given in section 2 is not the most general representation of higher-order abstract syntax trees of λ-terms. In that representation, variables are treated specially, making some calculations impossible. We have used that representation to make our results directly comparable to those of Mogensen. However, to allow for expressing more strategies, we will now change the representation as follows: e is again represented by the term $\lceil e \rceil$, where

$$\lceil e \rceil \;=\; \lambda a \lambda b \lambda c.\overline{e}, \text{ where } \begin{array}{rcl} \overline{x} & = & c\ x \\ \overline{\lambda x.e} & = & b\ (\lambda x.\overline{e}) \\ \overline{mn} & = & a\ \overline{m}\ \overline{n} \end{array}$$

The difference is the application of AST operator c to variables.

Now we can, for example, determine the size of a lambda expression:

$$\begin{array}{rcl} \text{size } e & = & \lceil e \rceil\ (\lambda mn.\text{inc}\ (\text{plus}\ m\ n)) \\ & & (\lambda g.\text{inc}\ (g\ 1)) \\ & & (\lambda x.1) \end{array}$$

Before using strategies based on this new capability, we need to change the partial evaluator slightly to accommodate the new representation. The new code is shown in Figure 5.

The results obtained previously, shown in Table 2, will change slightly, because representations of terms are larger. But now we can write more interesting strategies, such as Σ_{small}:

$$\begin{array}{l} \Sigma_{small} \;=\; \lambda n.\lambda \pi.\lambda \pi'. \\ \quad \textbf{let}\ r \;=\; \pi\ (\Sigma_{small}\ n) \\ \quad\quad \textbf{and}\ r' \;=\; \pi'\ (\Sigma_{small}\ n) \\ \quad\quad \textbf{in if size}(r'_2) \leq n \\ \quad\quad\quad \textbf{then}\ r_1\ \pi' \\ \quad\quad\quad \textbf{else}\ (D\ r_2\ (\Sigma_{small}\ n))_1\ \pi' \end{array}$$

This strategy takes as its arguments a number n and two terms π and π'. After applying itself to its argument π', it checks to see if the size of the resulting

```
R := \m. m A B C;
P := \m n s. R (\a b c. a (m a b c)
                          (n a b c)) s F;
C := \x.x;
B := \g s sel.
      sel (\x. g x s)
          (\a b c. b (\z.
             (snd (g (D (\a b c. c z)) s)) a b c));
A := \m n s. s m n;

D := (DQ DQ);
DQ := \q. \v. \s.
     (\sel. sel
        (\qq. \w. (qq qq)
                  (\a b c. a (v a b c)
                             (snd (w s) a b c)) s)
        (\qq. v)
       q );
```

Figure 5: Strategy-based partial evaluator, with modified term representation

expression r' is smaller than or equal to n. If so, it then reduces π by applying it to Σ_{all}, otherwise it residualizes π.

Here is the running example modified slightly to illustrate the operation of Σ_{small}. First we use $(\Sigma_{small}\ 1)$:

$$\begin{aligned} PE\quad & (\lambda x.((\lambda y.y\ y)\ (\lambda z.z))\ (\lambda q.q\ q))\ (\Sigma_{small}\ 1) \\ & = App(Abs(\lambda x.App(Abs(\lambda y.App(y,y)), \\ & \quad Abs(\lambda z.Var\ z))), Abs(\lambda q.App(Var\ q, Var\ q))) \\ & = \lceil(\lambda x.((\lambda y.y\ y)\ (\lambda z.z)))\ (\lambda q.q\ q)\rceil \end{aligned}$$

Since the smallest argument in the example is $(\lambda z.z)$, which has a size of 2, no applications are performed.

Next we use $(\Sigma_{small}\ 2)$:

$$\begin{aligned} PE\quad & (\lambda x.((\lambda y.y\ y)\ (\lambda z.z))\ (\lambda q.q\ q))\ (\Sigma_{small}\ 2) \\ & = App(Abs(\lambda x.Abs(\lambda z.Var\ z)), \\ & \quad Abs(\lambda q.App(Var\ q, Var\ q))) \\ & = \lceil(\lambda x.(\lambda z.z))\ (\lambda q.q\ q)\rceil \end{aligned}$$

The outer argument $(\lambda q.q\ q)$ has a size of 6, so it is residualized. However, the argument $(\lambda z.z)$ inside the body of the outermost function is small enough, so the $(\lambda y.y\ y)$ is applied to it. The result is $(\lambda z.z)\ (\lambda z.z)$, which again meets $(\Sigma_{small}\ 2)$'s criteria for reduction, so it is reduced to $(\lambda z.z)$.

