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 $\lambda$-calculus. His method is to regard every $\lambda$-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 trade-off 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 $\lambda$-calculus. The idea is to interpret every $\lambda$-expression as a pair of values: an equivalent $\lambda$-expression and a function, essentially representing the application of that $\lambda$-expression by $\beta$-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 $\beta$-reduction.

The only problem with Mogensen's solution is that the results of self-applications of his partial evaluator are extremely large $\lambda$-expressions, because $\lambda$-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 93-00413
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:

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

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

---

(1) \( P \ [M] \ [N] = [M \times N] \),  
where \( M \times N = \beta M \times N \)

(2) \( P \ [P] \ [M] = [P \times M] \),  
where \( P \times M = \beta P \times M \)

(3) \( P \ [P] \ [P] = [P \times P] \),  
where \( P \times P = \beta P \times P \)

---

Figure 1: Original Futamura Projections

---

(1) \( P \ [M] \ [N] = [M \times N] \),  
where \( M \times N = \beta M \times N \)

(2) \( P \ [P] \ [M] = [P \times M] \),  
where \( P \times M = \beta P \times M \)

(3) \( P \ [P] \ [P] = [P \times P] \),  
where \( P \times P = \beta P \times P \)

Figure 2: Futamura Projections with Strategies
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.

\[
\begin{align*}
&[\ ] : \text{A} \rightarrow \text{PEV} \\
&[x] \rightarrow x \\
&[m \cdot n] \rightarrow [m] \cdot [n] \\
&[x : e] \rightarrow \lambda g = \lambda x. [e] \\
&\text{let } g = \lambda x. [e] \text{ in } < g, \text{Abs} (\lambda w. (g (D(Var(w)))) | z) |
\end{align*}
\]

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

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 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 \( \langle D(\text{Var}(w)) \rangle \) and abstracting that variable (\( \text{Abs}(\lambda w. \ldots) \)). To turn the variable \( \text{Var}(w) \) 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 \( F: \text{PEV} \rightarrow \text{PEV} \) that “textually” applies \( e \). That is, given any \( \text{PEV} < \ldots, e', > \), \( F \) returns the PEV \( < F', \text{App}(e, e') > \), where \( F' \), when applied to \( \text{PEV} < \ldots, e'' > \), returns \( < \ldots, \text{App}(\text{App}(e, e'), e'') > \), 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.

\[
\text{normalize } e = [e]_2
\]

\( \text{PE } m \cdot n = \text{normalize}(m \cdot n) \)

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

\[
\begin{align*}
&\text{PE } (\lambda x. (((\lambda y. y) (\lambda z. z))) (\lambda q. q)) \\
&\rightarrow \text{normalize } (((\lambda x. ((\lambda y. y) (\lambda z. z))) (\lambda q. q)) \\
&\rightarrow \langle \langle (((\lambda y. y) (\lambda z. z))) (\lambda q. q) \rangle, \rangle \\
&\rightarrow \langle \langle (((\lambda y. y) (\lambda z. z))) (\lambda q. q) \rangle, \rangle \\
&\rightarrow \langle \langle (((\lambda y. y) (\lambda z. z))) (\lambda q. q) \rangle, \rangle \\
&\rightarrow \langle \langle (((\lambda y. y) (\lambda z. z))) (\lambda q. q) \rangle, \rangle \\
&\rightarrow \langle \langle (((\lambda y. y) (\lambda z. z))) (\lambda q. q) \rangle, \rangle \\
&\rightarrow \langle \langle (((\lambda y. y) (\lambda z. z))) (\lambda q. q) \rangle, \rangle \\
&\rightarrow \langle \langle (((\lambda y. y) (\lambda z. z))) (\lambda q. q) \rangle, \rangle \\
&\rightarrow \langle \langle (((\lambda y. y) (\lambda z. z))) (\lambda q. q) \rangle, \rangle \\
&\rightarrow \langle \langle (((\lambda y. y) (\lambda z. z))) (\lambda q. q) \rangle, \rangle \\
&\rightarrow \langle \langle (((\lambda y. y) (\lambda z. z))) (\lambda q. q) \rangle, \rangle \\
&\rightarrow \langle \langle (((\lambda y. y) (\lambda z. z))) (\lambda q. q) \rangle, \rangle \\
&\rightarrow \langle \langle (((\lambda y. y) (\lambda z. z))) (\lambda q. q) \rangle, \rangle \\
\end{align*}
\]

