CONTROLLING PARITAL EVALUATORS USING FUNCTIONAL PARAMETERS

BY

A. MATTOX BECKMAN, JR.

B.A., University of Illinois at Urbana-Champaign, 1993

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2003

Urbana, Illinois

Abstract

Partial evaluation is a source to source program transformation that makes use of symbolic interpretation to perform aggresive constant propagation. Most partial evaluators are written as monolithic programs that operate on their terms directly by recursing over the structure of the expressions. Several partial evaluators and code generators have been published that do not process their terms directly; instead, the partial evaluator is inlined into the program to be specialized via a source to source translation. This technique has resulted in some evaluators with interesting properties, such as a very small size and the ability to self-apply in the online case.

A weakness of these partial evaluators is that they lack a mechanism to control the specialization process, and as a result explode on inputs that more standard offline partial evaluators are able to handle. We propose the addition of a control mechanism called a *strategy*, a function that advises the partial evaluator about whether to perform a reduction. Like traditional methods such as binding time analysis, strategies can prevent combinatorial explosion by instructing the partial evaluator to residulaize at appropriate times. They also allow us to make tradeoffs, such as partial evaluation speed versus the size of the result. Strategies are a way of abstracting control, and allow us to study the effect of different reduction techniques, and even combine them. For example, strategies can look at both source terms or binding time annotations, which gives them the potential to allow a combination of online and offline techniques.

For larger languages, this technique generalizes to the composition of many small functions, each of which handles a certain aspect or implements a specific heuristic of partial evaluation. A new behavior can be implemented by means of a strategy, and added to the partial evaluator by composing it with the rest of the strategies being used. Behaviors shown include off-line partial evaluation, bounded static variation, reading of termination annotations, and type directed partial evaluation.

Table of Contents

List of Figures

Chapter 1

Introduction

A partial evaluator is a kind of interpreter that has the ability to accept a subset of its target program's input, performing symbolic computation whenever it does not have enough information to produce a final value for an expression. Instead of emitting a value, a partial evaluator may emit another program.

A canonical example of this involves a string formatting primitive `printf`. The function `printf` can be said to take two arguments: a list of format codes, and a list of values to be formatted. The output is a string. Supposing we have predefined functions `formatInt` and `formatFloat`, we could write the function `printf` in Scheme as in figure 1.1.

```
(define printf
  (lambda (flist vlist)
     (if (null? flist) ""
       (cond
         ((equal? (car flist) "%d")
          (string-append (formatInt (car vlist))
                         (printf (cdr flist) (cdr vlist))))
         ((equal? (car flist) "%f")
          (string-append (formatFloat (car vlist))
                         (printf (cdr flist) (cdr vlist))))
         (else (string-append (car flist)
                              (printf (cdr flist) vlist)))
)  ) ) )
```

Figure 1.1: An implementation of `printf`

Given a Scheme interpreter I, we can consider I a function that takes this function `printf` along with the arguments that belong to it, and returns the string as a value.

$$I(\texttt{printf}, \texttt{'("\%d" "\%f")}, \texttt{'(3 4.2)}) \Rightarrow \texttt{"3 4.2"}$$

The interpreter will execute the `printf` function, which will parse the strings in the format list. Each time an element of the format list is parsed, `printf` reads one of the values and passes it to the appropriate formatting functions. Suppose we always run `printf` with the same format string. In this case, `printf` will re-interpret the format list each time, coming to the same conclusions about how to format the values in the value list.

In contrast, a partial evaluator P takes the function `printf` as an argument, but only *some* of `printf`s values. Suppose we always ran `printf` with the same format list `'("%d" "%f")`. We could partially evaluate `printf` with respect to that list via the following equation:

$$P(\texttt{printf}, \texttt{'("\%d" "\%f")}) \Rightarrow \text{figure 1.2}$$

The code in figure 1.2 shows the result: everything that could have been done knowing only the value of `flist` (the format list) has now been done. What was left over has been *residualized*—emitted as residual code—to be run at a later time when the remaining inputs are available. In order to do this, values have been propagated symbolically, recursive calls have been unfolded, and a new function eliding the first argument has been created. Looking at it from a higher level perspective, the layer of interpretation has been eliminated. The new function takes a list with two elements and takes it for granted that the first element is an integer and the second argument is a float.

```
(define printfpe
  (lambda (vlist)
    (string-append (formatInt (car vlist))
                   (string-append (formatFloat (car (cdr vlist)))
                                  ""))))
```

Figure 1.2: Function `printf` after partial evaluation

This confers a major engineering advantage. Without a partial evaluator, a programmer concerned with the execution speed of the program would be tempted to make a more specific version of `printf`, one that only formats one integer followed by one float. This has some major disadvantages. It requires the programmer to duplicate work. The `printf` function will now have to be "rewritten" for this special case. Having to rewrite a function means that there are now two functions which will need to be tested, debugged, and maintained. Having two functions which perform largely the same function means more work later if we decide to change the semantics of the `printf` format string; we will have to remember that there are two versions of the function, and changes to one of them will need to be mirrored in the other. This leads to unreliable, hard to maintain code, and has been documented as a contributing factor in catastrophic failures in real-world systems[28].

1.0.1 On-line and Off-line Partial Evaluation

An important distinction is made in how a partial evaluator works. Some partial evaluators function very much like interpreters, except that the partial evaluator uses symbolic interpretation when the value of a variable or expression is not known. This is known as *on-line partial evaluation.* Such partial evaluators tend to be easy to write, very accurate, and very unstable, as we will discuss in the rest of the chapter.

An alternative is to use a preprocessor to determine which expressions should be knowable and which should be treated as symbols. The preprocessor makes the decisions, and the partial evaluator performs the resulting actions. This two-stage process is called *off-line partial evaluation.* Such partial evaluators tend to be stable, but miss opportunities for specialization that an on-line partial evaluator would have taken.

1.1 The Futamura Projections

The behavior of a partial evaluator P is often described via three equations[11] known as the Futamura Projections, named for their discoverer.

1.1.1 First Projection

Let M be the source code (in some language $\mathcal{L}$) of a program which takes two inputs s and d. M can be run by making use of an $\mathcal{L}$-interpreter I, as described by the following equation:

$$I(M, (s, d)) \Rightarrow x.$$

Here, I takes two parameters: M, the program to be interpreted; and s and d, the inputs[1] to M. The interpreter performs the necessary computations, producing the result x. (Assume that I is itself written in a top-level language $\mathcal{T}$.)

The first Futamura projection has a form similar to the equation for interpreters.

$$P(M, s) \Rightarrow M_s, \text{where } I(M_s, d) \Rightarrow x$$

Here, the partial evaluator P is applied to M and one input s. The result of the partial evaluation is a new program M_s, which takes the remaining input d and produces the result x that would have been produced had both inputs been made available at the beginning. While an interpreter needs to know both of the inputs to M to work, a partial evaluator only needs to know a subset of them.

The existence of a program such as M_s follows from the S-m-n theorem[10] of logic, and can be made trivially by currying M and applying it to s. However, we usually expect that most of the computations in M that depend only on s and other constants within M itself will have been completed and hard-coded into M_s.

If we have a partial evaluator for our top-level language $\mathcal{T}$, then we can apply it to the $\mathcal{L}$-interpreter, yielding a very interesting result:

$$P(I, M) \Rightarrow I_M, \text{where } I_M(s, d) \Rightarrow x$$

The program I_M takes the values that M would have needed during its interpretation. Note that I_M is "written" in the top-level language and encodes the execution of program M. In short,

[1] s and d represent *static* and *dynamic* input, about which more will be said later.

M has been compiled!

1.1.2 Second Projection

The second projection is similar to the first projection, except that the argument to the partial evaluator is itself a partial evaluator. When P and P' are instances of the same partial evaluator, it is called *self application*.

$$P'(P, M) \Rightarrow P_M, \text{where } P_M(s) \Rightarrow M_s.$$

The second partial evaluator P partially evaluates the term M, under control of the first partial evaluator P'. Not having its second input, P cannot complete the partial evaluation of M; it has to wait until s is supplied. Again, while this result could be achieved simply by currying P, it is understood that much or all of the work that P would have to do in processing M is completed in the term P_M. The term P_M is also called M_{gen}, for "M-generator".

In practice, it is not uncommon to observe that $P_M(s)$ runs an order of magnitude faster than $P(M, s)$. (Unfortunately, it is also not uncommon for $P_M(s)$ to be an order of magnitude *larger* than $P(M, s)$ before evaluation.)

Again, we can place an interpreter I in place of M.

$$P(P, I) \Rightarrow P_I, \text{where } P_I(M) \Rightarrow I_M.$$

This time, we've created P_I, which is a compiler for the language interpreted by I. If we can think of the first projection as being able to take an interpreter and use it to perform a compilation, then the second projection takes an interpreter and a partial evaluator and uses it to create a compiler.

1.1.3 Third Projection

The third projection uses three partial evaluators.

$$P''(P', P) \Rightarrow P'_P, \text{where } P'_P(M) \Rightarrow P_M.$$

The term P_P is also called P_{gen}, or a *code generator*.

Applying P_{gen} to an interpreter I has the same effect as the second projection: the interpreter is turned into a compiler P_I. The difference is that P_P is expected to run faster than $P(P, I)$.

1.2 Combinatorial Explosion

One serious problem that can occur during self application is known as *combinatorial code explosion.* The size of the result of a second or third projection could be worse than exponential in the size of the original partial evaluator.

To see why this happens, consider the following outline of a partial evaluator P:

$$P = \lambda e.\texttt{if known}(e)\ \texttt{then compute}(e)\ \texttt{else residualize}(e)$$

A partial evaluator works by asking, for each subexpression e, whether or not it knows enough about e to compute its value. If so, it performs a computation using e, otherwise it emits the source code for e.

Now consider what happens when P is applied to some fragment of M:

$$M = \ldots x + 20\ \ldots$$

The function P will compute $\texttt{known}(x + 20)$. If x is known to the partial evaluator (i.e., it is determined by constants in M or the input s) then its value will be added to 20. Otherwise the source code "$x + 20$" will be emitted. During this process P itself may be called recursively on these subexpressions.

This process breaks down during the second projection, when P is applied to M via another partial evaluator P', as in $P'(P, M)$. When P is processing $x + 20$, it will first ask if it knows the value of x. In this case, P has not yet been told which inputs will be given to M. It is not possible for P' to say whether P will know the value of x or not. Therefore, P' will decide that the value of $\texttt{known}(x + 20)$ in P is itself unknown, neither true nor false, causing both branches of the `if` statement in P to be processed. The call to `known` also will be unfolded. As these functions contain recursive calls to P, the end result is that each subexpression[2] of M ends up having its own slightly specialized version of P "wrapped around" it.

This phenomenon makes the second projection very expensive or impossible to compute, to say nothing of the third projection.

1.3 Binding Time Analysis

One effective method for preventing combinatorial code explosion is the use of *Binding Time Analysis*, or BTA. The Binding Time Analyzer is given a program and told which of the inputs will be known during the partial evaluation phase. The result is a program in which every sub-term has been annotated as either "static" (meaning that the term is known at partial evaluation time and should be treated as code) or "dynamic" (meaning that the term known only at run-time, and should for now be treated as text).[3] This style of partial evaluation which uses a BTA preprocessing phase is called *off-line* partial evaluation. The convention is to underline the terms that are dynamic[20]. When BTA is not used, the style is called *on-line* partial evaluation.

For example, the term

$$M = (\lambda y.\lambda x.\lambda f.f(+\ y\ 3)(+\ x\ 4))10$$

might be annotated as

$$M' = (\lambda y.\lambda x.\lambda f.f\underline{(+\ y\ 3)}(\underline{+}\ x\ \underline{4}))\underline{10}$$

if we told the binding time analyzer that y would be known.

The partial evaluator then gets M' and s as inputs. The parts that are underlined are *residualized.* They can be moved around like symbols or text, but not evaluated. Further, these underlined parts will appear in the output once the partial evaluation is finished. Only the parts that have not been underlined are evaluated.

[2]It can be worse, as some subexpressions, such as loops, may be unrolled before this "wrapping" occurs, further increasing the size of the target program.

[3]Some partial evaluators use more complex domains than this, including annotations such as "partially static".

The function `known` does not need to make recursive calls to P, all it has to do is read the annotation. Further, in the second projection $P'(P, M')$ the inner partial evaluator P is analyzed, and has access to the annotations in M' and therefore has the information it needs to decide the value for `known`. The resulting program $P_{M'}$ is therefore much smaller. The first partial evaluator to make use of this was Mix in 1984[15, 16].

The advantages of off-line partial evaluation are increased stability and control. Different strategies of partial evaluation can be tried by varying the algorithm used by the binding time analyzer, and many source-to-source transformations called "binding time improvements" have been published that increase the accuracy of BTA.

This increased stability is not free; the cost is lost opportunities for specialization. BTA is an approximation—many decisions have to be made about how to treat records that have some static data, and variables that will have either static or dynamic data during different points of the program's execution. Because an annotation of "static" is taken as a guarantee that the term will always be known, any doubt results in an annotation of "dynamic".

An example of this is given in [7]:

$$(\lambda f.(g\ (f\ d)\ f)\ \lambda a.a)$$

In this example g and d are dynamic variables. If we mark λf and $\lambda a.a$ as static, we will have the annotated term

$$(\lambda f.\underline{(g}\ (f\ \underline{d})\ f\underline{)}\ \lambda a.a)$$

the result will be the term

$$\underline{(g\ d}\ \lambda a.a\underline{)}$$

But this is incorrect, because $\lambda a.a$ is static, and should not appear in the final result. As a result, the $\lambda a.a$ will have to be marked dynamic, and the function call $(f\ d)$ left unfolded as $(\lambda a.a\ d)$. This kind of situation has been the inspiration for much research into *binding-time improvements*, source-to-source transformations that preserve the meaning of a term but allow for better results from the binding-time analysis.

Online partial evaluators re-evaluate the binding time of a term each time it is encountered, enabling them to take advantage of static data where an off-line partial evaluator would not have been able to. The previous example would have resulted in the term $(g\ d\ \lambda a.a)$ with no need for concern about the binding time of $\lambda a.a$.

Chapter 2

Fundamental Techniques

The purpose of this chapter is to present the ideas necessary to build a partial evaluator using the λ-calculus. For the λ-calculus, we use the following grammar:

$$\Lambda \ ::= \ (\Lambda_1 \ \cdots \ \Lambda_n) \mid \lambda x_1 \ldots x_n.\Lambda \mid x$$

This grammar is perhaps not the most common, but it is useful in that applications are easily noticed as they (and only they) are enclosed by parentheses.

2.1 Functional Representation

In order to represent programs in the λ-calculus, including integers, data-structures, and abstract syntax trees, we need to use functions. The techniques for these representations are well known, and we review them here for completeness. One important idea that needs to be emphasized is that *these functional representations are all in normal form.* The ability to represent arbitrary data by functions in normal form is an important part of the partial evaluators discussed here.

2.1.1 Church Numerals

Perhaps the best known functional representation is the Church Numeral[4]. A church numeral n is represented by the function $\lambda f\ x.(f(f(f \cdots (f\ x) \cdots)))$, where f is repeated n times. The Church numeral says that something should be repeated n times, and allows you to specify what that something is. Addition, multiplication, and exponentiation of Church numerals are all very straightforward.

$$\begin{aligned}\texttt{plus}\ m\ n &= \lambda fx.(m\ f\ (n\ f\ x))\\ \texttt{times}\ m\ n &= \lambda fx.(m\ (n\ f)\ x)\\ \texttt{power}\ m\ n &= \lambda fx.(m\ n\ f\ x)\end{aligned}$$

2.1.2 Disjoint Types and Pairs

Beyond integers, we can also represent arbitrary disjoint types[29]. Suppose we have a type S, with constructors $S_1 \dots S_n$. We can represent a term $t = S_i(t_1, t_2, \dots, t_m)$ by

$$\overline{t} = \lambda x_1 \dots x_n.(x_i\ \overline{t_1} \cdots \overline{t_m})$$

The x_i represent functions that specify what to do if the term t turns out to be an instance of the constructor S_i.

Booleans The boolean type has two constructors, `true` and `false`. They are represented as

$$\begin{aligned}\texttt{true} &= \lambda x_1 x_2.x_1\\ \texttt{false} &= \lambda x_1 x_2.x_2\end{aligned}$$

We can define functions on these boolean values by thinking in terms of what we want to do with them. For example, to define boolean `and` and `or`, we have the functions

$$\begin{aligned}\texttt{and} &= \lambda ab.(a\ b\ \texttt{false})\\ \texttt{or} &= \lambda ab.(a\ \texttt{true}\ b)\end{aligned}$$

To take the boolean conjunction of a and b, we apply a to b first; since if a is true, then b is the value we want to return. The second argument is `false`, which is the value of the expression should a turn out to be false.

Similarly, an `if` statement is defined by

$$\texttt{if} = \lambda cte.(c\ t\ e)$$

Where c is the boolean conditional, and t and e are the `then` and `else` branches, respectively.

Pairs The pair has one constructor with two arguments.

$$\begin{aligned}\texttt{pair}\ t_1\ t_2 &= \lambda x_1.(x_1\ t_1\ t_2)\\ \pi_1 &= \lambda p.(p\ \lambda xy.x)\\ \pi_2 &= \lambda p.(p\ \lambda xy.y)\end{aligned}$$

We have only one function, which tells us what we are supposed to do with the two components of the pair. It is interesting to note that the boolean values `true` and `false` correspond to the pair selection operations π_1 and π_2.