If we use $(\Sigma_{small}\ 6)$, the entire expression is reduced.

Keep in mind that these strategies, though very costly to apply (especially since all arithmetic is done using Church numerals), do not appear in residual code and therefore do not impede self-application of the partial evaluator.

Often strategies produce unexpected results. For example, consider the term

$$((\lambda a.(\lambda b.(\lambda c.c)\ (\lambda r.r\ r))\ (\lambda x.x))\ (\lambda q.q\ q))$$

There are three applications at the top level. If we apply the $(\Sigma_{small}\ 3)$ strategy none of these applications will be performed, even though the argument to the λb term has a size of 2. This is because the argument to the λa term has a size of 6, and must be performed first in order for the λb redex to be reachable. What we need is a strategy that behaves like Σ_{all} for one level, and then behaves like Σ_{small} afterwards.

The solution is to compose strategies. For this example, we want a strategy that will perform the first few β-reductions at the top level, and then gives control to Σ_{small}. This new strategy is called $\Sigma_{n,then}$, and is a generalization of Σ_n.

$$\Sigma_{n,then}\ =\ \lambda n.\lambda\Sigma.n\ (\lambda \mathrm{E}.\lambda\pi.(\pi\ \mathrm{E})_1)\ \Sigma$$

Thus, $(\Sigma_{n,then}\ n\ \Sigma)$ performs β-reductions at the top n levels of the term, then reverts to Σ.

Using this strategy with our example gives us:

$$\begin{aligned} PE\quad & ((\lambda a.(\lambda b.(\lambda c.c)\ (\lambda r.r\ r))\ (\lambda x.x))\ (\lambda q.q\ q)) \\ & \quad (\Sigma_{n,then}\ 1\ (\Sigma_{small}\ 3)) \\ & = \lceil(\lambda c.c)\ (\lambda r.r\ r)\rceil \end{aligned}$$

The first two applications were reduced, while the third was residualized.

These strategies allow finer control of partial evaluation in self-application as well. For example, we can create an Ack_{gen} using the second projection. If we use the Σ_{none} strategy, Ack_{gen} is of size 269, and needs 845 β-reductions to execute when applied to 2 and Σ_{all}. If we use $(\Sigma_{n,then}\ 5\ (\Sigma_{small}\ 20))$ the size is 263, and needs 843 β-reductions. Finally, using Σ_{all} results in a much larger term of size 4446, but it only needs 548 β-reductions. Lack of space prevents us from presenting more experimental results.

6 Annotations

Some desired strategies — such as expanding calls to specific, named functions — cannot be expressed, because they are based on extrinsic considerations. We can accommodate these strategies by changing the representation yet again, to include an annotation field in each λ-term. Though this clearly crosses the line from on-line to off-line partial evaluation — since the annotations on each term will be made by some preprocessing step — we feel it is still interesting to see how strategies can use these annotations. Furthermore, it demonstrates that the use of strategies is in some sense more general than the use of binding-time analysis.

For our final change of representations, we add an annotation field to abstractions. For this example the annotation will be a boolean value which expresses whether or not we want to perform a β-reduction if given the opportunity. Annotations of this type are discussed in chapter 7 of [3].

A λ-expression e is represented by

$$\lambda abc.\overline{e}, \text{ where } \begin{array}{lcl} \overline{x} & = & (c\ x) \\ \overline{\lambda x.e} & = & (b\ X\ \lambda x.\overline{e}) \\ \overline{mn} & = & (a\ \overline{m}\ \overline{n}) \\ \text{X} & is & T \text{ or } F. \end{array}$$

The translation of terms to PEV's must take into account the transmission of annotations from one term to another:

$$\begin{array}{lcl} [\![x]\!] & \mapsto & x \\ [\![m\ n]\!] & \mapsto & \lambda\Sigma.\ \Sigma\ [\![m]\!]\ [\![n]\!] \\ [\![\lambda x.e]\!] & \mapsto & \text{let } g\ =\ (\lambda x.[\![e]\!]\ \Sigma) \text{ in} \\ & & \quad < T,\ < g, Abs(\lambda w.(g\ D(Var(w)) \\ & & \quad\quad \Sigma)_1) >> \end{array}$$

Our last version of the partial evaluator is shown in Figure 6.