Mogensen shows how to render these ideas into \( \lambda \)-calculus. First, we need to represent the abstract syntax of \( \lambda \)-terms within \( \lambda \)-calculus. A \( \lambda \)-term \( e \) is translated to an \( \text{Expr} \) by the function \([\cdot]\), defined as:

\[
[e] = \lambda x b. e, \text{ where } \frac{x}{\overline{\overline{x}}} = a \overline{\overline{a}} \frac{x}{\overline{\overline{x}}}
\]

Then, an \( \text{Expr} \) is translated to a \( \text{PEV} \) by applying it to terms \( A \) and \( B \):

\[
\begin{align*}
A &\equiv \lambda p. \lambda y'. p_1 y' \\
B &\equiv \lambda q. \langle g, (\lambda a b (\lambda w. (g (D(\lambda a b w)))) _2 a b) \rangle \\
D &\equiv Y \lambda d. \lambda m. < m, (\lambda y d. (\lambda a b (m a b) (v_2 a b)) >
\end{align*}
\]

In other words, \([\cdot]\) operates on terms of type \( A \), the \( \lambda \)-term corresponding to it in our implementation, which we call \( P \), operates on \( \text{Expr's} \).

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

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

\footnote{We will often call the first and second components of a PEV the function and expression components, respectively.}
\begin{align*}
F &:= \textit{m \rightarrow n, R} (\textit{\lambda a. b. a (m a b) (m a b)}); \\
R &:= \textit{1m, mABF}; \\
A &:= \textit{1m, mTN}; \\
B &:= \textit{g, sel, sel} (\textit{\lambda a. b. b (\langle w, g \ (D \ \lambda a. b. F \ a b) \rangle)}); \\
D &:= \textit{g Q}; \\
Q &:= \textit{\langle sel, sel \ (\textit{\langle q, \ m \ (\textit{\lambda a. b. a (m a b)} \ (v \ F \ a b)) \rangle \ (\textit{\langle q, m \rangle})}) \ q \rangle}; \\
F &:= \textit{\langle fst snd, snd \rangle}; \\
T &:= \textit{\langle fst snd, fst \rangle};
\end{align*}

Figure 3: Mogensen’s partial evaluator

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, \( P[M][N] \) returns the representation \( \langle M, N \rangle \) of an expression \( M[N] \). For any \( X, M, N, X =_{\beta} M \ N \ X \), but \( M(x) \) normalizes more quickly than \( M \ N \ X \).

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

\[
\text{Ack} = \ \lambda m. n. m \ ((\lambda f. \ m \ \ (m \ f \ (\lambda x. x))) \ (\lambda m. f. \ (m \ f \ (m \ f \ x))) \ n)
\]

Some of his results are summarized in Table 1. The \( \rho \) column indicates the speedup, measured in \( \beta \)-reductions, obtained by applying \( P[M][N] \) to an argument \( X \), as compared to calculating \( M \ N \ X \) directly. For example, it takes 2.7 times as many \( \beta \)-reductions to produce \( \text{Ack1} \) by calculating \( P[\text{Ack2}] \ [1] \) as it does to produce it by applying \( \text{Ack2} \) to \([1]\).

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 \( \beta \)-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.

\[
\text{Table 1: Excerpt from Table 2 of [4]}
\]

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:

\[
\text{PEV = Strategy \rightarrow Result} \\
\text{Result = (PEV \rightarrow Result) \times Expr} \\
\text{Strategy = PEV \rightarrow PEV \rightarrow Result}
\]

Again, a \( \lambda \)-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 \( \lambda \)-terms needs to be modified. As above, \( [\epsilon] \) is the \( PEV \) associated with \( \epsilon \):

\[
\begin{align*}
[x] &\rightarrow x \\
[m n] &\rightarrow \lambda \Sigma. \Sigma \ [m] \ [n] \\
[\lambda x. \epsilon] &\rightarrow \text{let } g = \lambda x. ([\epsilon] \Sigma) \text{ in } < g, \text{Abs}(\lambda w. [g \ (D[\text{Var}(w)]) \Sigma]) \Sigma > \\
\end{align*}
\]

where \( D \epsilon \Sigma = < \text{Abs}(\lambda w. D(\text{App}(\epsilon, (\nu \Sigma) \Sigma)) \Sigma), \epsilon > \).