Recursive Types

This technique also generalizes to recursive types. The list type has two constructors, one that takes no arguments (`nil`) and one that takes two arguments (`cons`). Some standard definitions are

$$\begin{aligned}\texttt{nil} &= \lambda x_1 x_2.x_1\\ \texttt{cons}\ t_1\ t_2 &= \lambda x_1 x_2.(x_2\ t_1\ t_2)\\ \texttt{nil?} &= \lambda l.(l\ \texttt{true}\ \lambda ht.\texttt{false})\\ \texttt{car} &= \lambda l.(l\ \texttt{error}\ \lambda xy.x)\\ \texttt{cdr} &= \lambda l.(l\ \texttt{error}\ \lambda xy.y)\end{aligned}$$

The three "selector" functions are similar to those for pairs, except we also need to determine not only the contents of the list, but what kind of list is being examined. The `error` term can be anything that the user would like to return in the event of an error, such as taking the `car` or `cdr` of an empty list.

Similarly, binary trees also have two constructors. The empty tree takes zero arguments; and a branch takes three arguments, the data, the left child, and the right child.

$$\begin{aligned}
\texttt{empty} &= \lambda x_1 x_2 . x_1 \\
\texttt{branch}\ t_1\ t_2\ t_3 &= \lambda x_1 x_2 . (x_2\ t_1\ t_2\ t_3) \\
\texttt{empty?} &= \lambda l . (l\ \texttt{true}\ \lambda dht . \texttt{false}) \\
\texttt{getdata} &= \lambda l . (l\ \texttt{error}\ \lambda xyz . x) \\
\texttt{getleft} &= \lambda l . (l\ \texttt{error}\ \lambda xyz . y) \\
\texttt{getright} &= \lambda l . (l\ \texttt{error}\ \lambda xyz . z)
\end{aligned}$$

The balanced binary tree containing numerals 0–6 would look like this:

$$\begin{array}{l}
\lambda eb.(b\ \lambda fx.(f\ (f\ (f\ x))) \\
\quad \lambda eb.(b\ \lambda fx.(f\ x) \\
\quad\quad \lambda eb.(b\ \lambda fx.x\ \lambda eb.e\ \lambda eb.e) \\
\quad\quad \lambda eb.(b\ \lambda fx.(f\ (f\ x))\ \lambda eb.e\ \lambda eb.e)) \\
\quad \lambda eb.(b\ \lambda fx.(f\ (f\ (f\ (f\ (f\ x))))) \\
\quad\quad \lambda eb.(b\ \lambda fx.(f\ (f\ (f\ (f\ x))))\ \lambda eb.e\ \lambda eb.e) \\
\quad\quad \lambda eb.(b\ \lambda fx.(f\ (f\ (f\ (f\ (f\ (f\ x))))))\ \lambda eb.e\ \lambda eb.e)))
\end{array}$$

2.2 Higher Order Abstract Syntax

Abstract syntax trees are easily represented with this method, with an important modification. In the λ-calculus there are three kinds of terms. Application nodes take two arguments, one for the function and one for the argument. Abstraction and Variable nodes take one argument, the expression being represented. The variables are special: the argument to the constructor is the variable bound by the abstraction itself—this is what makes this syntax “higher order” [24, 18]. This has a major advantage; the variable scope rules are given to us for free by the underlying interpreter. Otherwise, it is very cumbersome to represent variables, since we would need to differentiate between variables explicitly, and provide our own function to handle variable substitution. The constructors for the abstract syntax tree are as follows:

$$\texttt{App}\ m\ n = \lambda abc.(a\ m\ n)$$
$$\texttt{Abs}\ \lambda x.m = \lambda abc.(b\ \lambda x.m)$$
$$\texttt{Var}\ x = \lambda abc.(c\ x)$$

The translation from λ-terms into higher order abstract syntax is as follows:

$$\lfloor x \rfloor = \lambda abc.(c\ x)$$
$$\lfloor M\ N \rfloor = \lambda abc.(a\ \lfloor t_1 \rfloor\ \lfloor t_2 \rfloor)$$
$$\lfloor \lambda x.M \rfloor = \lambda abc.(b\ \lambda x.\lfloor M \rfloor)$$

We will assume that all variables are bound. The translation scheme itself does not require it, but our evaluators would need a more complex representation to distinguish between free and bound variables otherwise.

For example, we can translate the expression $(\lambda q.q\ \lambda xw.(x\ w))$ as
$\lambda abc.(a\ \lambda abc.(b\ \lambda q.\lambda abc.(c\ q))\ \lambda abc.(b\ \lambda x.\lambda abc.(b\ \lambda w.\lambda abc.(a\ \lambda abc.(c\ x)\ \lambda abc.(c\ w))))))$

The expression has been represented by a function that takes three arguments: the first is a function that tells what we want done with application nodes, the second is a function that tells what we want done with abstractions, and the third is a function that tells what we want done with variables. As an example of how we could use such representations, suppose we wanted to find the size of an expression. The following term accomplishes that:

$$\begin{aligned}
\mathit{Size} &= (Y\ \lambda \mathit{size}\ m.(m \\
&\quad \lambda mn.(\texttt{plus one}\ (\texttt{plus}\ (\mathit{size}\ m)\ (\mathit{size}\ n))) \\
&\quad \lambda m.(\texttt{plus one}\ (\texttt{size}\ (m\ \texttt{one}))) \\
&\quad \lambda x.x)) \\
Y &= \lambda h.(\lambda x.(h\ (x\ x))\ \lambda x.(h\ (x\ x))) \\
\texttt{one} &= \lambda fx.(f\ x)
\end{aligned}$$

The Y combinator is the standard implementation of the fix-point operator. The second line of *Size* counts the size of application nodes (1 + size of both children), and similarly the third line counts the size of abstraction nodes. By passing the value `one` into the abstraction, variables will be assigned the value of 1 as a base case to the recursion. Therefore the function on the fourth line can just return the variable itself.

Similarly, if we wanted to count the number of occurrences of variables, we omit the `plus one` parts of the two functions.

$$\begin{aligned} CountVars \;=\; (Y\ \lambda countvars\ m.(m \\ &\lambda mn.(\texttt{plus}\ (countvars\ m)\ (countvars\ n)) \\ &\lambda m.(\texttt{countvars}\ (m\ \texttt{one})) \\ &\lambda x.x)) \end{aligned}$$

In addition to simple things like counting nodes, it is also possible to interpret expressions using this technique. Mogensen gives the following self-interpreter in [18].

$$\begin{aligned} E \;=\; (Y\ \lambda em.(m\quad &\lambda mn.((e\ m)\ (e\ n)) \\ &\lambda m.\lambda v.(e\ (m\ v)) \\ &\lambda x.x)) \end{aligned}$$

The second line contains a function that interprets applications by interpreting the parts and applying the results. The third line interprets functions by feeding an argument to the function and interpreting the result. The last line interprets variables, which represent themselves.

The input to this function is a term in higher order abstract syntax (HOAS), but the output is a raw, unrepresented λ-term. This term will have been reduced according to the reduction scheme of the base language, using the same parameter passing scheme and level of normalization.

2.3 Partial Evaluation

Most interpreters do not perform every possible computation in a term. Rather, they stop once the term is in Weak Head Normal Form (WHNF). In this form, the root of the term is either an abstraction or an application of a non-abstraction to a set of parameters, each of which are also in WHNF. An intuitive way of expressing this is to say that in WHNF we do not evaluate anything under a λ.

Normalization is a more aggressive form of interpretation, in which a term is first reduced to WHNF, and then all sub-terms are normalized. In contrast to WHNF, the reduction to normal form processes terms underneath a λ, and no reducible expressions will remain.

2.3.1 Normalization

In addition to a self-interpreter, Mogensen also presents a self-normalizer. One difference between the self-normalizer and the self-interpreter is that the self-normalizer returns its result in HOAS, instead of as an actual λ-calculus term. Representing the output in HOAS allows us to analyze the expression returned by the normalizer.

Outputting the result in HOAS causes the normalizer to be more complex than the self-interpreter. The reason for this complexity can be seen in the way an application $(M\ N)$ is handled. In the self-interpreter, we simply interpret M and N, and then apply one result to the other. Since the output is an unrepresented λ-term, we do not need to be concerned with precisely what kind of λ-term it is. But in normalization, we need to return a representation of the result, not just the result itself.

For the example of M applied to N, we have two possibilities of interest: M could normalize into a function, or into something else. If M normalizes to a function, we can apply that result to N as usual, and require the resulting term to be responsible for representing the result. In other words, we need to be able to produce an "executable" version of M. But, if M turns out not to be a function, the steps just described will not work. For this case, we need to return a representation of M applied to the normal form of N. In other words, we need to be able to produce a "textual" version of M. Because we cannot know beforehand whether M will reduce to a function or something else, for each term we need to be able to retrieve it in both executable form or represented form.

2.3.2 PEVs

To do this we need to make use of two techniques. The first technique is to represent λ terms as pairs; the first part of the pair will contain the function which reduces the term, and the second part of the pair contains the HOAS representation of it.

This idea is very significant, as it encapsulates a property of programs during partial evaluation: expressions undergoing partial evaluation are both code being executed and data being manipulated. Thus, the first half of the pair represents unfolding or specialization, and the second half of the pair represents residualization. In our research, we describe this type as a recursive type $PEV \equiv (PEV \rightarrow PEV) \times Exp$, where PEV stands for *Partial Evaluation Value.* For any PEV, we have the option of applying it as a function, or retrieving its representation.

The second technique is the recursive function D[1], which has the property that $((D\ \lceil x \rceil)\ \lceil y \rceil) \rightarrow (D\ \lceil (x\ y) \rceil)$. It has type $Exp \rightarrow PEV$, and is used to force the residualization of an expression. It is similar to the $\downarrow$ operator of Danvy's type directed partial evaluation, where we can take a live function and get back its text if we know the type[7]. The implementation of D is as follows:

$$D\ =\ (Y\ \lambda pm.\lambda x.(x\ \lambda v.(p\ \lambda abc.(b\ m\ (v\ \lambda ab.b)))\ m))$$

The variable p is the fix-point of D, and m is the argument, which we assume is a λ-term in HOAS. The $\lambda x.(x...)$ creates a pair/PEV. The first part is what to do if this term is applied to another. We capture the argument of such an application in v, and create a representation of m applied to the representation of v. (We apply v to F because we assume that v itself is also a PEV.) This representation of $(m\ v)$ is fed to a recursive call to D so that it can be applied to more arguments. The other half of the pair returned by D is just m itself.

The complete normalizer (modified to coincide more closely with the format of the partial evaluator) is as follows.

$$\begin{array}{lcl}
R & = & \lambda m.(R'\ m\ \lambda ab.b) \\
R' & = & (Y\ \lambda rm.(m \quad \lambda mn.((r\ m)\ \lambda abc.a\ (r\ n)) \\
 & & \qquad\qquad \lambda m.(\lambda gx.(x\ g\ \lambda abc.(c\ \lambda w.(g\ (D\ \lambda abc.(a\ w)\ \lambda ab.b)))) \\
 & & \qquad\qquad\quad \lambda v.(r\ (m\ v))) \\
 & & \qquad\qquad \lambda x.x)) \\
D & = & (Y\ \lambda pmx.(x\ \lambda v.(p\ \lambda abc.(b\ m\ (v\lambda ab.b)))\ m))
\end{array}$$

[1]In [18], this function is called P.

There are three functions passed to m in the R' combinator above, and they correspond to the functions in the self-interpreter. Each of these functions create a PEV out of a different kind of term. The first handles applications by normalizing the function m, taking the function part of the resulting PEV, and applying it to the normalized term n. The B combinator implements the first technique described above, in creating a PEV out of a function. The first half is the function g itself, while the second half is the same function that has been residualized by the D combinator.

2.3.3 Eliminating Recursion

In [19], Mogensen extends the idea of this normalizer to that of a self-applicable online partial evaluator. The reason that the normalizer is not self-applicable to begin with is that it does not have a normal form, which will cause self-application not to terminate.

The source of the problem is the two Y combinators, which are used for the recursions in the normalizer. The first Y combinator is used to propagate the three functions throughout the expression being normalized. The second Y combinator is used in the residualization-forcing function D, which builds successively larger applications as more arguments are given to it. As it turns out, both of these recursions can be eliminated.

In the case of the first recursion, notice that the same three functions are passed into every node, though in fact the representation would allow for a program which passed different functions to different nodes. This representation is more general than what is actually required. Rather than using recursion to propagate these functions, we can eliminate the recursion altogether and pass the combinators to all of the sub-terms simultaneously by using the following modified HOAS translation scheme $\lceil \cdot \rceil$.

$$\begin{aligned}
\lceil M \rceil &= \lambda\ a\ b\ .\ \lfloor M \rfloor \\
\lfloor x \rfloor &= x \\
\lfloor M\ N \rfloor &= a\ \lfloor t_1 \rfloor\ \lfloor t_2 \rfloor \\
\lfloor \lambda\ x\ .\ M \rfloor &= b\ (\lambda\ x\ .\ \lfloor M \rfloor)
\end{aligned}$$

This eliminates one source of recursion; the resulting normalizer follows. The initial Y is eliminated, further, the three functions now do not need to reference R directly as they did before, so we can define them as separate combinators.

$$\begin{aligned}
R &= \lambda m.(R'\ m\ \lambda ab.b) \\
R' &= \lambda m.(m\ AB) \\
A &= \lambda mn.((m\ \lambda ab.a)\ n) \\
B &= \lambda gx.(x\ g\ \lambda ab.(b\ \lambda w.(g\ ((D\ \lambda ab.w)\ \lambda ab.b)))\ m) \\
D &= (Y\ \lambda pmx.(x\ \lambda v.(p\ \lambda abc.(b\ m\ (v\lambda ab.b)))\ m))
\end{aligned}$$

The second problematic recursion is in the D combinator. The D combinator was first defined as

$$D \equiv (Y\ \lambda d.\lambda m.\lambda x.(x\ \lambda v.(d\ \lambda ab.(a\ (m\ a\ b)\ (v\ F\ a\ b)))\ m))$$

which Mogensen explains is a solution to the equation

$$D =_\beta \lambda m.\lambda x.(x\ \lambda v.(D\ \lambda ab.(a\ (m\ a\ b)\ (v\ F\ a\ b)))\ m)$$

Again, this recursion is more general than necessary, since we know that x will be a boolean value, and may not need to expand the recursive call to D. For an ordinary function x we would have to assume that the first argument is always used.

Therefore, we rewrite the equation as

$$D\ M\ X =_\beta (X\ \lambda v.(D\ \lambda ab.(a\ (M\ a\ b)\ (v\ F\ a\ b)))\ M)$$

where X is restricted to being either T or F. This means that X is a projection function, which will either select the first half of the pair (in which case we will need to expand the inner call to D) or else the second half, where we will not need D at all. Either way, we can wait until X is given before expanding D.

We can cause the recursion to occur inside the term rather than on the outer level by rewriting D as:

$$D \equiv (Q\ Q)$$

where

$$\begin{aligned} Q \equiv\ & \lambda q.\lambda m.\lambda x.(x\ \lambda q'.\lambda v.(q'\ q'\ \lambda ab.(a\ (m\ a\ b)\ (v\ F\ a\ b))) \\ & \lambda q'.m \\ & q). \end{aligned}$$

This new implementation of D has a normal form, and with that, so will the new normalizer. This normalizer can be turned into a partial evaluator by accepting two arguments rather than one in the main function.

$$P \equiv \lambda mn.(R\ \lambda ab.(a\ (m\ a\ b)\ (n\ a\ b))\ F)$$

This partial evaluator is self-applicable, and follows the behaviors specified by the Futamura projections. As we will show later, the evaluation still undergoes combinatorial code explosion, but because the partial evaluator is so small, we are able to contain the computation in the amount of memory available in today's computers. The full source is in figure 2.1.

With the recursion eliminated from the combinators, the actions taken by the combinators to effect partial evaluation are more apparent. Recalling that a PEV is a pair in which the first half is a function of type $PEV \to PEV$ and the second half is a HOAS represented expression, we can describe these combinators them here in terms of PEVs, .

$$\begin{aligned} &A : PEV \to PEV \to PEV \\ &B : (PEV \to PEV) \to PEV \\ &D : Exp \to PEV \end{aligned}$$

The A combinator takes two arguments m and n which will have been converted into PEVs by the previous actions of the partial evaluator. It then takes the first component of the function argument m and applies it to n, resulting in a new PEV. The function of the A combinator, then, is to perform the application of any two PEVs given to it. Note that it will always choose to specialize, never to residualize.

The B combinator converts an abstraction g into a PEV. The first component of the PEV should be the function g itself—since the body of g has been converted into a PEV, this gives us the correct type.

$$
\begin{array}{lcl}
T & = & \lambda ab.a \\
F & = & \lambda ab.b \\
Q & = & \lambda x.(x\ \lambda q'v.(q'\ q'\ \lambda ab.(a\ (m\ a\ b)\ (v\ F\ a\ b))) \\
 & & \qquad\quad \lambda q'.m \\
 & & \qquad\quad q) \\
D & = & (Q\ Q) \\
A & = & \lambda mn.(m\ T\ n) \\
B & = & \lambda gx.(x\ g\ \lambda ab.(b\ \lambda w.(g\ (D\ \lambda ab.w)\ F\ a\ b))) \\
R & = & \lambda m.(m\ A\ B\ F) \\
P & = & \lambda mn.(R\ \lambda ab.(a\ (m\ a\ b)\ (n\ a\ b)))
\end{array}
$$

Figure 2.1: Mogensen's Partial Evaluator

In order to form the second component it is necessary to use the D combinator. The D combinator is the residualization operator. It takes an expression as an argument and returns a PEV, with the property that for any expression w and $PEV\ M$, $((D\ w)_1\ M) \Rightarrow (D\ \lfloor(w\ M_2)\rfloor$. The action of D is to produce a PEV that can only be residualized, even if we take its first component.