Because the representation of expressions has changed, strategies such as Σ_{none} will need to be modified. Projecting the first element from the $Result$ pair returns another pair consisting of the annotation and the expression. Here is the modified Σ_{none}:

$$\Sigma_{none}\ =\ \lambda\pi.((D\ ((\pi\ \Sigma_{none})_2)_2\ \Sigma_{none})_2)_1$$

A more interesting strategy is Σ_{marked}:

$$\begin{array}{l} \Sigma_{marked}\ =\ \lambda\pi. \\ \quad \textbf{let } r\ =\ \pi\ \Sigma_{marked} \\ \quad \textbf{in if } r_1 \textbf{ then } (r_2)_1 \\ \quad\quad\quad \textbf{else } ((D\ ((t\ \Sigma_{none})_2)_2\ \Sigma_{none})_2)_1 \end{array}$$

Similar to Σ_{small}, it first applies its argument π to itself to get r. But instead of checking the size of r, it checks the annotation field. If that is $true$, then it returns the static part of r; otherwise, it uses Σ_{none} and D to residualize.

As an example, consider the expression from the previous section. With Σ_{marked} we could annotate the first two abstractions to be reduced, and the final one to be residualized. Here is the example again, with the abstractions marked for residualization underlined.

```
R := \m. m A B C;
P := \m n s. R (\a b c. a (m a b c)
                              (n a b c)) s F F;
C := \x.x;

B := \n g s. (\ sel. sel n
      (\selExp. selExp
          (\x. g x s)
          (\a b c. b T (\z.
             (fst (snd (g (D (\a b c. c z))
                          s))) a b c)))


A := \m n s. s m n;

D := (DQ DQ);
DQ := \q. \v. \s. (\sel. sel T
      (\selExp. selExp
         (\qq. \w. (qq qq)
            (\a b c. a (v a b c)
                       (snd (snd (w s)) a b c)) s)
         (\qq. v)
        q));
```

Figure 6: Strategy-based partial evaluator, using annotations

$$\begin{array}{ll} PE & ((\underline{\lambda a}.(\underline{\lambda b}.(\lambda c.c)\ (\lambda r.r\ r))\ (\lambda x.x))\ (\lambda q.q\ q))\ \Sigma_{marked} \\ & = \lceil(\lambda c.c)\ (\lambda r.r\ r)\rceil \end{array}$$

If a binding time analyzer were employed to annotate terms, our partial evaluator with the Σ_{marked} strategy would mimic an off-line partial evaluator.

7 Conclusions

We have presented a technique for controlling on-line partial evaluation while preserving self-applicability. Strategies are used by the partial evaluator as "advisors" on when to β-expand, but do not appear in residual code. Thus, no matter how complicated the decision-making process they employ, self-applicability is not compromised. Strategies can consider intrinsic properties of λ-terms and, crossing over into the realm of off-line partial evaluation, can consider annotations on terms that express the results of pre-processing.

A number of research problems present themselves from this work. The results obtained from many of the strategies we have defined are far from intuitive. A more systematic experimental study of space-time trade-offs in partial evaluation could lead to a better understanding of what approaches to partial evaluation are best. It would also be interesting to incorporate binding-time analysis into strategies; section 6 discussed how this can be done, but we have not yet experimented with an actual binding-time analyzer. Note that strategies allow for partial evaluation of terms that have no normal form, that is, terms with explicit recursion. This fact could allow us to substantially simplify and generalize P.

The systematic construction of strategies by way of a "strategy algebra" seems well within reach, and would be useful. $\Sigma_{n,then}$ is an example of a simple composition of strategies, and it would appear that many such combinations are possible.

The most exciting possibility for this approach is that strategies might be applicable in partial evaluators based on *first-order* abstract syntax. This would go a long way toward bridging the gap between on-line and self-applicable partial evaluators. Strategies could make static-vs.-dynamic decisions with great precision, as is now done only in on-line partial evaluators, yet self-applicability would be preserved.

Acknowledgments

We would like to thank our colleagues Uday Reddy and Bill Harrison for helpful discussions over the course of this research.

References

[1] B. Grant, M. Mock, M. Philipose, C. Chambers, S. Eggers, Annotation-Directed Run-Time Specialization in C, Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), June 1997. Amsterdam, Netherlands.

[2] Neil D. Jones, Peter Sestoft, and Ha rald Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989. DIKU Report 91/12.

[3] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International Series in Computer Science. Englewood Cliffs, NJ: Prentice Hall, 1993.

[4] Torben Æ. Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In *Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, June 1995*, pages 39–44. New York: ACM, 1995.

[5] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[6] Eric Ruf. *Topics in Online Partial Evaluation.* PhD thesis, Stanford University, California, February 1993. Published as technical report CSL-TR-93-563.

[7] E. Volanschi, C. Consel, G. Muller, C. Cowan, Declarative specialization of objet-oriented programs. OOPSLA '97, Atlanta, Oct. 1997, pp. 286–300.