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

\[
\text{normalize } \epsilon \Sigma = ([\epsilon] \Sigma) \Sigma \\
\text{PEV m n } \Sigma = \text{normalize } (m \ n) \Sigma
\]

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

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

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

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

$$PE \ (\lambda x. (\lambda y. y) (\lambda z. z)) (\lambda q. q) \Sigma_{all}$$

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

$$PE \ (\lambda x. (\lambda y. y) (\lambda z. z)) (\lambda q. q) \Sigma_{all}$$

To again compare our results to Mogensen’s directly, if we denote $P[\text{Ack}]$ our $\Sigma_{all}$ by Ack$_{2,\text{none}}$, we see in Table 2 that Ack$_{2,\text{none}}$ 3 reduces to 9 after 76 $\beta$-reductions, compared to the 65 for Ack$_{2,\text{alt}}$. Of course, it takes fewer $\beta$-reductions to calculate Ack$_{2,\text{none}}$ and, perhaps more importantly, Ack$_{2,\text{none}}$ 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 $\Sigma_n$ performs the first $n$ $\beta$-reductions (for some $n$) and then no more.

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

Here, $n$ is a church numeral which will cause the $\lambda E$ term to be applied $n$ times, followed by $\Sigma_{none}$. For example:

$$PE \ (\lambda x. (\lambda y. y) (\lambda z. z)) (\lambda q. q) \Sigma_{none}$$

In other words, the evaluator did one $\beta$-reduction the outer one and then stopped.

Table 2 shows results using $\Sigma_n$ with Ack, for $n = 1, 4, 8$. We can see that as we go from (1) up to (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$_{2,\text{none}}$ and Ack$_{2,\text{alt}}$.

Another “counting” strategy is the one that abstains from doing top-level reductions, but does reductions within subexpressions.
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, \(\Sigma_{all}\) 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 \((\lambda x. y)\) if <very complicated, but terminating, condition> then \(\Sigma_{none}\) else \(\Sigma_{none}\) is equivalent to \(\Sigma_{none}\).

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: \(P [P] [[M]] \Sigma = [M_{gen}]\). When \(M_{gen}\) is applied, it will be supplied with another, perhaps different, strategy \(\Sigma'\). Thus, \(M_{gen}\) might be produced using an non-aggressive strategy like \(\Sigma_{none}\) (so it will be small), but then applied using an aggressive strategy \(\Sigma_{all}\). Of course, the latter application will be more costly than if \(M_{gen}\) had been produced using \(\Sigma_{all}\), 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 \(\Sigma_{all}\) are directly comparable to Mogensen’s results. The size of the results of the first projection, when \(\Sigma_{all}\) 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 \(\Sigma_{all}\) strategy, because it is too large. By using less aggressive strategies we are able to get some approximations (i.e., \(P_{gen10}, P_{gen30}\), and \(P_{gen70}\)) 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.
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 $\lambda$-expression” cannot be written. In this section, we make a simple change in the abstract syntax of $\lambda$-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 $\lambda$-terms given in section 2 is not the most general representation of higher-order abstract syntax trees of $\lambda$-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: $\epsilon$ is again represented by the term $[\epsilon]$, where

$$[\epsilon] = \lambda a \lambda b. c. \overline{\epsilon}, \text{ where } \overline{\epsilon} = e x \overline{\lambda n. \epsilon} = b (\lambda x. \overline{\epsilon})$$

The difference is the application of AST operator $e$ to variables.

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

$$\text{size } \epsilon = [\epsilon] \cdot (\lambda m n. \text{inc} \text{ (plus } m n)) \cdot (\lambda x. 1)$$

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 $\Sigma_{\text{small}}$:

$$\Sigma_{\text{small}} = \lambda n. \lambda \pi. \lambda \pi' . \text{let } r = \pi (\Sigma_{\text{small}} n) \text{ and } r' = \pi' (\Sigma_{\text{small}} n) \text{ in if size } (r') \leq n \text{ then } r \pi' \text{ else } (D r_2 (\Sigma_{\text{small}} n)) \pi'$$

This strategy takes as its arguments a number $n$ and two terms $\pi$ and $\pi'$. After applying itself to its argument $\pi'$, it checks to see if the size of the resulting
expression $r'$ is smaller than or equal to $n$. If so, it then reduces $\pi$ by applying it to $\Sigma_{\text{alt}}$, otherwise it residualizes $\pi$.