The B combinator creates a new abstraction for w, applies D to w, and then applies g to the result. Given the property of D mentioned above, this causes a residualization of the entire body of g. The application to $F\ a\ b$ seen at the end is there to select out the expression component of the resulting PEV, and then to perform book-keeping with the HOAS.

Examining the code for Mogensen's evaluator, in particular, the A combinator, it becomes clear that there is little to control the progress of the partial evaluation. *All* applications are performed. This is an effect of the use of this particular HOAS: all applications share the same representation, and so all applications will be handled using the A combinator. The subject of this thesis is how to add control mechanisms to the partial evaluator, and how to expand these techniques to larger languages.

2.4 Related Work

Online partial evaluators are often considered not to be self-applicable in practice, due to the combinatorial explosion that results when self-application is tried. However, it is possible to self-apply successfully without binding time analysis. In 1991, Glück gave an example of a self-applicable partial evaluator that uses Turchin's principle of metasystem translation[12].

In chapter 7 of [26], Ruf also discusses techniques for generating code generators without BTA by using special data structures which allow the partial evaluator to infer binding time information at run-time. These techniques do not allow self-application, however.

In [30] Peter Theimann published a code generator that made use of HOAS. Like Mogensen's evaluator, Theimann's is very small ("six lines"). The target program itself becomes its own code generator, i.e., the code generator has been inlined. Beyond that, the two systems are quite different: Theimann's evaluator is a multi-level code generator, is off-line, and is written in continuation passing style.

These HOAS-based partial evaluators are interesting in that they tend to be very small, and have a simple structure. In effect, they cause the partial evaluator to be *inlined* into the code being specialized, creating a self-specializing program. This simplicity is also their greatest limitation—there is no mechanism to control the partial evaluation process. In [5], we published a modification of Mogensen's partial evaluator, in which we introduced strategies as a control mechanism. While we were able to demonstrate some of the tradeoffs and control that strategies have the potential to offer, we were not able to stabilize the third projection.

Chapter 3

Strategies

3.1 Strategy Based Partial Evaluator

Examining the code for Mogensen's evaluator, in particular, the A combinator, it becomes clear that there is little to control the progress of the partial evaluation. *All* applications are performed. This is an effect of the use of this particular HOAS: all applications share the same representation, and so all applications will be handled using the same A combinator.

The decision about whether an application should be taken is made when the A combinator selects the first component of its first argument. Hypothetically, we could instead use this version of A, which always residualizes application.

$$A \;=\; \lambda mn.(D\ \lambda ab.(a\ (m\ F\ a\ b)\ (n\ F\ a\ b)))$$

This would cause the partial evaluator to return its argument unchanged.

To add control, we need to have a more sophisticated decision mechanism. Suppose we have a function s which could take two PEVs, decide whether they should be applied or residualized, and then perform that operation for us. Then we could rewrite the A combinator. Instead of $A \equiv \lambda mn.(m\ T\ n)$, we would have $A \equiv \lambda mns.(s\ m\ n)$. This function s is called a *strategy*.

The effect of Mogensen's partial evaluator is to convert a source file into a function which expands itself as much as possible. With strategies, we will convert a source file into a function which will expand itself according to the advice given to it. For our initial discussion, we will consider the advice given by the strategies to be compulsory. The partial evaluator creates this self-expanding function and the strategy decides how far the expansion should go.

In fact, the addition of strategies to the partial evaluator suggests a number of benefits. First, we will have a mechanism to control the inlined partial evaluator, as we have already explained. One of the interesting features of the Mogensen evaluator is its simplicity, and the addition of strategies will detract only minimally from that simplicity. Another benefit of strategies is the modularity they bring to the partial-evaluation process. This modularity comes in two ways. First, it separates the decision-making code from the rest of the partial evaluator. Second, as we will show later, strategies can be composed, allowing us to combine two or more partial evaluation techniques according to our specifications.

The combination of the modularity and the simplicity of the underlying evaluator suggests that it may be possible to have some of the benefits of online evaluators (such as accuracy and simplicity) and the benefits of offline evaluators (self-application). For example, we could have strategies that made use of a binding-time analysis, but could selectively "override" a decision made by the BTA to take advantage of opportunities for specialization that otherwise would have been missed. Another way this could happen is by putting complicated, unstable code inside the strategy so that self-application will not be affected by it.

The addition of strategies changes the type of PEVs:

$$\begin{aligned} PEV &\equiv \text{Strategy} \rightarrow \text{Result} \\ \text{Result} &\equiv (PEV \rightarrow \text{Result}) \times \text{Exp} \\ \text{Strategy} &\equiv PEV \rightarrow PEV \rightarrow \text{Result} \end{aligned}$$

Initially, a PEV asks for a strategy to tell it when the terms it encodes should be specialized and when they should be residualized. The result then becomes the pair representing both function and data. From the type you can see that a strategy is propagated to the sub-terms of a PEV.

As a running example, consider the terms $V = \lambda x.(\lambda y.(y\ x)\ \lambda z.z)$ and $I = \lambda q.q$. Applying V to I would reduce to $\lambda q.q$. This is also the result if we use Mogensen's partial evaluator:

$$P\ V\ I \Rightarrow Abs(\lambda q.q)$$

$$
\begin{array}{lcl}
P & = & \lambda mns.(R\ \lambda ab.(a\ (m\ a\ b)\ (n\ a\ b))\ s\ F) \\
R & = & \lambda m.(m\ A\ B) \\
B & = & \lambda gs.\lambda x.(x \\
 & & \qquad\qquad \lambda v.(g\ v\ s) \\
 & & \qquad\qquad \lambda ab.(b\ \lambda z.(g\ (D\ \lambda ab.z)\ s\ F\ a\ b))) \\
A & & \lambda mns.(s\ m\ n) \\
D & = & (Q\ Q) \\
Q & = & \lambda qvs.\lambda x.(x \\
 & & \qquad\qquad \lambda q'w.(q'\ q'\ \lambda ab.(a\ (v\ a\ b)\ (w\ s\ F\ a\ b))\ s) \\
 & & \qquad\qquad \lambda q'.v \\
 & & \qquad q) \\
T & & \lambda ab.a \\
F & & \lambda ab.b
\end{array}
$$

Figure 3.1: Strategy-based partial evaluator

3.2 Simple Strategies

The simplest strategies to write are those that make the same decision in all circumstances. There are two of these, called *Expand All*, which always instructs the partial evaluator to reduce, and *Expand None*, which always instructs the evaluator to residualize. In the following discussion, let $\ll \cdot \gg$ denote a PEV, and subscripts 1 and 2 denote taking the first or second component of a PEV. Further, we will use constructors *App* and *Abs* rather than explicit higher order abstract syntax.

3.2.1 Expand All

The *Expand All* strategy, or Σ_{All}, has the following definition.

$$\Sigma_{All} \equiv \lambda m\ n.((m\ \Sigma_{All})_1\ n)$$

The Σ_{All} strategy is passed to the first term m, producing a Result. Then it takes the function component and applies that to n, producing another Result.

To implement this in λ-calculus, we η-contract away the n and use the Y combinator to handle the recursion.

$$\Sigma_{All} \equiv (Y\ \lambda\sigma m.(m\ \sigma\ T))$$

The strategy partial evaluator P^{σ}, given Σ_{All}, will behave almost exactly like Mogensen's evaluator.

$$\begin{array}{rcl}
P^{\sigma}\ [\![\lambda x.(\lambda y.(y\ x)\ \lambda z.z)]\!]\ [\![\lambda q.q]\!]\ \Sigma_{All} & \Rightarrow & \\
(\ll (\lambda x.(\lambda y.(y\ x)\ \lambda z.z)\ \lambda q.q) \gg\ \Sigma_{All})_2 & \Rightarrow & \\
(\Sigma_{All}\ \ll (\lambda x.(\lambda y.(y\ x)\ \lambda z.z)) \gg\ \ll \lambda q.q \gg)_2 & \Rightarrow & \\
((\ll (\lambda x.(\lambda y.(y\ x)\ \lambda z.z)) \gg\ \Sigma_{All})_1\ \ll \lambda q.q \gg)_2 & \Rightarrow & \\
(\lambda x.(\ll (\lambda y.(y\ x)\ \lambda z.z) \gg\ \Sigma_{All})\ \ll (\lambda q.q) \gg)_2 & \Rightarrow & \\
(\ll (\lambda y.(y\ \lambda q.q)\ \lambda z.z) \gg\ \Sigma_{All})_2 & \Rightarrow & \\
(\Sigma_{All}\ \ll \lambda y.(y\ \lambda q.q) \gg\ \ll \lambda z.z \gg)_2 & \Rightarrow & \\
((\ll \lambda y.(y\ \lambda q.q) \gg\ \Sigma_{All})_1\ \ll \lambda z.z \gg)_2 & \Rightarrow & \\
(\lambda y.(\ll (y\ (\lambda q.q)) \gg\ \Sigma_{All})\ \ll \lambda z.z \gg)_2 & \Rightarrow & \\
(\ll (\lambda z.z\ \lambda q.q) \gg\ \Sigma_{All})_2 & \Rightarrow & \\
(\Sigma_{All}\ \ll \lambda z.z \gg\ \ll \lambda q.q \gg)_2 & \Rightarrow & \\
((\ll \lambda z.z \gg\ \Sigma_{All})_1\ \ll \lambda q.q \gg)_2 & \Rightarrow & \\
(\lambda z.(\ll z \gg\ \Sigma_{All})\ \ll \lambda q.q \gg)_2 & \Rightarrow & \\
(\ll \lambda q.q \gg\ \Sigma_{All})_2 & \Rightarrow & \\
[\![\lambda q.q]\!] & \equiv & Abs(\lambda q.q)
\end{array}$$

The Σ_{All} strategy is passed into the sub-terms of the input, causing all the computations to be done.

3.2.2 Expand None

The *Expand None* strategy, written Σ_{None}, tells the partial evaluator not to perform any β-reductions. It has the following definition.

$$\Sigma_{None} \equiv \lambda m.(D\ (m\ \Sigma_{None})_2\ \Sigma_{None})_1$$

The Σ_{None} strategy takes a PEV m, applies it to Σ_{None} and takes the expression component. This is passed into the D operator, which returns a PEV. In order to make the types match, we pass Σ_{None} to the PEV[1] and then take the first component.

$$\begin{array}{rl}
P^{\sigma}\ [\![\lambda x.(\lambda y.(y\ x)\ \lambda z.z)]\!]\ [\![\lambda q.q]\!]\ \Sigma_{None} & \Rightarrow \\
(\ll (\lambda x.(\lambda y.(y\ x)\ \lambda z.z)\ \lambda q.q) \gg\ \Sigma_{None})_2 & \Rightarrow \\
(\Sigma_{None}\ \ll \lambda x.(\lambda y.(y\ x)\ \lambda z.z) \gg \ll \lambda q.q \gg)_2 & \Rightarrow \\
((D\ (\ll \lambda x.(\lambda y.(y\ x)\ \lambda z.z) \gg\ \Sigma_{None})_2\ \Sigma_{None})_1\ \ll \lambda q.q \gg)_2 & \Rightarrow\ \cdots \\
((D\ Abs(\lambda x.App(Abs(\lambda y.App(y,y)), Abs(\lambda z.z)))\ \Sigma_{None})_1\ \ll \lambda q.q \gg)_2 & \Rightarrow \\
(D\ App(Abs(\lambda x.App(Abs(\lambda y.App(y,y)), Abs(\lambda z.z))), Abs(\lambda q.q))\ \Sigma_{None})_2 & \Rightarrow \\
App(Abs(\lambda x.App(Abs(\lambda y.App(y,y)), Abs(\lambda z.z))), Abs(\lambda q.q)) &
\end{array}$$

The process is similar to that of Σ_{All}, but the expression component is taken instead of the function component, and the D combinator creates a residualizing PEVs out of the expressions.

3.3 Composing Strategies

Both Σ_{All} and Σ_{None} pass copies of themselves to the sub-terms of the PEVs on which they operate. There is no reason the strategy passed to a sub-term must be the same one, however. By having strategies pass different strategies, we can produce different partial evaluation results. Further, this mechanism allows us to compose strategies.

[1]Due to the definition of D, it really doesn't matter which strategy is passed to it; all will have the same result.

3.3.1 Expand N

A simple composition is *Expand N*, which allows the user to specify how many levels of β-reductions to perform.

$$\Sigma_n \equiv ((n\ \lambda\sigma m.(m\ \sigma)_1)\ \Sigma_{None}),\ \text{where } n \text{ is some Church numeral}$$

The Church numeral n indicates the number of levels to expand. This numeral is applied to a term that is identical to the body of the Σ_{All} strategy. (In the case of Σ_{All}, the Y combinator[2] makes an infinite chain of instructions to expand.) In the Σ_n case, this term is duplicated only n times, and after all these copies are passed into some PEV, the final PEV is applied to Σ_{None} to stop any further expansion.

In this example we use Σ_2 to expand only the first two levels of β-reductions.

$$\begin{array}{rcl}
(P^{\sigma}\ [\![\lambda x.(\lambda y.(y\ x)\ \lambda z.z)]\!]\ [\![\lambda q.q]\!]\ \Sigma_2) & \Rightarrow & \\
(\ll (\lambda x.(\lambda y.(y\ x)\ \lambda z.z)\ \lambda q.q) \gg\ \Sigma_2)_2 & \Rightarrow & \\
(\Sigma_2\ \ll \lambda x.(\lambda y.(y\ x)\ \lambda z.z) \gg\ \ll \lambda q.q \gg)_2 & \Rightarrow & \\
((\ll \lambda x.(\lambda y.(y\ x)\ \lambda z.z) \gg\ \Sigma_1)_1\ \ll \lambda q.q \gg)_2 & \Rightarrow & \\
(\lambda x.(\ll (\lambda y.(y\ x)\ \lambda z.z) \gg\ \Sigma_1)\ \ll \lambda q.q \gg)_2 & \Rightarrow & \\
(\ll (\lambda y.(y\ \lambda q.q)\ \lambda z.z) \gg\ \Sigma_1))_2 & \Rightarrow & \\
(\Sigma_1\ \ll \lambda y.(y\ \lambda q.q) \gg\ \ll \lambda z.z \gg)_2 & \Rightarrow & \\
((\ll \lambda y.(y\ \lambda q.q) \gg\ \Sigma_{None})_1\ \ll \lambda z.z \gg)_2 & \Rightarrow & \\
(\lambda y. \ll (y\ \lambda q.q) \gg\ \Sigma_{None}\ \ll \lambda z.z \gg)_2 & \Rightarrow & \\
(\ll (\lambda z.z\ \lambda q.q) \gg\ \Sigma_{None})_2 & \Rightarrow & \\
(\Sigma_{None}\ \ll \lambda z.z \gg\ \ll \lambda q.q \gg)_2 & \Rightarrow & \cdots \\
(D[\![(\lambda z.z\ \lambda q.q)]\!]\ \Sigma_{None})_2 & \Rightarrow & \\
[\![(\lambda z.z\ \lambda q.q)]\!] & \equiv & App(Abs(\lambda z.z), Abs(\lambda q.q))
\end{array}$$

[2] We can think of the Y combinator as an implementation of the Church numeral for infinity.

3.3.2 Expand Below N

Using a similar technique we can write a strategy called *Expand Below N*, which residualizes the topmost β-reductions, but performs the ones lower than them in the tree.

$$\Sigma_{below-n} \equiv \lambda\sigma m.(n\ (D\ (m\ \sigma)_2\ \sigma)_1\ \Sigma_{all}),\ \text{where } n \text{ is some Church numeral}$$

This strategy $\Sigma_{below-n}$ passes $\Sigma_{below-(n-1)}$ to subterms, and residualizes the result. When n reaches zero, Σ_{all} is passed instead, which specializes everything below it.

$$\begin{aligned}
(P^{\sigma}\ [\![\lambda x.(\lambda y.(y\ x)\ \lambda z.z)]\!]\ [\![\lambda q.q]\!]\ \Sigma_{<1}) &\Rightarrow \\
(\ll (\lambda x.(\lambda y.(y\ x)\ \lambda z.z)\ \lambda q.q) \gg\ \Sigma_{<1})_2 &\Rightarrow \\
(\Sigma_{<1}\ \ll \lambda x.(\lambda y.(y\ x)\ \lambda z.z) \gg \ll \lambda q.q \gg)_2 &\Rightarrow \\
(D\ \mathit{App}((\ll \lambda x.(\lambda y.(y\ x)\ \lambda z.z) \gg\ \Sigma_{All})_2, (\ll \lambda q.q \gg\ \Sigma_{<1})_2)\ \mathit{Ex})_2 &\Rightarrow \quad \cdots \\
(D\ \mathit{App}(\mathit{Abs}(x,x), \mathit{Abs}(q,q))\ \mathit{Ex})_2 &\Rightarrow \\
\mathit{App}(\mathit{Abs}(x,x), \mathit{Abs}(q,q))
\end{aligned}$$

The Σ_x is to indicate that the choice of strategy does not matter.

This technique of composition gives rise to a class of strategies which make decisions based on their own structure, but do not look at the content or structure of the terms being specialized. Neither the simple strategies presented above nor these structured strategies are able to make decisions based on the terms being processed. It is possible, though, to build a customized strategy based on the structure of the term to be processed.

3.4 Propagation

Looking closely at the expansions of Σ_n and $\Sigma_{<n}$, it becomes apparent that "level" does not necessarily mean "distance from the root node". The level is decremented each time the strategy is

passed to the function part of an application. The argument will be folded into the term and given a strategy perhaps much later in the computation. If the strategy reaches Σ_{None}, the D combinator passes this strategy to the arguments of the residualized PEV, causing all terms to be residualized.

For example, consider the following reduction, where each of the variables I_a through I_h represent the identity function:

$$\begin{array}{rl}
(P^{\sigma}\ [\![(((I_a\ I_b)\ (I_c\ I_d))\ ((I_e\ I_f)\ (I_g\ I_h)))]\!]\ \Sigma_2) & \Rightarrow \\
(\Sigma_2\ \ll ((I_a\ I_b)\ (I_c\ I_d)) \gg\ \ll ((I_e\ I_f)\ (I_g\ I_h)) \gg)_2 & \Rightarrow \\
((\ll ((I_a\ I_b)\ (I_c\ I_d)) \gg\ \Sigma_1)_1\ \ll ((I_e\ I_f)\ (I_g\ I_h)) \gg)_2 & \Rightarrow \\
((\Sigma_1\ \ll (I_a\ I_b) \gg\ \ll (I_c\ I_d) \gg)_1\ \ll ((I_e\ I_f)\ (I_g\ I_h)) \gg)_2 & \Rightarrow \\
(((\ll (I_a\ I_b) \gg\ \Sigma_0)_1\ \ll (I_c\ I_d) \gg)_1\ \ll ((I_e\ I_f)\ (I_g\ I_h)) \gg)_2 & \Rightarrow \\
(((\Sigma_0\ \ll I_a \gg\ \ll I_b \gg)_1\ \ll (I_c\ I_d) \gg)_1\ \ll ((I_e\ I_f)\ (I_g\ I_h)) \gg)_2 & \Rightarrow \\
(((I_D\ App([\![I_a]\!],[\![I_b]\!])\ \Sigma_0)_1\ \ll (I_c\ I_d) \gg)_1\ \ll ((I_e\ I_f)\ (I_g\ I_h)) \gg)_2 & \Rightarrow \\
((I_D\ App(I_{A}pp([\![I_a]\!],[\![I_b]\!]),(\ll (I_c\ I_d) \gg\ \Sigma_0)_2)\ \Sigma_0)_1\ \ll ((I_e\ I_f)\ (I_g\ I_h)) \gg)_2 & \Rightarrow \\
((I_D\ App(I_{A}pp([\![I_a]\!],[\![I_b]\!]),(\Sigma_0\ \ll I_c \gg\ \ll I_d \gg)_2)\ \Sigma_0)_1\ \ll ((I_e\ I_f)\ (I_g\ I_h)) \gg)_2 & \Rightarrow \\
((I_D\ App(I_{A}pp([\![I_a]\!],[\![I_b]\!]),App([\![I_c]\!],[\![I_d]\!]))\ \Sigma_0)_1\ \ll ((I_e\ I_f)\ (I_g\ I_h)) \gg)_2 & \Rightarrow \\
(I_D\ App(I_{A}pp(I_{A}pp([\![I_a]\!],[\![I_b]\!]),App([\![I_c]\!],[\![I_d]\!])),(\ll ((I_e\ I_f)\ (I_g\ I_h)) \gg\ \Sigma_0)_2)\ \Sigma_0)_2 & \Rightarrow^{+} \\
(I_D\ App(I_{A}pp(I_{A}pp([\![I_a]\!],[\![I_b]\!]),App([\![I_c]\!],[\![I_d]\!])),App(I_{A}pp([\![I_e]\!],[\![I_f]\!]),App([\![I_g]\!],[\![I_h]\!])))\ \Sigma_0)_2 & \Rightarrow \\
App(I_{A}pp(I_{A}pp([\![I_a]\!],[\![I_b]\!]),App([\![I_c]\!],[\![I_d]\!])),App(I_{A}pp([\![I_e]\!],[\![I_f]\!]),App([\![I_g]\!],[\![I_h]\!]))) &
\end{array}$$

The propagation can be visualized in figure 3.2.

The strategy decrements itself as it travels using something similar to a depth-first traversal down the left side of the tree, and becomes Σ_0 before it reaches I_a. The D combinator then passes Σ_0 across to the right side of the tree.

The consequences can be seen if we replace $((I_e\ I_f)\ (I_g\ I_h))$ with a smaller term, say $(I_e\ I_f)$, when we get the following reduction.

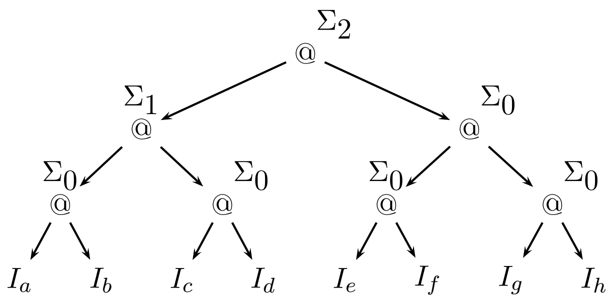

Figure 3.2: Propagation of *Expand-n* through a large tree

$$
\begin{aligned}
(P^{\sigma}\ \llbracket(((I_a\ I_b)\ (I_c\ I_d))\ (I_e\ I_f))\rrbracket\ \Sigma_2) &\Rightarrow \\
(\Sigma_2\ \ll ((I_a\ I_b)\ (I_c\ I_d)) \gg\ \ll (I_e\ I_f) \gg)_2 &\Rightarrow \\
((\ll ((I_a\ I_b)\ (I_c\ I_d)) \gg\ \Sigma_1)_1\ \ll (I_e\ I_f) \gg)_2 &\Rightarrow \\
((\Sigma_1\ \ll (I_a\ I_b) \gg\ \ll (I_c\ I_d) \gg)_1\ \ll (I_e\ I_f) \gg)_2 &\Rightarrow \\
(((\ll (I_a\ I_b) \gg\ \Sigma_0)_1\ \ll (I_c\ I_d) \gg)_1\ \ll (I_e\ I_f) \gg)_2 &\Rightarrow \\
(((\Sigma_0\ \ll I_a \gg\ \ll I_b \gg)_1\ \ll (I_c\ I_d) \gg)_1\ \ll (I_e\ I_f) \gg)_2 &\Rightarrow \\
(((I_D\ \mathit{App}(\llbracket I_a\rrbracket, \llbracket I_b\rrbracket)\ \Sigma_0)_1\ \ll (I_c\ I_d) \gg)_1\ \ll (I_e\ I_f) \gg)_2 &\Rightarrow \\
((I_D\ \mathit{App}(I_A\mathit{pp}(\llbracket I_a\rrbracket, \llbracket I_b\rrbracket), (\ll (I_c\ I_d) \gg\ \Sigma_0)_2)\ \Sigma_0)_1\ \ll (I_e\ I_f) \gg)_2 &\Rightarrow \\
((I_D\ \mathit{App}(I_A\mathit{pp}(\llbracket I_a\rrbracket, \llbracket I_b\rrbracket), (\Sigma_0\ \ll I_c \gg\ \ll I_d \gg)_2)\ \Sigma_0)_1\ \ll (I_e\ I_f) \gg)_2 &\Rightarrow \\
((I_D\ \mathit{App}(I_A\mathit{pp}(\llbracket I_a\rrbracket, \llbracket I_b\rrbracket), \mathit{App}(\llbracket I_c\rrbracket, \llbracket I_d\rrbracket))\ \Sigma_0)_1\ \ll (I_e\ I_f) \gg)_2 &\Rightarrow \\
(I_D\ \mathit{App}(I_A\mathit{pp}(I_A\mathit{pp}(\llbracket I_a\rrbracket, \llbracket I_b\rrbracket), \mathit{App}(\llbracket I_c\rrbracket, \llbracket I_d\rrbracket)), (\ll (I_e\ I_f) \gg\ \Sigma_0)_2)\ \Sigma_0)_2 &\Rightarrow\ \cdots \\
(I_D\ \mathit{App}(I_A\mathit{pp}(I_A\mathit{pp}(\llbracket I_a\rrbracket, \llbracket I_b\rrbracket), \mathit{App}(\llbracket I_c\rrbracket, \llbracket I_d\rrbracket)), \mathit{App}(\llbracket I_e\rrbracket, \llbracket I_f\rrbracket))\ \Sigma_0)_2 &\Rightarrow \\
\mathit{App}(I_A\mathit{pp}(I_A\mathit{pp}(\llbracket I_a\rrbracket, \llbracket I_b\rrbracket), \mathit{App}(\llbracket I_c\rrbracket, \llbracket I_d\rrbracket)), \mathit{App}(\llbracket I_e\rrbracket, \llbracket I_f\rrbracket)) &
\end{aligned}
$$

Even though $(I_e\ I_f)$ was less than two levels from the root, it was still residualized, since it was more than two applications away from the start of the reduction.

The traversal becomes more complex in the presence of abstractions. The strategy is still propagated using a depth-first search, but the tree over which it traverses is changing dynamically. Consider, for example, the expression ($\lambda x.(x\ M)\ (a\ b)$), illustrated in figure 3.3. The strategy is passed down the left side of the tree, but part of the right side (marked by Σ_2) receives the strategy before the M term does, because it is passed into the tree by the actions of the node processed by Σ_3. One implication of this is that part of the body of a function may be left unprocessed if the argument to that function is sufficiently large.

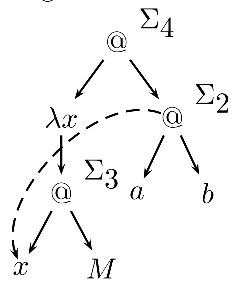

Figure 3.3: Propagation of *Expand-n* through a changing tree

There are a few things we can do to influence the propagation of the strategy. The strategies listed to this point have all passed a strategy into the function, and then passed the argument into the result. We could have the strategy pass itself into the argument as well, if we can find a way to turn that *Result* back into a PEV.

The most straightforward way to do this is to place the result inside an abstraction. The abstraction would capture any strategy that was given to it in the future, and return the result that was generated by the strategy that was given to it initially.

We can rewrite the *Expand N* strategy like this:

$$\Sigma_{bn} \equiv (n\ \lambda\sigma pq.((p\ \sigma)_1\ (\lambda x.(q\ \sigma)))\Sigma_{None}),\ \text{where } n \text{ is some Church numeral}$$

This causes the current strategy to be passed to both sides of the tree, rather than just the first. This causes a breadth-first traversal to occur. Figures 3.2 now looks like figure 3.4. The right child of the root node was processed using Σ_1, instead of Σ_0.

Further, the expression in figure 3.3 will be processed as in figure 3.5, in that the argument passed into x will be pre-processed by Σ_3, and will throw away the Σ_2 when it is given to the result.

This method of changing the behavior of a PEV is interesting because it, in effect, allows

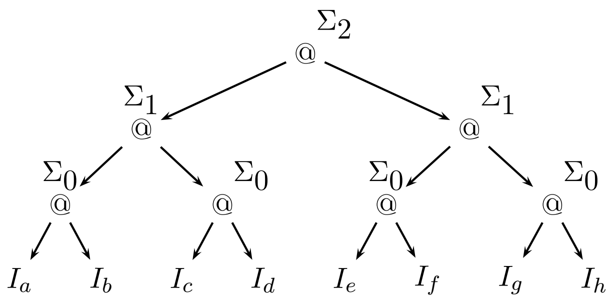

Figure 3.4: Propagation of *Expand-Breadth-n* through a large tree

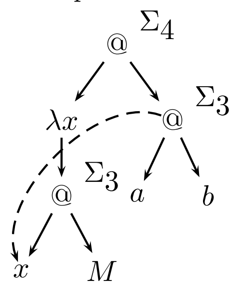

Figure 3.5: Propagation of *Expand-Breadth-n* through a changing tree

a PEV to ignore a strategy that has been given to it. It also allows us finer control over the propagation of strategies. The Σ_n strategies propagate according to the execution of a program, but the Σ_{bn} strategies propagate according to the structure of a program's abstract syntax tree.

3.5 Experimental Results

In order to observe the effect that strategies have on the partial evaluation process, we applied the partial evaluator to three programs using a variety of strategies. The three programs we used were Ackermann's function, the exponentiation function, and a small interpreter. The interpreter and the exponentiation function are standard examples used for partial evaluators. Interpreters show the effect of self-application. Ackermann's function is used as a running example in Mogensen's work, and therefore makes a good basis for comparison. All three of these programs use Church numerals in this section, and will be expanded in the following chapters to use integers when we expand the language.

The Church numeral version of Ackermann's function is given by

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

Ackermann's function is only practically applied to C_2 or C_3; at C_4 it becomes prohibitively expensive to run[3]. Figure 3.6 show the results of running Ackermann's function with C_2 and C_3 as the first argument. The β column represents the number of β reductions required.

Expression	β	Expression	β
$Ack\ C_2\ C_0$	16	$Ack\ C_3\ C_0$	33
$Ack\ C_2\ C_1$	30	$Ack\ C_3\ C_1$	173
$Ack\ C_2\ C_2$	50	$Ack\ C_3\ C_2$	833
$Ack\ C_2\ C_3$	76	$Ack\ C_3\ C_3$	3685
$Ack\ C_2\ C_4$	108	$Ack\ C_3\ C_4$	15,529
$Ack\ C_2\ C_5$	146	$Ack\ C_3\ C_5$	63,789
$Ack\ C_2\ C_6$	190	$Ack\ C_3\ C_6$	258,609
$Ack\ C_2\ C_7$	240	$Ack\ C_3\ C_7$	1,041,461

Figure 3.6: Ackermann's function timings

Mogensen's partial evaluator is able to specialize Ackermann's function, which gives some improvement. We use M to denote Mogensen's evaluator, and show the results in figure 3.7.

Expression	β	Speedup (β)	Expression	β	Speedup (β)
$M\ Ack\ C_2$	492	n/a	$M\ Ack\ C_3$	971	n/a
$Ack_2\ C_0$	8	2.0	$Ack_3\ C_0$	22	1.5
$Ack_2\ C_1$	21	1.43	$Ack_3\ C_1$	155	1.12
$Ack_2\ C_2$	40	1.2	$Ack_3\ C_2$	800	1.04
$Ack_2\ C_3$	65	1.17	$Ack_3\ C_3$	3621	1.02
$Ack_2\ C_4$	96	1.125	$Ack_3\ C_4$	15,402	1.01
$Ack_2\ C_5$	133	1.1	$Ack_3\ C_5$	63,535	1.00
$Ack_2\ C_6$	176	1.08	$Ack_3\ C_6$	258,100	1.00
$Ack_2\ C_7$	225	1.07	$Ack_3\ C_7$	1,040,441	1.00

Figure 3.7: M applied to Ackermann's function

Figure 3.8 shows the corresponding computations using the strategy evaluator (which we will denote P) and the *Expand All* strategy. Though it takes more β-reductions to generate Ack_2 and

[3]Here C_n represents the Church numeral for n

Ack$_3$, they produce the same results as in figure 3.7.

Expression	β	Speedup (β)	Expression	β	Speedup (β)
$P\ Ack\ C_2\ \Sigma_{All}$	709	n/a	$P\ Ack\ C_3\ \Sigma_{All}$	1401	n/a
$Ack_2\ C_0$	8	2.0	$Ack_3\ C_0$	22	1.5
$Ack_2\ C_1$	21	1.43	$Ack_3\ C_1$	155	1.12
$Ack_2\ C_2$	40	1.2	$Ack_3\ C_2$	800	1.04
$Ack_2\ C_3$	65	1.17	$Ack_3\ C_3$	3621	1.02
$Ack_2\ C_4$	96	1.125	$Ack_3\ C_4$	15,402	1.01
$Ack_2\ C_5$	133	1.1	$Ack_3\ C_5$	63,535	1.00
$Ack_2\ C_6$	176	1.08	$Ack_3\ C_6$	258,100	1.00
$Ack_2\ C_7$	225	1.07	$Ack_3\ C_7$	1,040,441	1.00

Figure 3.8: *P* and *Expand All* applied to Ackermann's function

If we use *Expand None*, we get the same results that we would get if *Ack* were left unexpanded. This is shown in figure 3.9. It takes far fewer β-reductions to generate Ack_{Cn} with this strategy than it does with *Expand All*.

Expression	β	Speedup (β)	Expression	β	Speedup (β)
$P\ Ack\ C_2\ \Sigma_{None}$	446	n/a	$P\ Ack\ C_3\ \Sigma_{None}$	473	n/a
$Ack_2\ C_0$	16	1.00	$Ack_3\ C_0$	33	1.00
$Ack_2\ C_1$	30	1.00	$Ack_3\ C_1$	173	1.00
$Ack_2\ C_2$	50	1.00	$Ack_3\ C_2$	833	1.00
$Ack_2\ C_3$	76	1.00	$Ack_3\ C_3$	3685	1.00
$Ack_2\ C_4$	108	1.00	$Ack_3\ C_4$	15,529	1.00
$Ack_2\ C_5$	146	1.00	$Ack_3\ C_5$	63,789	1.00
$Ack_2\ C_6$	190	1.00	$Ack_3\ C_6$	258,609	1.00
$Ack_2\ C_7$	240	1.00	$Ack_3\ C_7$	1,041,461	1.00

Figure 3.9: *P* and *Expand None* applied to Ackermann's function

From the results of *Expand All* and *Expand None*, one can see that there is potential to make some tradeoffs between efficiency at PE time and at runtime. This can be shown more clearly by using the *Expand N* strategy. Figure 3.10 shows the costs of generating *Ack*$_2$ and *Ack*$_3$ for various values of n, and how well these specialized functions do when applied to C_0 and C_1. These two values were chosen because of the large speedups encountered for them in the previous timings.