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

$$PE \quad (\lambda x.((\lambda y.y y) (\lambda z.z)) (\lambda q.q)) \quad (\Sigma_{\text{small}} 1)$$

$$= \text{App}(\text{Abs}(\lambda x.\text{App}(\lambda s.\text{Abs}(\lambda y.\text{App}(y,y)), \text{Abs}(\lambda z.\text{Var}(z))), \text{Abs}(\lambda q.\text{App}(\text{Var}(q),\text{Var}(q)))) \quad = \quad [(\lambda x.((\lambda y.y y) (\lambda z.z)) (\lambda q.q)]$$

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

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

$$PE \quad (\lambda x.((\lambda y.y y) (\lambda z.z)) (\lambda q.q)) \quad (\Sigma_{\text{small}} 2)$$

$$= \text{App}(\text{Abs}(\lambda x.\text{Abs}(\lambda z.\text{Var}(z))), \text{Abs}(\lambda q.\text{App}(\text{Var}(q),\text{Var}(q)))) \quad = \quad [(\lambda x.((\lambda y.y y) (\lambda z.z)) (\lambda q.q)]$$

The outer argument $(\lambda q.q)$ 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 ($\Sigma_{\text{small}}$ 2)’s criteria for reduction, so it is reduced to $(\lambda z.z)$.

If we use ($\Sigma_{\text{small}}$ 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

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

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

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

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

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

Using this strategy with our example gives us:

$$PE \quad (\lambda a. (\lambda b. (\lambda c. (\lambda x. x))) (\lambda q.q)) \quad (\Lambda_{\Sigma,\text{then}} 1 \quad (\Sigma_{\text{small}} 3))$$

$$= \quad [(\lambda c.c) (\lambda r.r)]$$

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 Ackgen, using the second projection. If we use the $\Sigma_{n,\text{one}}$ strategy, Ackgen is of size 269, and needs 845 $\beta$-reductions to execute when applied to 2 and $\Sigma_{alt}$. If we use $\Sigma_{n,\text{then}} 5 \quad (\Sigma_{\text{small}} 20)$ the size is 263, and needs 843 $\beta$-reductions. Finally, using $\Sigma_{alt}$ results in a much larger term of size 4446, but it only needs 548 $\beta$-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 $\lambda$-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

$$\lambda e. \overline{e}, \text{ where } \overline{e} = (c \ x)$$

$$\overline{\lambda e. \overline{e}} = (b \ X \ \lambda x. \overline{e}) \tag{m}$$

$$\overline{\lambda x. \overline{e}} = (a \overline{\overline{m}} \overline{\overline{m}}) \tag{x is T or F.}$$

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

$$[x] \rightarrow x$$

$$[m \ n] \rightarrow \lambda \Sigma. \ [m] \ [n]$$

$$[\lambda x. e] \rightarrow \text{let } g = (\lambda x. [e]) \Sigma \text{ in}$$

$$\begin{align*}
\text{ if } & \text{ T, then } <g, \text{Abs}(\lambda w. (g \ D(Var(w)))\tag{\Sigma(1)}) \text{ else } <\text{let } g = (\lambda x. [e]) \Sigma \text{ in} \tag{\Sigma(2)}
\end{align*}$$

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

Because the representation of expressions has changed, strategies such as $\Sigma_{none}$ 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 $\Sigma_{none}$:

$$\Sigma_{none} = \lambda \pi. ((D ((\pi \ \Sigma_{none})2) \ 2 \ \Sigma_{none})2)$$

A more interesting strategy is $\Sigma_{marked}$:

$$\Sigma_{marked} = \lambda \pi.$$

$$\text{let } r = \pi \ \Sigma_{marked} \text{ in if } r1 \text{ then } (r2)1 \text{ else } ((D ((\pi \ \Sigma_{marked})2) \ 2 \ \Sigma_{none})2)1$$

Similar to $\Sigma_{smooth}$, it first applies its argument $\pi$ 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 $\Sigma_{none}$ and $D$ to residualize.

As an example, consider the expression from the previous section. With $\Sigma_{marked}$ 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.

$$P := \text{\underline{\lambda m n s. R (\lambda a b c. a (m a b c)}} \ (n a b c)) \ \text{ s F F;}}$$

$$A := \text{\underline{\lambda x. x;}}$$

$$D := (\text{DS DS;})$$

$$\text{Figure 6: Strategy-based partial evaluator, using annotations}$$

$$\begin{align*}
\text{P} & \Rightarrow (\text{Abs}(\lambda w. (g \ D(Var(w)))\tag{\Sigma(1)})\text{ else } <\text{let } g = (\lambda x. [e]) \Sigma \text{ in} \tag{\Sigma(2)})
\end{align*}$$

If a binding time analyzer were employed to annotate terms, our partial evaluator with the $\Sigma_{marked}$ 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