You will notice in figure 3.10 that increases in n do not correlate completely to increases in the number of β-reductions needed to generate the specialized Ackermann functions. This has to do with the structure of the terms being evaluated; the cost of residualization may be higher if one more step of computation is taken than if it is not taken. Typically, though, a larger value for n

Expression	Name	β	C_0 β	C_0 Speedup	C_1 β	C_1 Speedup
$P\ Ack\ C_2\ \Sigma_0$	Ack_{2,Σ_0}	449	16	1.00	30	1.00
$P\ Ack\ C_2\ \Sigma_1$	Ack_{2,Σ_1}	424	15	1.07	29	1.03
$P\ Ack\ C_2\ \Sigma_2$	Ack_{2,Σ_2}	418	15	1.07	29	1.03
$P\ Ack\ C_2\ \Sigma_3$	Ack_{2,Σ_3}	412	15	1.07	29	1.03
$P\ Ack\ C_2\ \Sigma_4$	Ack_{2,Σ_4}	478	13	1.23	27	1.11
$P\ Ack\ C_2\ \Sigma_5$	Ack_{2,Σ_5}	678	11	1.45	26	1.15
$P\ Ack\ C_3\ \Sigma_0$	Ack_{3,Σ_0}	476	33	1.00	173	1.00
$P\ Ack\ C_3\ \Sigma_1$	Ack_{3,Σ_1}	451	32	1.03	172	1.01
$P\ Ack\ C_3\ \Sigma_3$	Ack_{3,Σ_3}	445	32	1.03	172	1.01
$P\ Ack\ C_3\ \Sigma_3$	Ack_{3,Σ_3}	439	32	1.03	172	1.01
$P\ Ack\ C_3\ \Sigma_4$	Ack_{3,Σ_4}	613	30	1.10	170	1.02
$P\ Ack\ C_3\ \Sigma_5$	Ack_{3,Σ_5}	944	28	1.18	170	1.02
$P\ Ack\ C_3\ \Sigma_6$	Ack_{3,Σ_6}	940	28	1.18	170	1.02
$P\ Ack\ C_3\ \Sigma_7$	Ack_{3,Σ_7}	1140	27	1.22	169	1.02
$P\ Ack\ C_3\ \Sigma_8$	Ack_{3,Σ_8}	1136	27	1.22	169	1.02
$P\ Ack\ C_3\ \Sigma_9$	Ack_{3,Σ_9}	1205	25	1.32	167	1.04
$P\ Ack\ C_3\ \Sigma_{10}$	$Ack_{3,\Sigma_{10}}$	1186	24	1.375	166	1.04
$P\ Ack\ C_3\ \Sigma_{15}$	$Ack_{3,\Sigma_{15}}$	1156	24	1.375	166	1.04
$P\ Ack\ C_3\ \Sigma_{20}$	$Ack_{3,\Sigma_{20}}$	1429	23	1.43	165	1.05
$P\ Ack\ C_3\ \Sigma_{35}$	$Ack_{3,\Sigma_{35}}$	1409	22	1.5	157	1.10
$P\ Ack\ C_3\ \Sigma_{30}$	$Ack_{3,\Sigma_{30}}$	1359	22	1.5	157	1.10

Figure 3.10: Ack_2 and Ack_3 for various values of *Expand N*

corresponds to a larger value for β. Once n is high enough, P will behave like M or like P with *Expand All*.

The other version of *Expand N*, called *Expand Breadth-N* here, yields slightly different results. In general, *Expand Breadth-N* reduces further than the corresponding *Expand N*, since the strategy is copied to multiple places. Figure 3.12 shows these results. Sometimes strategies can do some surprising things. When we apply *Expand Breadth-N* to a recursive version (given in figure 3.11) of the exponential function[4] the computation explodes for $n > 4$. This happens because computations that normally would be passed *Expand None* ignore that strategy and unfold themselves instead. An example that shows this is given in figure 3.17.

$$\begin{array}{lll}
\mathit{plus} & = & \lambda m\ n.\lambda f\ x.(m\ f\ (n\ f\ x)) \\
\mathit{times} & = & \lambda m\ n.\lambda f\ x.(m\ (n\ f)\ x) \\
\mathit{pow} & = & \lambda m\ n.(m\ n) \\
 & & \\
 & & \textit{We need these next three to make subtraction work} \\
\mathit{cToOpt} & = & \lambda c.(c\ \lambda x\ s\ n.(s\ x)\ \lambda s\ n.n) \\
\mathit{chopOpt} & = & \lambda o.(o\ \lambda t.t\ \lambda s\ n.n) \\
\mathit{optToC} & = & (Y\ \lambda otc\ opt\ f\ x.(opt\ \lambda t.(f\ (otc\ t\ f\ x))\ x)) \\
 & & \\
\mathit{dec} & = & \lambda m.(\mathit{optToC}\ (\mathit{chopOpt}\ (\mathit{cToOpt}\ m))) \\
\mathit{sub} & = & \lambda m\ n.(\mathit{optToC}\ (n\ \mathit{chopOpt}\ (\mathit{cToOpt}\ m))) \\
\mathit{exp} & = & (Y\ \lambda e\ n\ x.(\mathit{isZero}\ n\ c1\ (\mathit{times}\ x\ (e\ (\mathit{dec}\ n)\ x))))
\end{array}$$

Figure 3.11: Recursive versions of arithmetical functions

3.6 Second Futamura Projection

The addition of strategies to the partial evaluator also changes the way self-application works. We can consider a strategy to be a third parameter to the partial evaluator, resulting in a modification to the Futamura projections, show in figure 3.13

Note that in the second and third projections we have the ability to use a different strategy at each stage. We could use a more aggressive strategy for the first evaluator, where we are willing to spend more time, and a faster strategy for the second evaluator. Or, if we find a strategy especially

[4]Using Church numerals n^m can be represented simply as $(m\ n)$. This makes a rather uninteresting example for our partial evaluator, so we use a version that ignores this property of Church numerals and instead makes explicit use of the Y combinator.

Expression	Name	β	$C_0\ \beta$	C_0 Speedup	$C_1\ \beta$	C_1 Speedup
$P\ Ack\ C_2\ \Sigma_{b0}$	$Ack_{2,\Sigma_{b0}}$	449	16	1.00	30	1.00
$P\ Ack\ C_2\ \Sigma_{b1}$	$Ack_{2,\Sigma_{b1}}$	425	15	1.07	29	1.03
$P\ Ack\ C_2\ \Sigma_{b2}$	$Ack_{2,\Sigma_{b2}}$	416	15	1.07	29	1.03
$P\ Ack\ C_2\ \Sigma_{b3}$	$Ack_{2,\Sigma_{b3}}$	407	15	1.07	29	1.03
$P\ Ack\ C_2\ \Sigma_{b4}$	$Ack_{2,\Sigma_{b4}}$	728	10	1.60	24	1.25
$P\ Ack\ C_2\ \Sigma_{b5}$	$Ack_{2,\Sigma_{b5}}$	678	8	2.00	21	1.42
$P\ Ack\ C_3\ \Sigma_{b0}$	$Ack_{3,\Sigma_{b0}}$	476	33	1.00	173	1.00
$P\ Ack\ C_3\ \Sigma_{b1}$	$Ack_{3,\Sigma_{b1}}$	452	32	1.03	172	1.01
$P\ Ack\ C_3\ \Sigma_{b3}$	$Ack_{3,\Sigma_{b3}}$	443	32	1.03	172	1.01
$P\ Ack\ C_3\ \Sigma_{b3}$	$Ack_{3,\Sigma_{b3}}$	434	32	1.03	172	1.01
$P\ Ack\ C_3\ \Sigma_{b4}$	$Ack_{3,\Sigma_{b4}}$	1624	26	1.27	166	1.04
$P\ Ack\ C_3\ \Sigma_{b5}$	$Ack_{3,\Sigma_{b5}}$	1430	22	1.50	155	1.17
$P\ Ack\ C_3\ \Sigma_{b6}$	$Ack_{3,\Sigma_{b6}}$	1369	22	1.50	155	1.17
$P\ Ack\ C_3\ \Sigma_{b7}$	$Ack_{3,\Sigma_{b7}}$	1338	22	1.50	155	1.17
$P\ Ack\ C_3\ \Sigma_{b8}$	$Ack_{3,\Sigma_{b8}}$	1338	22	1.50	155	1.17

Figure 3.12: Ack_2 and Ack_3 for various values of *Expand Breadth-N*

$$\begin{array}{ll}
1 & P(M, s, \sigma) \Rightarrow M_s, \text{where } I(M_s, d) \Rightarrow x \\
2 & P'(P, M, \sigma') \Rightarrow P_M, \text{where } P_M(s, \sigma) \Rightarrow M_s \\
3 & P''(P', P, \sigma'') \Rightarrow P'_P, \text{where } P'_P(M, \sigma') \Rightarrow P_M
\end{array}$$

Figure 3.13: Modified Futamura Projections

Expression	β	Speedup
$P\ P\ Ack\ \Sigma_{All}\ \Rightarrow\ P_{Ack}$	46,365	n/a
$P\ P\ Exp\ \Sigma_{All}\ \Rightarrow\ P_{Exp}$	576,805	n/a
$P_{Ack}\ C_2\ \Sigma_{All}\ \Rightarrow\ Ack_2$	510	1.39
$P_{Ack}\ C_3\ \Sigma_{All}\ \Rightarrow\ Ack_3$	1008	1.38
$P_{Exp}\ C_2\ \Sigma_{All}\ \Rightarrow\ Exp_2$	986	1.43
$P_{Exp}\ C_4\ \Sigma_{All}\ \Rightarrow\ Exp_4$	3117	1.42

Figure 3.14: Second projections of P with Ackermann and exponential

well suited to self-applications, we can use it for the self-applicative stages and then use some other strategy when we want to specialize a different program.

Figure 3.14 shows some timings for the second projection. The expressions Ack_2, Ack_3, Exp_2 and Exp_4 are identical to the ones generated in the first projection. The second projection for Exp was especially expensive. Some of this is due to the expense of computing with Church numerals—especially decrement and equality—and some of this is due to standard combinatorial explosion issues.

The next set of figures shows how using different strategies allows us to trade efficiency in one stage to another. Figures 3.15 shows a second projection of the recursive exponential function using various values of n for *Expand N*. Figure 3.16 shows the resulting generator being used to specialize the exponential function with respect to the Church numerals 2 and 4, using *Expand All*. Notice that as n increases, the amount of time to generate the second projection increases, but the time needed to run the result decreases. Figures 3.17 and 3.18 shows similar experiments using the *Expand Below-N* strategy instead of *Expand N*. The output of the second projection is similar to the output shown in figure 3.15, but the time needed to generate the second projection grows much faster as n increases.

Expression	β	Size
$P\ P\ Exp\ \Sigma_1 \Rightarrow P_{Exp,1}$	5422	929
$P\ P\ Exp\ \Sigma_4 \Rightarrow P_{Exp,4}$	5389	922
$P\ P\ Exp\ \Sigma_8 \Rightarrow P_{Exp,8}$	6792	1180
$P\ P\ Exp\ \Sigma_{16} \Rightarrow P_{Exp,16}$	58,656	11,139
$P\ P\ Exp\ \Sigma_{64} \Rightarrow P_{Exp,64}$	406,924	77,405
$P\ P\ Exp\ \Sigma_{128} \Rightarrow P_{Exp,128}$	626,578	115,712
$P\ P\ Exp\ \Sigma_{256} \Rightarrow P_{Exp,256}$	576,641	110,025
$P\ P\ Exp\ \Sigma_{512} \Rightarrow P_{Exp,512}$	576,641	110,025
$P\ P\ Exp\ \Sigma_{All} \Rightarrow P_{Exp,All}$	576,805	110,025

Figure 3.15: Second projections with *Expand N*

Examining a strategy, one can distinguish between two kinds of expressions. Some of the code of the strategy is responsible for deciding whether or not to perform the β-reduction, and for propagation. The rest of the code performs applications or residualizations. It is significant that the decision-making code does not end up in the residualized expression; the decision-making code can be expensive or even non-terminating (all of the strategies presented here make use of the Y

Expression	β	Expression	β
$P_{Exp,1}\ C_2\ \Sigma_{All}$	1408	$P_{Exp,1}\ C_4\ \Sigma_{All}$	4477
$P_{Exp,4}\ C_2\ \Sigma_{All}$	1407	$P_{Exp,4}\ C_4\ \Sigma_{All}$	4476
$P_{Exp,8}\ C_2\ \Sigma_{All}$	1402	$P_{Exp,8}\ C_4\ \Sigma_{All}$	4471
$P_{Exp,16}\ C_2\ \Sigma_{All}$	1401	$P_{Exp,16}\ C_4\ \Sigma_{All}$	4470
$P_{Exp,64}\ C_2\ \Sigma_{All}$	1260	$P_{Exp,64}\ C_4\ \Sigma_{All}$	4213
$P_{Exp,128}\ C_2\ \Sigma_{All}$	986	$P_{Exp,128}\ C_4\ \Sigma_{All}$	3117
$P_{Exp,256}\ C_2\ \Sigma_{All}$	986	$P_{Exp,256}\ C_4\ \Sigma_{All}$	3117
$P_{Exp,512}\ C_2\ \Sigma_{All}$	986	$P_{Exp,512}\ C_4\ \Sigma_{All}$	3117
$P_{Exp,All}\ C_2\ \Sigma_{All}$	986	$P_{Exp,All}\ C_4\ \Sigma_{All}$	3117

Figure 3.16: Performance of exponential function generator with *Expand N*

Expression	β	Size
$P\ P\ Exp\ \Sigma_{b1}\ \Rightarrow\ P_{Exp,b1}$	5409	929
$P\ P\ Exp\ \Sigma_{b4}\ \Rightarrow\ P_{Exp,b4}$	5355	922
$P\ P\ Exp\ \Sigma_{b6}\ \Rightarrow\ P_{Exp,b6}$	6728	1180
$P\ P\ Exp\ \Sigma_{b8}\ \Rightarrow\ P_{Exp,b8}$	393,397	76,376
$P\ P\ Exp\ \Sigma_{b10}\ \Rightarrow\ P_{Exp,b10}$	401,071	78,112

Figure 3.17: Second projections with *Expand Breadth-N*

Expression	β	Expression	β
$P_{Exp,b1}\ C_2\ \Sigma_{All}$	1408	$P_{Exp,b1}\ C_4\ \Sigma_{All}$	4477
$P_{Exp,b4}\ C_2\ \Sigma_{All}$	1407	$P_{Exp,b4}\ C_4\ \Sigma_{All}$	4476
$P_{Exp,b6}\ C_2\ \Sigma_{All}$	1402	$P_{Exp,b6}\ C_4\ \Sigma_{All}$	4471
$P_{Exp,b8}\ C_2\ \Sigma_{All}$	1399	$P_{Exp,b8}\ C_4\ \Sigma_{All}$	4454
$P_{Exp,b10}\ C_2\ \Sigma_{All}$	1351	$P_{Exp,b10}\ C_4\ \Sigma_{All}$	4376

Figure 3.18: Performance of exponential function generator with *Expand Breadth-N*

combinator, and none of them have a normal form) without needing to be concerned about the effect they will have on the residualized code.

3.7 Third Futamura Projection and Code Explosion

The third projection is much more difficult to obtain, especially with the addition of strategies. If we take the third projection of M, it takes 232,631 β reductions. On the system used for experimentation, this took 38 seconds. If we take the third projection of P with *Expand All*, the computation crashes the computer after many hours of computations. Much of our work on the reducer in chapter 6 was motivated by the question of why this happens, and wanting to be sure that lack of sharing in the reducer itself was not responsible for the failed result.

3.7.1 Combinatorial Explosion

A partial evaluator is an expanded interpreter that makes use of symbolic interpretation when the values of expressions are not know. The structure of a partial evaluator can be written in the form

$$\texttt{if}\ \mathit{known}(\mathit{exp})\ \texttt{then}\ \mathit{eval}(\mathit{exp})\ \texttt{else}\ \mathit{residualize}(\mathit{exp}).$$

If the expression is known, then evaluate it as it would be in a standard interpreter; otherwise residualize it. Note that the evaluator makes use of both information (the value of an expression) and meta-information (whether or not it knows the value of an expression) to perform its task. The information may be known or not known, but the meta-information is always available.

This changes during the second and third projections, however. A partial evaluator suffers from a lack of information during the partial evaluator process, which causes combinatorial code explosion. To understand why, consider the three partial evaluators given in the third Futamura projection, $P_A(P_M, P_P) \Rightarrow P_{gen}$. The first one, P_A, is in the "active" position. The P_A evaluator is told both the input program and that input program's static input. It will be able to examine an expression, determine whether or not it knows enough to perform a computation statically, and act accordingly.

The second partial evaluator P_M is in the "middle" position. Since a partial evaluator is a kind of interpreter, we can say that P_A is running P_M. Evaluator P_M will have P_P (the "passive" evaluator) as its first argument, but nothing yet specified for its second argument. Consider the effect this has on the `if` statement above; it is already true that P_M might know enough about *exp* to evaluate it, or it might know so little about *exp* that it needs to residualize it. But, since the second argument to P_M has not yet been given, it may be that P_M cannot even tell if it knows *exp* or not. Not only is the information lacking, but the meta-information is lacking also. As a result, P_A will decide that P_M will have to wait until runtime before it can discover the value of *known*(*exp*), and therefore will residualize the call to *known*(*exp*), and then attempt to specialize *both* branches of the `if` statement. Note that *eval* and *residualize* are known, and will attempt to break apart *exp* into sub-expressions. In fact, since *eval* and *residualize* will contain calls to the partial evaluator itself, the effect is that P_M is partially specialized with respect to many of the sub-expressions of P_P. This is the essence of combinatorial code explosion.

This leads to one obvious question: why does the Third Projection work for M and not for P? The main reason seems to be the size of the evaluators. Mogensen's M is only 131 nodes, while P is 151 nodes (not including a strategy). The memory available on today's computers is sufficient to contain the computation needed for the third projection of M, but not for P.

The Futamura projections are written with a single partial evaluator in mind, but in fact the equations still work even if the three partial evaluators are different. It is instructive to take the 8 permutations of the Third Projection of both M and P. The results are in figure 3.19. Changing the active partial evaluator from M to P results in roughly a %50 slowdown, with similar results from changing the passive evaluator from M to P. But changing the middle evaluator—the one subject to combinatorial explosion—causes a slowdown of two orders of magnitude.

Exp				Beta	CPU
M	M	M		232,621	38
M	M	P		350,358	61.48
M	P	M		26,630,792	3321 (55 min)
M	P	P		crashed	crashed
P	M	M	*EAll*	361,680	245
P	M	P	*EAll*	546,211	428
P	P	M	*EAll*	35,217,577	9762 (2h 42m)
P	P	P	*EAll*	crashed	crashed

Figure 3.19: Third Futamura Projections with M and P

3.7.2 Third Projection and Strategies

The problem with the lack of information is that the partial evaluator continues to try making reductions even though it doesn't know enough information to terminate in a reasonable period of time. This is why binding-time analysis helps: by limiting the evaluator to reducing expressions it knows about *for sure*, we make termination much more likely.

This is one area where strategies can show their benefit. In figure 3.20 there are a few runs of the third projection of Ackermann's function with various values of n for *Expand N*. As n becomes larger, more time is needed to run the third projection, but the corresponding P_{gen} runs faster.

Expression	Size	β
$P\ P\ P\ \Sigma_{10}\ \Rightarrow\ P_{gen_1 0}$	644	9184
$P\ P\ P\ \Sigma_{50}\ \Rightarrow\ P_{gen_5 0}$	47,672	643,247
$P\ P\ P\ \Sigma_{70}\ \Rightarrow\ P_{gen_7 0}$	170,154	2,297,039
$P_{gen_1 0}\ Ack\ \Sigma_{All}$	3842	52,022
$P_{gen_5 0}\ Ack\ \Sigma_{All}$	3842	51,992
$P_{gen_7 0}\ Ack\ \Sigma_{All}$	3842	51,520

Figure 3.20: Third projection

Chapter 4

Expanding the Strategies

Strategies have several properties that offer benefits when added to a partial evaluator. First, strategies add a decision mechanism to the partial evaluator. Instead of simply expanding at every opportunity, strategies can make more nuanced decisions about the reductions they examine. Second, because strategies themselves are kept outside of the partial evaluator, we can consider them "off-budget" during self-application. In particular, strategies can contain code (such as the Y-combinator) that has no normal form, but without causing non-termination. Though the additional code needed to propagate strategies through the partial evaluator disables the third projection, we are still able to use the second projection and select different tradeoffs. Third, strategies are composable, making the partial evaluator more modular and flexible.

In the previous chapter, we were able to demonstrate these benefits, but the ability of strategies to do interesting things has been sharply limited by the fact that the language we are using has only three kinds of terms, and only one kind of reduction. This is a limitation because it gives the strategy very little information to use to make its decision. Even if we are able to analyze the terms easily, just knowing that the current node is, for example, an application node does not give us much information. To address this limitation we will investigate several methods of expanding the ability of strategies to make decisions. These methods include adding combinators to the HOAS for representing variables, using a hybrid of HOAS and concrete syntax, adding types to the language, and changing the process that strategies use from simple selection to a more general form of term manipulation.

4.1 Content-Observing Strategies

The strategies presented in the previous chapter are only minimally aware of the terms they are reducing. Though the structure of the target term controls the propagation of the strategy, the decision of whether to unfold or residualize an application is made according to the internal structure of the strategy itself, rather than the content of the terms. In order to be more useful, strategies need to have more knowledge about the terms they are reducing.

Mogensen hints at this in his discussion of the combinatorial explosion observed at the third projection of M, explaining that the "non-critical reduction of non-linear redexes" was the problem. This is one symptom of lack of binding-time information, but it does raise the question: what would happen if the strategy were able to detect non-linear redexes? Other obvious strategies like "expand if the argument is small", or "only expand calls to function f" are also impossible to write.

Answering these questions with the current representation is not possible partly because of the assumption that all variables are bound. For example, consider the *size* operation:

$$\begin{aligned} \mathit{size}\ e\ &= (e \quad \lambda ab.(\mathit{inc}\ (\mathit{plus}\ a\ b)) \\ &\qquad\quad \lambda g.(\mathit{inc}\ (g\ C_1))) \end{aligned}$$

Variables are bound to the value 1, and at other node types *size* simply takes the size of the children and increments. The following example shows what happens when taking the size of the term $\lambda x.(x\ x)$.

$(\mathit{size}\ [\![\lambda x.(x\ x)]\!])\ \rightarrow$

$(\mathit{size}\ \lambda ab.(b\ \lambda x.(a\ x\ x)))\ \rightarrow$

$(\mathit{inc}\ (\lambda x.(\mathit{inc}\ (\mathit{plus}\ x\ x))\ C_1))\ \rightarrow$

$(\mathit{inc}\ (\mathit{inc}\ (\mathit{plus}\ C_1\ C_1)))\ \rightarrow$

$(\mathit{inc}\ (\mathit{inc}\ C_2))\ \rightarrow$

$(\mathit{inc}\ C_3)\ \rightarrow$

C_4

This won't work in our case, because during partial evaluation we will reduce inside function bodies, underneath a λ, and therefore need to deal with free variables. As an example, suppose we are processing the term $\lambda y.\lambda x.(x\ y)$, and need to take the size of the body of the λy abstraction.

The computation would look like this:

$size\ [\![\lambda x.(x\ y)]\!]\ \rightarrow$

$size\ \lambda ab.(b\ \lambda x.(a\ x\ y))\ \rightarrow$

$(inc(\lambda x.(inc\ (plus\ x\ y))\ C_1))\ \rightarrow$

$(inc(inc\ (plus\ C_1\ y)))$

At this point no further computation can be done, and the size function will fail to return a useful result.

4.1.1 Adding a Third Combinator

One way to allow for more interesting strategies is to expand the representation to handle variables directly. The new representation will take three inputs instead of two. This new representation is shown in figure 4.1.

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

Figure 4.1: HOAS representation allowing variables

With this representation, we can know explicitly if a node is a variable, even if it is unbound. This allows us to write a new size function:

$$\begin{array}{lll} size\ e & = (e & \lambda ab.(inc\ (plus\ a\ b)) \\ & & \lambda g.(inc\ (g\ C_1)) \\ & & \lambda x.C_1) \end{array}$$

Now if we try to take the size of $\lambda x.(x\ y)$, we are successful.

$size\ [\![\lambda x.(x\ y)]\!]\ \rightarrow$

$size\ \lambda abc.(b\ \lambda x.(a\ (c\ x)\ (c\ y)))\ \rightarrow$

$(inc(\lambda x.(inc\ (plus\ C_1\ C_1))\ C_1))\ \rightarrow$

$(inc(inc\ (plus\ C_1\ C_1))) \rightarrow \cdots$

C_4

The change in representation will necessitate a corresponding change in the partial evaluator,

$$
\begin{array}{lcl}
P & = & \lambda mns.(R\ \lambda abc.(a\ (m\ a\ b\ c)\ (n\ a\ b\ c))\ s\ F) \\
R & = & \lambda m.(m\ A\ B\ C) \\
C & = & \lambda x.x \\
B & = & \lambda gs.\lambda x.(x \\
 & & \qquad\qquad \lambda v.(g\ v\ s) \\
 & & \qquad\qquad \lambda abc.(b\ \lambda z.(g\ (D\ \lambda abc.(c\ z))\ s\ F\ a\ b\ c))) \\
A & = & \lambda mns.(s\ m\ n) \\
D & = & (Q\ Q) \\
Q & = & \lambda qvs.\lambda x.(x \\
 & & \qquad\qquad \lambda q'w.(q'\ q'\ \lambda abc.(a\ (v\ a\ b\ c)\ (w\ s\ F\ a\ b\ c))\ s) \\
 & & \qquad\qquad \lambda q'.v \\
 & & \qquad q) \\
T & = & \lambda ab.a \\
F & = & \lambda ab.b
\end{array}
$$

Figure 4.2: Strategy-based partial evaluator, with represented variables

which is given in figure 4.2. A new combinator C is added which simply returns its argument; the partial evaluator itself is no more powerful than the version that does not represent variables.

Expand Small

One strategy we can write with this new representation is *Expand Small*, shown in figure 4.3. It takes a number n to quantify what is meant by "small" and two PEVs π and π'. After applying itself to these two PEVs it checks to see if the argument π' is smaller than n. If so, it reduces π, otherwise, it residualizes π. We will represent this strategy by $\Sigma_{small,n}$, where n is the parameter quantifying smallness.

Here is an example to illustrate the operation of *Expand Small*. First we use $\Sigma_{small,1}$:

$$
\begin{array}{rl}
P\ [\![\lambda x.(\lambda y.(y\ y)\ \lambda z.z)]\!]\ [\![\lambda q.(q\ q)]\!]\ \Sigma_{small,1} & \Rightarrow \\
\ll (\lambda x.(\lambda y.(y\ y)\ \lambda z.z)\ \lambda q.(q\ q)) \gg\ \Sigma_{small,1} & \Rightarrow \\
(\Sigma_{small,1}\ \ll \lambda x.(\lambda y.(y\ y)\ \lambda z.z) \gg\ \ll \lambda q.(q\ q) \gg) & \Rightarrow \\
(\Sigma_{small,1}\ \ll \lambda x.(\lambda y.(y\ y)\ \lambda z.z) \gg\ (\pi_2\ \ll \lambda q.(q\ q) \gg\ \Sigma_{small,1})) & \Rightarrow \\
(\Sigma_{small,1}\ \ll \lambda x.(\lambda y.(y\ y)\ \lambda z.z) \gg\ Abs(\lambda q.App(Var(q), Var(q)))) & \Rightarrow
\end{array}
$$

$$\begin{aligned}
\mathit{App}((\ll \lambda x.(\lambda y.(y\ y)\ \lambda z.z) \gg\ \Sigma_{small,1}), \mathit{Abs}(\lambda q.\mathit{App}(Var(q), Var(q)))) &\Rightarrow \\
\mathit{App}(\mathit{Abs}(\lambda x.(\ll (\lambda y.(y\ y)\ \lambda z.z) \gg\ \Sigma_{small,1})), \mathit{Abs}(\lambda q.\mathit{App}(Var(q), Var(q)))) &\Rightarrow \\
\mathit{App}(\mathit{Abs}(\lambda x.(\Sigma_{small,1}\ \ll \lambda y.(y\ y) \gg \ll \lambda z.z \gg)), \mathit{Abs}(\lambda q.\mathit{App}(Var(q), Var(q)))) &\Rightarrow \\
\mathit{App}(\mathit{Abs}(\lambda x.(\Sigma_{small,1}\ \ll \lambda y.(y\ y) \gg\ (\ll \lambda z.z \gg\ \Sigma_{small,1}))), & \\
\mathit{Abs}(\lambda q.\mathit{App}(Var(q), Var(q)))) &\Rightarrow \\
\mathit{App}(\mathit{Abs}(\lambda x.(\Sigma_{small,1}\ \ll \lambda y.(y\ y) \gg\ \mathit{Abs}(\lambda z.Var(z)))), \mathit{Abs}(\lambda q.\mathit{App}(Var(q), Var(q)))) &\Rightarrow \cdots \\
\mathit{App}(\mathit{Abs}(\lambda x.\mathit{App}(\mathit{Abs}(\lambda y.\mathit{App}(Var(y), Var(y))), \mathit{Abs}(\lambda z.Var(z)))), & \\
\mathit{Abs}(\lambda q.\mathit{App}(Var(q), Var(q)))) &\Rightarrow \\
\equiv [\![(\lambda x(\lambda y.(y\ y)\ \lambda z.z)\ \lambda q.(q\ q))]\!] &
\end{aligned}$$

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

Next we use $\Sigma_{small,2}$:

$$\begin{aligned}
P\ [\![\lambda x.(\lambda y.(y\ y)\ \lambda z.z)]\!]\ [\![\lambda q.(q\ q)]\!]\ \Sigma_{small,2} &\Rightarrow \\
\ll (\lambda x.(\lambda y.(y\ y)\ \lambda z.z)\ \lambda q.(q\ q)) \gg\ \Sigma_{small,2} &\Rightarrow \\
(\Sigma_{small,2}\ \ll \lambda x.(\lambda y.(y\ y)\ \lambda z.z) \gg \ll \lambda q.(q\ q) \gg) &\Rightarrow \\
\mathit{App}(\mathit{Abs}(\lambda x.(\ll (\lambda y.(y\ y)\ \lambda z.z) \gg\ \Sigma_{small,2})), \mathit{Abs}(\lambda q.\mathit{App}(Var(q), Var(q)))) &\Rightarrow \\
\mathit{App}(\mathit{Abs}(\lambda x.(\Sigma_{small,2}\ \ll \lambda y.(y\ y) \gg \ll \lambda z.z \gg)), \mathit{Abs}(\lambda q.\mathit{App}(Var(q), Var(q)))) &\Rightarrow \\
\mathit{App}(\mathit{Abs}(\lambda x.(\ll (\lambda z.z\ \lambda z.z) \gg\ \Sigma_{small,2})), \mathit{Abs}(\lambda q.\mathit{App}(Var(q), Var(q)))) &\Rightarrow \\
\mathit{App}(\mathit{Abs}(\lambda x.(\Sigma_{small,2}\ \ll \lambda z.z \gg \ll \lambda z.z \gg)), \mathit{Abs}(\lambda q.\mathit{App}(Var(q), Var(q)))) &\Rightarrow \\
\mathit{App}(\mathit{Abs}(\lambda x, \mathit{Abs}(\lambda z.Var(z))), \mathit{Abs}(\lambda q.\mathit{App}(Var(q), Var(q)))) & \\
\equiv [\![(\lambda x.\lambda z.z\ \lambda q.(q\ q))]\!] &
\end{aligned}$$

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.

Expand N Then

Often partial evaluation produces unfortunate results because the source code is ordered inconveniently. For example, consider the term

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

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. This is a limitation of curried form in partial evaluation. We can unfold (specialize) an uncurried function like $\lambda(x, y).M$ if it is applied to two arguments when the second is known but the first is unknown. This is not true of the curried form $\lambda x.\lambda y.M$, unless we do something to recognize this situation and "lift" the λy outside of the function.

One way to solve this using strategies is to have a strategy that behaves like *Expand All* for one level, and then like *Expand Small* afterwards. We can accomplish this by composing 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 *Expand N Then*, and is a generalization of *Expand N*.

$$\Sigma_{n,then}\ =\ \lambda n.\lambda\sigma.(n\ \lambda e.\lambda\pi.(\pi\ e)_2\ \sigma)$$

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:

$$\begin{aligned}P\quad & [\![(\lambda abc.\lambda d.(d\ a\ b\ b)\ \lambda x.(x\ x)\ \lambda q.q)]\!]\ [\![\lambda y.(y\ y)]\!](\Sigma_{n,then}\ C_2\ (\Sigma_{small}\ C_3))\\ & = [\![(\lambda cd.(c\ \lambda x.(x\ x)\ \lambda q.q)\ \lambda y.(y\ y))]\!]\end{aligned}$$

The first two applications were reduced, while the third was residualized since it was too large.

The λ-calculus renderings of Σ_{small} and $\Sigma_{n,then}$ are in figure 4.3.

$$\begin{array}{rcl}
\Sigma_{n,then} & = & \lambda n.\lambda s.(n\ \lambda e.\lambda t.(fst\ (t\ e))\ s) \\
\Sigma_{small} & = & \lambda n.(Y\ \lambda e_{small}. \\
 & & \quad \lambda \pi_1 \pi_2. \\
 & & \quad\quad (\lambda r_1 r_2.(lt\ (size\ (snd\ r_2))\ n) \\
 & & \quad\quad\quad (fst\ r_1\ \pi_2) \\
 & & \quad\quad\quad (fst\ (D\ (snd\ r_1)\ e_{small})\ \pi_2) \\
 & & \quad\quad (\pi_1\ e_{small}) \\
 & & \quad\quad (\pi_2\ e_{small})))
\end{array}$$

Figure 4.3: *Expand Small* and *Expand N Then*

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 need 548 β-reductions.

4.2 First Order Abstract Syntax

Another approach to handling the problem of being unable to observe our terms is to move to a concrete representation. To create a concrete representation we added two new operators to our language: quote and comma. These operators are staging operators and are similar to systems such as Scheme's backquote and comma syntax[1]. Normal terms are considered to be at stage 0 and will be evaluated.

The quote adds one to the staging level, and indicates that the term should be evaluated at the next stage. For example, consider the term $(\lambda a.'(\lambda b.b\ \lambda c.c)\ \lambda d.d)$. The result of the execution will be the term $'(\lambda b.b\ \lambda c.c)$. For the next phase of execution, we remove the $'$ to get $(\lambda b.b\ \lambda c.c)$, which can be reduced. A quoted expression can have quotes within it; a quoted term within a state n term will be at stage $n+1$. This process can continue indefinitely.

The comma operator, also called *anti-quote*, subtracts one from the staging level. It is used to cause computation to occur at the current stage inside of an expression of a later stage. For example, in $(\lambda a.'(\lambda x.x\lambda\ ,a)\ '\lambda y.y)$ the $\lambda y.y$ term will be placed inside of a stage 1 expression by a stage 0 application. The result will be $'(\lambda x.x\ \lambda y.y)$.

Modifying the partial evaulator to use these operators yields the program in figure 4.4. The most visible difference is that terms that would be output in the form $\lambda a\ b.\overline{M}$ now have the form $'M$. The concretization of the terms makes it easy to write language primitives in the reducer to examine them, and simplifies their display. It also simplifies the reducer, allowing us to use a weak head normal form reduction rather than full normalization. This is because the staging operators eliminate the need to reduce the body of an abstraction to perform the computations.

Another benefit is that this format eliminates a lot of "administrative β-reductions" associated with HOAS. This can be seen by examining the code for B, where $\lambda ab.(b\ \lambda w.(g\ (D\ \lambda ab.w)\ F\ a\ b))$ is replaced by $'\lambda w.,(g\ (D\ 'w)\ F)$.

While this technique is interesting, in practice it does not give us much more than what we started with. While it is true that we could add new primitives to the reducer to allow for examination of the concrete syntax, it is equally true that now we have no choice in the matter—we must add them if we want to examine the terms at all. With HOAS, we already have the means to examine terms. We also lose some ability we had before, in that once an expression is reduced, we are unable to process it any further. In HOAS, we could take an expression and process it with the partial evaluation combinators, or other combinators.

$$\begin{array}{lcl}
P & = & \lambda mns.(R\ \lambda ab.(a\ (m\ a\ b)\ (n\ a\ b))\ s\ F) \\
R & = & \lambda m.(m\ A\ B) \\
B & = & \lambda gs.\lambda x.(x \\
 & & \quad \lambda x.(g\ x\ s) \\
 & & \quad '\lambda z.,(snd\ (g\ (D\ 'z)\ s))) \\
A & = & \lambda mns.(s\ m\ n) \\
D & = & (Q\ Q) \\
Q & = & \lambda qvs.\lambda x.(x \\
 & & \quad \lambda q'.\lambda w.(q'\ q'\ 'v\ (fst\ (w\ s))\ s) \\
 & & \quad \lambda q'.v \\
 & & \quad q)
\end{array}$$

Figure 4.4: Hybrid FOAS strategy-based partial evaluator

4.3 Annotated Terms

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 pre-processing 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 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 [17].

A λ-expression e is represented by

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

The translation of terms to PEV's is as follows:

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

The version of the partial evaluator that handles annotations is in figure 4.5.

Because the representation of expressions has changed, strategies such as *Expand 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 *Expand None*

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

$$
\begin{array}{lcl}
P & = & \lambda mns.(R\ \lambda abc.(a\ (m\ a\ b\ c)\ (n\ a\ b\ c))\ s\ F\ F) \\
R & = & \lambda m.(m\ A\ B\ C) \\
C & = & \lambda x.x \\
B & = & \lambda ngs.\lambda x.(x\ n \\
 & & \qquad\qquad \lambda y.(y \\
 & & \qquad\qquad \lambda v.(g\ v\ s) \\
 & & \qquad\qquad \lambda abc.(b\ T\ \lambda z.(g\ (D\ \lambda abc.(c\ z))\ s\ F\ T\ a\ b\ c)))) \\
A & = & \lambda mns.(s\ m\ n) \\
D & = & (Q\ Q) \\
Q & = & \lambda qvs.\lambda x.(x\ T \\
 & & \qquad\qquad \lambda y.(y \\
 & & \qquad\qquad \lambda q'w.(q'\ q'\ \lambda abc.(a\ (v\ a\ b\ c)\ (w\ s\ F\ F\ a\ b\ c))\ s) \\
 & & \qquad\qquad \lambda q'.v) \\
 & & \qquad q) \\
T & = & \lambda ab.a \\
F & = & \lambda ab.b
\end{array}
$$

Figure 4.5: Strategy-based partial evaluator, with annotations

Here is a strategy, *Expand Marked*, that uses these annotations:

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

Similar to *Expand 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 the annotation field is *true*, then it returns the static part of r; otherwise, it uses *Expand None* and D to residualize.

As an example, consider the expression from the previous section. With *Expand 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.

$$
\begin{aligned}
&P\quad (\underline{\lambda a}.(\underline{\lambda b}.(\lambda c.c\ \lambda r.(r\ r))\ \lambda x.x)\ \lambda q.(q\ q))\ \Sigma_{marked} \\
&\qquad = [\![(\lambda c.c\ \lambda r.(r\ r))]\!]
\end{aligned}
$$

This form of PEV can be used in a number of ways. If a binding time analyzer were employed to annotate terms, our partial evaluator with the *Expand Marked* strategy would mimic an off-line partial evaluator. We could also use the annotation to describe other things, such as indicating which variable is an induction variable. Further, we could specify that the type of annotations could be something other than boolean.

4.3.1 Expand Linear

One strategy that would be nice to have is *Expand Linear*, which causes an expansion to occur if the operator is linear or constant—i.e., the variable it binds occurs one or zero times in the body of the function. This could help reduce the effects of combinatorial explosion by disallowing the kinds of function applications that would generate exponential behavior. Unfortunately, we cannot write in λ-calculus a function which can test for linearity unless we can guarantee that there are no free variables in the term. The reasons are the same as the the discussion of the *size* function in section 4.1.

Another solution is to use *Expand Marked* as a form of *Expand Linear*. We use a preprocessor to mark linear abstractions and then let *Expand Marked* do the work.

One note about all of these content-observing strategies is in order. While these strategies would seem to work well in theory, there is a major limitation that makes them difficult to use in practice. In order for the strategy to examine the structure of a term, it needs to extract the residual portion of its PEV. This is not expensive to do once, but to do this for every sub-term causes the algorithm to require an exponential time complexity—unless the results of these residualizations can be shared somehow.

4.4 Booleans and Integers

The previous sections discussed expanding the representation of the terms to increase the power of strategies. Another approach to this is to expand the language by adding simple types and keywords. Both partial evaluators M and P have the effect of asking "what can we do if we have control over β-reductions?". But not all β reductions mean the same thing. For example, consider the church numeral exponentiation: $(C_3\ C_3)$. This computation requires 16 β-reductions, all of which are really "administrative" in that they are not meant to indicate an actual function, but instead are there to preserve the format and representation of integer arithmetic. The computation

$(C_4\ C_4)$ requires 89 β-reductions.

Furthermore, unlike "real" function application, there is seldom a need to forbid the reduction of such terms. If the opportunity exists to add two integers together, it is almost always desirable to perform the addition. Unfortunately, these representational β-redexes look just like any other, and so the strategies will have to process them as if they were general β-redexes. This could have a strange effect in that, for example, arithmetic could be partially done. We could end up applying $(C_4\ C_4)$ under Σ_n, where n is less than the 89 reductions needed. As a result, the exponentiation will be stopped mid-way. This hardly seems like a meaningful or useful result.

Similarly, given the term $(f\ \lambda ab.a)$, what do we say about f? It could be that f takes any function and $\lambda ab.a$ happened to be the one given; or that f is expecting a projection function; or perhaps even that the input to f is supposed to be a boolean. In contrast, a term like $(f\ \mathit{true})$ leaves no room for doubt, and the internals of f can be reduced accordingly.

4.4.1 Expanding the Representation

We added booleans, integers, the `if` keyword, and arithmetic operators to the language. The new grammar is $\Lambda = i \mid b \mid \texttt{if}\ \Lambda\ \Lambda\ \Lambda \mid x \mid (\Lambda\ \Lambda) \mid \lambda x.\Lambda$. We use prefix notation, because we want parenthesis to be used only for application. The HOAS for the language also needs to be extended.

$$\begin{aligned}
\lceil M \rceil &= \lambda\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}.\ \lfloor M \rfloor \\
\lfloor x \rfloor &= (c\ x) \\
\lfloor M\ N \rfloor &= a\ \lfloor t_1 \rfloor\ \lfloor t_2 \rfloor \\
\lfloor \lambda\ x\ .\ M \rfloor &= b\ (\lambda\ x\ .\ \lfloor M \rfloor) \\
\lfloor i \rfloor &= (c_i\ i) \\
\lfloor b \rfloor &= (c_b\ b) \\
\lfloor \texttt{if}\ M\ N\ P \rfloor &= (k_i\ \lfloor M \rfloor\ \lfloor N \rfloor\ \lfloor P \rfloor) \\
\lfloor \oplus \rfloor &= (k_{ub|ui|bb|bi}\ \oplus)
\end{aligned}$$

The new variables c_b and c_i represent boolean and integer constants, respectively. The variable k_i represents `if` expressions. The variables k_{ub} and k_{ui} represent unary operations returning booleans and integers, and k_{bb} and k_{bi} represent binary operations that return booleans and integers. The fact that operators work on constants makes it simple for the partial evaluator to decide if the arguments are known or unknown, and whether it is able to call the operator or residualize it. We need to know the output type and arity of the operators in order to represent any return values correctly.

First we add expand Mogensen's evaluator to handle the larger language. The combinators for constants are straightforward, and are listed in figure 4.6.

$$\begin{array}{lcl} C_B & = & \lambda b\, p.(p\ error\ \lambda a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}.(c_b\ b)) \\ C_I & = & \lambda i\, p.(p\ error\ \lambda a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}.(c_i\ i)) \end{array}$$

Figure 4.6: Constant handling in M^c

The PEV returned tells what happens in two cases: the first case is what to do if the user tries to apply the value to something. This is a type error, and should "never happen", so the free variable `error` is returned. The second case is what happens if we don't apply the term to anything, but simply reduce it as much as we can right now. In this event we return the representation of the constant.

The `if` keyword is more complex, and is handled by the K_I combinator in figure 4.7. Unlike the constants, it does make sense that the `if` expression could be applied to something.

$$\begin{array}{lcll} K_I & = & \lambda c\, t\, e.\ (\lambda r\ u.\ (c\ F & \lambda a\ b.\ r \\ & & & \lambda a.\ r \\ & & & \lambda a.\ r \\ & & & u \\ & & & \lambda i.\ r \\ & & & \lambda c\, t\ e.\ r \\ & & & \lambda u.\ r \\ & & & \lambda u.\ r \\ & & & \lambda o.\ r \\ & & & \lambda o.\ r) \\ & & \multicolumn{2}{l}{(D\ \lambda a\ b\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}.(k_i\quad (c\ F\ a\ b\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi})} \\ & & \multicolumn{2}{l}{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (t\ F\ a\ b\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi})} \\ & & \multicolumn{2}{l}{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (e\ F\ a\ b\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi})))} \\ & & \multicolumn{2}{l}{\lambda b.\texttt{if}\ b\ t\ e)} \end{array}$$

Figure 4.7: If handling in M

The first ten lines examine the conditional part of the `if` expression and see what kind of term it is after partial evaluation. If it turns into a boolean, then the last line (which will be captured by the u abstraction) captures the value of that boolean and uses it to return the appropriate branch of the `if` expression. The return values t and e both have type PEV. If the condition turns into anything other than a boolean, the remaining lines (captured by the r abstraction) call the D combinator with a representation of the `if` expression, which also returns a PEV. Thus, we always return a PEV, and the actual PEV returned will depend on whether or not the `if` expression can be reduced. In keeping with the aggressive nature of M, the expression is always reduced if possible.

Binary and unary operations are far more complex, for two reasons. First, there are three common types for both the binary operations ($int \to int \to int, int \to int \to bool$, and $bool \to bool \to bool$) and the unary operators ($int \to int$, $int \to bool$, and $bool \to bool$). These types must be maintained for proper functioning of the partial evaluator. Second, we have to treat the operators as functions that can be passed to other functions if we want to be able to represent them with HOAS. For example, $(a\ (a\ (k_{bi}\ +)\ x)\ y)$ would represent $x + y$; the $+$ is not activated here, it is to be passed as an argument to k_{bi}.

The solution, then, is to create four different combinators, given in figure 4.8, for each of the combinations of arity (binary and unary) and output type (boolean and integer). These combinators serve as a wrapper around the operators. This wrapper function turns the operator into a PEV. This PEV needs to handle three cases. First, if we try to residualize this PEV, we should get back the representation of the operator. If we apply something to it, the PEV needs to reduce the argument and examine it. If the argument reduces to a constant—the second case—the PEV feeds the constant to the operator, and represents the result with c_i or c_u, as appropriate. In the third case, the argument reduces to something else, and the PEV residualizes the application of the operator to the argument. The third case also applies if the argument is dynamic, which is why we reintroduce the C combinator to this representation.

The case of binary operators is similar, except that the result of an application is represented by k_{ub} or k_{ui} instead of c_b or c_i. Further, the call to D needs to be moved into the `if` expression, since only one of the branches will return an expression; the k_{ub} and k_{ui} combinators will already return PEVs.

If we can assume the type-correctness of the input program, we do not need to have separate combinators based on the input to the function; we only need to keep track of the output. This reduces the number of combinators we need, and also allows for the comparison operators to work on both integers and booleans. We could simplify things further by using only one representation of constants, but we would no longer be able to distinguish between booleans and integers.

$$
\begin{array}{lll}
K_{UB} & = & \lambda o\, p. \\
& & (p\ \lambda z.(\lambda z_f. \\
& & \quad (D\ if\ (isConst\ z_f) \\
& & \qquad (getConst\ z_f\ \lambda v. \\
& & \qquad\quad \lambda a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}.(c_b\ (o\ v))) \\
& & \qquad \lambda a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}.(a\ (k_{ub}\ o)\ (z_f\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}))) \\
& & \quad (z\ F)) \\
& & \ \lambda abc_bc_ik_ik_{ub}k_{ui}k_{bb}k_{bi}.(k_{ub}\ o)) \\
K_{BB} & = & \lambda o\, p. \\
& & (p\ \lambda z.(\lambda z_f. \\
& & \quad (D\ if\ (isConst\ z_f) \\
& & \qquad (getConst\ z_f\ \lambda v.(K_{UB}\ (o\ v))) \\
& & \qquad \lambda a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}.(a\ (k_{bb}\ o)\ (z_f\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}))) \\
& & \quad (z\ F)) \\
& & \ \lambda a\ b\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}.(k_{bb}\ o)) \\
K_{UI} & = & \lambda o\, p. \\
& & (p\ \lambda z.(\lambda z_f. \\
& & \quad if\ (isConst\ z_f) \\
& & \qquad (getConst\ z_f\ \lambda v. \\
& & \qquad\quad \lambda a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}.(c_i\ (o\ v))) \\
& & \qquad (D\lambda a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}.(a\ (k_{ui}\ o)\ (z_f\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}))) \\
& & \quad (z\ F)) \\
& & \ \lambda abc_bc_ik_ik_{ub}k_{ui}k_{bb}k_{bi}.(k_{ui}\ o)) \\
K_{BI} & = & \lambda o\, p. \\
& & (p\ \lambda z.(\lambda z_f. \\
& & \quad if\ (isConst\ z_f) \\
& & \qquad (getConst\ z_f\ \lambda v.(K_{UI}\ (o\ v))) \\
& & \qquad (D\lambda a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}.(a\ (k_{bi}\ o)\ (z_f\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}))) \\
& & \quad (z\ F)) \\
& & \ \lambda a\ b\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}.(k_{bi}\ o))
\end{array}
$$

Figure 4.8: Binary and unary operator combinators in M_c

The *isConst* and *getConst* functions are defined as utility functions in figure 4.9. The *getConst* function takes a continuation.

The resulting partial evaluator, which we will call M^c, is much larger than the original M, at

$$
\begin{array}{llllll}
isConst = & \lambda i.(i & \lambda a\, b.false & getConst = & \lambda i\, k.(i & \lambda a\, b.(k\ error) \\
& & \lambda b.false & & & \lambda b.(k\ error) \\
& & \lambda b.true & & & \lambda b.(k\ b) \\
& & \lambda i.true & & & \lambda i.(k\ i) \\
& & \lambda c\, t\, e.false & & & \lambda c\, t\, e.(k\ error) \\
& & \lambda u\, x.false & & & \lambda u\, x.(k\ error) \\
& & \lambda u\, x.false & & & \lambda u\, x.(k\ error) \\
& & \lambda o\, x\, y.false & & & \lambda o\, x\, y.(k\ error) \\
& & \lambda o\, x\, y.false) & & & \lambda o\, x\, y.(k\ error))
\end{array}
$$

Figure 4.9: *is* and *get* for constants

2567 nodes. The full code is in figures 4.20 and 4.21.

To determine the effect of these changes, we ran M and M^c against two versions of the recursive version of the exponential function. One used only Church numerals, the other used `if` and integers; the exact code is given in figure 4.10. The *isZero*, *times* and *dec* functions are Church-numeral based.

$$
\begin{array}{lcl}
exp & = & (Y\ \lambda e\ n\ x.(isZero\ n\ c_1\ (times\ x\ (e\ (dec\ n)\ x)))) \\
exp^c & = & (Y\ \lambda e\ n\ x.if\ (=\ n\ 0)\ 1\ (*\ x\ (e\ (-\ n\ 1)\ x)))
\end{array}
$$

Figure 4.10: Versions of the exponential function

For a baseline, we ran *exp* and exp^c against arguments ranging from 2 to 5. The results are in figure 4.11.

	exp			
x \ y	c_2	c_3	c_4	c_5
c_2	86	93	102	113
c_3	143	174	227	308
c_4	219	334	599	1110
c_5	326	717	1938	4919

	exp^c			
x	2	3	4	5
2	13	13	13	13
3	17	17	17	17
4	21	21	21	21
5	25	25	25	25

Figure 4.11: Exponential function running times

Next, we ran against three kinds of partial evaluation. First, we used M with the *exp*. Next, we used M^c with *exp*, to see the effect of adding the extra combinators to the representation. The results are in figures 4.12 and 4.13. While both M and M^c produce identical residual programs, the cost of the new combinators is a 25% slowdown.

The payoff comes when we use the expanded language. The next test shows M^c applied to

	$((M\ exp\ x)\ y)$				
$c_x \backslash y$	$size(M\ exp\ x)$	c_2	c_3	c_4	c_5
c_2	599	12	19	28	39
c_3	887	25	56	109	190
c_4	1218	50	165	430	941
c_5	1529	99	490	1711	4692

Figure 4.12: $(M\ exp)$

	$((M^c\ exp\ x)\ y)$					
x/y	$size(M^c\ exp\ x)$	ρ	2	3	4	5
2	808	0.74	12	19	28	39
3	1177	0.75	25	56	109	190
4	1596	0.76	50	165	430	941
5	2065	0.77	99	490	1711	4692

Figure 4.13: $(M^c\ exp)$

exp^c. The results are in figure 4.14, with ρ values given to compare with both M and M^c applied to *exp*. As the complexity of the Church-numeral operations increase, we see better performance from $(M^c\ exp^c)$, and eventually we are able to do even better than the smaller representation, as the elimination of the Church-arithmetic overhead catches up to the addition of the extra representational overhead. Further, for exp^c, the specialized versions are very efficient, only needing the initial β-reduction.[1]

	Size and Speedup			β-reductions			
x	$(M^c\ exp^c\ x)$	$\rho(M\ exp\ c_x)$	$\rho(M^c\ exp\ c_x)$	2	3	4	5
2	719	0.83	1.12	1	1	1	1
3	983	0.90	1.20	1	1	1	1
4	1250	0.97	1.28	1	1	1	1
5	1517	1.05	1.36	1	1	1	1

Figure 4.14: $(M^c\ exp^c)$

We are also able to perform the second projection with M^c, but only for the simplest expressions (like $\lambda x.x$), and even then it is very slow. An experiment with the second projection of exponential did not terminate after several days.

[1]Of course, the δ-reductions are not taken into account here. The point is that a δ-reduction is much cheaper than a β-reduction.

4.4.2 Adding Strategies

The process of adding strategies to M^c is similar to the process of adding strategies to M. As mentioned before, strategies are only responsible for advising about β-reductions, so they will not be consulted for these other operations, except when they are needed to turn PEVs into expressions. They are simply discarded when passed into the constant representing combinators.

$$
\begin{array}{lcl}
K_{UB} & = & \lambda o\ \sigma\ p. \\
 & & (p\ \lambda z.(\lambda z_f. \\
 & & \quad (D\ if\ (isConst\ z_f) \\
 & & \quad\quad (getConst\ z_f\ \lambda v. \\
 & & \quad\quad\quad \lambda a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}.(c_b\ (o\ v))) \\
 & & \quad\quad \lambda a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}.(a\ (k_{ub}\ o)\ (z_f\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}))\ \sigma) \\
 & & \quad (z\ \sigma\ F)) \\
 & & \ \lambda abc_bc_ik_ik_{ub}k_{ui}k_{bb}k_{bi}.(k_{ub}\ o)) \\
K_I & = & \lambda c\ t\ e\ \sigma.\ (\lambda r\ u.\ (c\ \sigma\ F\quad \lambda a\ b.\ r \\
 & & \qquad\qquad\qquad\qquad\quad \lambda a.\ r \\
 & & \qquad\qquad\qquad\qquad\quad \lambda a.\ r \\
 & & \qquad\qquad\qquad\qquad\quad u \\
 & & \qquad\qquad\qquad\qquad\quad \lambda i.\ r \\
 & & \qquad\qquad\qquad\qquad\quad \lambda c\ t\ e.\ r \\
 & & \qquad\qquad\qquad\qquad\quad \lambda u.\ r \\
 & & \qquad\qquad\qquad\qquad\quad \lambda u.\ r \\
 & & \qquad\qquad\qquad\qquad\quad \lambda o.\ r \\
 & & \qquad\qquad\qquad\qquad\quad \lambda o.\ r) \\
 & & (D\ \lambda a\ b\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}.(k_i\quad (c\ \sigma\ F\ a\ b\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (t\ \sigma\ F\ a\ b\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (e\ \sigma\ F\ a\ b\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}))\ \sigma) \\
 & & \lambda b.(\texttt{if}\ b\ t\ e\ \sigma))
\end{array}
$$

Figure 4.15: Sample of changes needed for P^c

By adding conditionals and integer arithmetic to the language explicitly, we eliminate the need to perform these functions via β-reduction. In figure 4.16, we see the result of P and P^c running on various programs. The arguments to P^c are in typed form. Figure 4.15 shows the changes in two of the combinators.

In figures 4.16, 4.17, and 4.18 we see the corresponding experiments to figures 4.12, 4.13 and 4.14. The effect of moving to a larger language is much more striking. In the M^c experiments, we needed much more numerical computation to occur before the new representation paid off. In the P^c

	$((P\ exp\ x)\ y)$				
$c_x \backslash y$	$size(P\ exp\ x)$	c_2	c_3	c_4	c_5
c_2	1409	12	19	28	39
c_3	2648	25	56	109	190
c_4	4478	50	165	430	941
c_5	7021	99	490	1711	4692

Figure 4.16: $(P\ exp)$ with Σ_{all}

	$((P^c\ exp\ x)\ y)$					
$c_x \backslash y$	$size(P^c\ exp\ x)$	ρ	2	3	4	5
2	1629	0.86	12	19	28	39
3	2983	0.89	25	56	109	190
4	4967	0.90	50	165	430	941
5	7712	0.91	99	490	1711	4692

Figure 4.17: $(P^c\ exp)$ with Σ_{all}

	Size and Speedup			β-reductions			
x	$(P^c\ exp^c\ x)$	$\rho(P\ exp\ c_x)$	$\rho(P^c\ exp\ c_x)$	2	3	4	5
2	985	1.43	1.65	1	1	1	1
3	1477	1.79	2.02	1	1	1	1
4	2053	2.18	2.42	1	1	1	1
5	2710	2.60	2.85	1	1	1	1

Figure 4.18: $(P^c\ exp^c)$ with Σ_{all}

experiments, the new representation pays off immediately. This is to be expected, because the strategy versions of the evaluator increase the expense of a β-reduction relative to the Mogensen versions.

4.5 Strategies and η-Reduction

All of the strategies we have presented to this point have operated on PEVs by selecting one component of the pair or the other, and then either applying or concatenating them. But there is nothing intrinsic to strategies that restrict their operation to selecting and combining. In particular, by making use of η expansion, we can achieve many of the benefits seen in other partial evaluators, such as [7]. In fact, the η rule is one of many so-called *binding-time improvements*, a subject of active research[27, 14, 8].

A good example of where η-expansion could be helpful is in terms like $(\lambda ab.(f\ a\ b)\ A\ B)$ in which we want to residualize the inner application to A but still perform the outer application to B. Earlier we discussed the *Expand N Then* strategy, which would allow the application to B, but at the cost of expanding the application to A, even if it did not meet the criteria we desired for expansion. The *Expand N Then* strategy forces us to unfold an application that we would rather have residualized.

For this example, what we really want is to take a term of the form $(\lambda ab.(f\ a\ b)\ A\ B)$ and reduce it to $(\lambda a.(f\ a\ B)\ A)$. Using the η-rule, we can build a strategy that will do this for us. Typically, a strategy that residualizes an application $(M\ N)$ does so by taking the HOAS representations of M and N, and then building an application $App(M, N)$ with them. Because the result is a represented λ term, it cannot be evaluated any further unless we embed a partial evaluator in our strategies.

Instead of creating a completely residualized term, we could instead create a partially residualized term in which the application to A was residualized, but then abstract over the call that would have applied the result to B. This would look like $\lambda y.(App(Abs(x, ((M_1\ (D\ Var(x)))_1\ y)_2), X))$. The inner call to $(M\ (D\ Var(x))$ "uses up" the inner abstraction, allowing access to the outer abstraction. The y in this function should have type PEV, which gives this term the type $PEV \rightarrow Exp$. If we pair this term with an Exp, we have a new PEV.

Modifying the $\Sigma_{small,n}$ strategy with this new form of residualization yields figure 4.19.

```
$\Sigma_{2nd,n} = \lambda \pi \pi'.$
  let $r = (\pi\ \Sigma_{2nd,n})$ in
  let $r' = (\pi'\ \Sigma_{2nd,n})$ in
    if $(size\ r'_2) < n$
        then $(r_1\ \pi')$
        else $\lambda s.(s$
            $\lambda y.App(Abs(\lambda x.((r_1\ (D\ Var(x)))_1\ y)), r'_2)$
            $App(r\ F, r'\ F))$
```

Figure 4.19: *Expand Second*

When we apply this to the example, we observe the following sequence:

$$
\begin{aligned}
(\ll (\lambda ab.\lambda f.(f\ a\ b)\ \lambda q.(q\ q)\ \lambda r.r) \gg\ \Sigma_{2nd,3}) &\quad \Rightarrow \\
(\Sigma_{2nd,3}\ \ll (\lambda ab.\lambda f.(f\ a\ b)\ \lambda q.(q\ q)) \gg\ \ll \lambda r.r \gg &\quad \Rightarrow \\
(\ll (\lambda ab.\lambda f.(f\ a\ b)\ \lambda q.(q\ q)) \gg\ \Sigma_{2nd,3}\ \ll \lambda r.r \gg &\quad \Rightarrow \\
((\Sigma_{2nd,3}\ \ll \lambda ab.\lambda f.(f\ a\ b) \gg\ \ll \lambda q.(q\ q) \gg\ \ll \lambda r.r \gg &\quad \Rightarrow^+ \\
(\lambda y.App(Abs(x, (\ll \lambda ab.\lambda f.(f\ a\ b) \gg\ (D\ Var(x)))), Abs(q, App(q, q)))\ \ll \lambda r.r \gg) &\quad \Rightarrow \\
(\lambda y.App(Abs(x, (\ll \lambda ab.\lambda f.(f\ a\ b) \gg\ \Sigma_{2nd,3}\ (D\ Var(x))\ y)), & \\
Abs(q, App(q, q)))\ \ll \lambda r.r \gg) &\quad \Rightarrow^+ \\
App(Abs(x, Abs(f, App((\ll (f\ (D\ Var(x))\ \lambda r.r) \gg\ \Sigma_{2nd,3}), Abs(q, App(q, q)))\ ))) &\quad \Rightarrow^+ \\
App(Abs(x, Abs(f, App(App(App(Var(f), Var(x)), Abs(r, Var(r))), Abs(q, App(q, q)))))) &\quad \Rightarrow^+ \\
[\![(\lambda xf.(((f\ x)\ \lambda r.r))\ \lambda q.(q\ q))]\!] &
\end{aligned}
$$

The effect of this strategy is to change one of the assumptions of the original evaluator: that the applications are processed by selecting the function or textual versions of the function and its argument. In this case, a new kind of function is produced that accomplishes a completely different kind of computation. The use of η-expansion (the $Abs(x, \cdots)$, in this case) is seen in many different areas of partial evaluation, and it is perhaps not surprising to find that it is useful for this technique as well.

This technique can be used in any strategy. The effect is to change the meaning of residualization from an operation that freezes the current and future applications to an operation that only freezes the current application, but lets future applications proceed. However, adding this technique to the other strategies examined so far is not as interesting, as their decisions to residualize are made more uniformly that this strategy.

4.6 Adding Contexts

One future direction to study is the capability to reduce terms like this one:

$$\lambda x.(\texttt{not}\ (\texttt{if}\ x\ \texttt{true}\ \texttt{false}))$$

No matter what x is, we can move the `not` into the branches of the `if` expression. The current representation makes that difficult, because to get to the subexpressions of the `if` you have to residualize the expression first, which means the best you could do is output

$$\lambda x.\texttt{if}\ x\ (\texttt{not}\ \texttt{true})\ (\texttt{not}\ \texttt{false})$$

The η trick used earlier does not seem to be applicable, because we are not content just to change how `not` is applied, but we actually need to move it into the body of the `if` expression. PEVs don't have enough structure to allow for such movement, at least in their current form. One solution to this is to change the type of PEV from *Stragety* $\rightarrow$ *Result* to *Strategy* $\rightarrow$ *Continuation*, where continuations take results. This would allow the argument to a function to determine how it is applied.

$$
\begin{array}{ll}
T := & \lambda ab.a \\
F := & \lambda ab.b \\
Q := & \lambda qms.\lambda p.(p\ \lambda q'v.(q'\ q'\ \lambda abcc_bc_ik_ik_{ub}k_{ui}k_{bb}k_{bi}. \\
 & \quad (a\ (m\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}) \\
 & \qquad (v\ s\ F\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}))\ s) \\
 & \quad \lambda q'.m \\
 & \quad q) \\
D := & (Q\ Q) \\
A := & \lambda mns.(s\ m\ n) \\
B := & \lambda gsp.(p\ \lambda x.(g\ x\ s) \\
 & \qquad \lambda a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}. \\
 & \quad (b\ \lambda w.(g\ (D\ \lambda abcc_bc_ik_ik_{ub}k_{ui}k_{bb}k_{bi}.(c\ w)) \\
 & \qquad s\ F\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}))) \\
C := & \lambda x.x \\
C_B := & \lambda b_vsp.(p\lambda e.e\lambda abcc_bc_ik_ik_{ub}k_{ui}k_{bb}k_{bi}.(c_b\ bv)) \\
C_I := & \lambda i_vsp.(p\lambda e.e\lambda abcc_bc_ik_ik_{ub}k_{ui}k_{bb}k_{bi}.(c_i\ iv)) \\
K_I := & \lambda a_{cond}tes.(\lambda a_{unfold}a_{resid}.(a_{cond}\ s\ F\ \lambda ab.a_{resid}\ \lambda a.a_{resid}\ \lambda c.a_{resid} \\
 & \qquad a_{unfold}\ \lambda i.a_{resid}\ \lambda cte.a_{resid}\ \lambda u.a_{resid}\ \lambda u.a_{resid}\ \lambda o.a_{resid}\ \lambda o.a_{resid}) \\
 & \quad \lambda b.(\texttt{if}\ b\ t\ e\ s) \\
 & \quad (D\ \lambda abcc_bc_ik_ik_{ub}k_{ui}k_{bb}k_{bi}.(k_i\ (a_{cond}\ s\ F\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}) \\
 & \qquad (t\ s\ F\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}) \\
 & \qquad (e\ s\ F\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}))\ s)) \\
p_{const} := & \lambda i_v.(i_v \\
 & \quad \lambda ab.\texttt{false}\lambda b.\texttt{false}\ \lambda c.\texttt{false}\ \lambda b.\texttt{true}\ \lambda i.\texttt{true} \\
 & \qquad \lambda cte.\texttt{false}\ \lambda u.\texttt{false}\ \lambda u.\texttt{false}\ \lambda o.\texttt{false}\ \lambda o.\texttt{false}) \\
g_{const} := & \lambda i_vk.(i_v\ \lambda ab.(k\ \lambda e.e)\ \lambda b.(k\ \lambda e.e)\ \lambda c.(k\ \lambda e.e)\ \lambda b.(k\ b)\ \lambda i.(k\ i) \\
 & \quad \lambda cte.(k\ \lambda e.e)\ \lambda u.(k\ \lambda e.e)\ \lambda u.(k\ \lambda e.e)\ \lambda o.(k\ \lambda e.e)\ \lambda o.(k\ \lambda e.e))
\end{array}
$$

Figure 4.20: Source of P^C, part 1

```
$K_{UB} :=$  $\lambda osp.$
    $(p\ \lambda z_z.(\lambda z_{zf}.$
       $(D\ \texttt{if}(p_{const}\ z_{zf})$
          $(g_{const}\ z_{zf}\ \lambda v.$
             $\lambda abcc_b c_i k_i k_{ub} k_{ui} k_{bb} k_{bi}.(c_b\ (o\ v)))$
          $\lambda abcc_b c_i k_i k_{ub} k_{ui} k_{bb} k_{bi}.(a\ (k_{ub}\ o)\ (z_{zf}\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}))$
       $s)$
       $(z_z\ s\ F))$
       $\lambda abcc_b c_i k_i k_{ub} k_{ui} k_{bb} k_{bi}.(k_{ub}\ o))$
$K_{UI} :=$  $\lambda osp.$
    $(p\ \lambda z_z.(\lambda z_{zf}.$
       $(D\ \texttt{if}(p_{const}\ z_{zf})$
          $(g_{const}\ z_{zf}\ \lambda v.$
             $\lambda abcc_b c_i k_i k_{ub} k_{ui} k_{bb} k_{bi}.(c_i\ (o\ v)))$
          $\lambda abcc_b c_i k_i k_{ub} k_{ui} k_{bb} k_{bi}.(a\ (k_{ui}\ o)\ (z_{zf}\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}))$
       $s)$
       $(z_z\ s\ F))$
       $\lambda abc_b c_i k_i k_{ub} k_{ui} k_{bb} k_{bi}.(k_{ui}\ o))$
$K_{BB} :=$  $\lambda osp.$
    $(p\ \lambda z_z.(\lambda z_{zf}.$
       $\texttt{if}\ (p_{const}\ z_{zf})$
          $(g_{const}\ z_{zf}\ \lambda v.(K_{UB}\ (o\ v)\ s))$
             $(D\ \lambda abcc_b c_i k_i k_{ub} k_{ui} k_{bb} k_{bi}.(a\ (k_{bb}\ o)\ (z_{zf}\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi})\ s)$
          $(z_z\ s\ F))$
       $\lambda abcc_b c_i k_i k_{ub} k_{ui} k_{bb} k_{bi}.(k_{bb}\ o))$
$K_{BI} :=$  $\lambda osp.$
    $(p\ \lambda z_z.(\lambda z_{zf}.$
       $\texttt{if}\ (p_{const}\ z_{zf})$
          $(g_{const}\ z_{zf}\ \lambda v.(K_{UI}\ (o\ v)\ s))$
             $(D\ \lambda abcc_b c_i k_i k_{ub} k_{ui} k_{bb} k_{bi}.(a\ (k_{bi}\ o)\ (z_{zf}\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}))\ s)$
          $(z_z\ s\ F))$
       $\lambda abcc_b c_i k_i k_{ub} k_{ui} k_{bb} k_{bi}.(k_{bi}\ o))$
$R :=$       $\lambda m.(m\ A\ B\ C\ C_B\ C_I\ K_I\ K_{UB}\ K_{UI}\ K_{BB}\ K_{BI});$
$P :=$       $\lambda mns.(R\ \lambda abcc_b c_i k_i k_{ub} k_{ui} k_{bb} k_{bi}.$
       $(a\ (m\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi})$
          $(n\ a\ b\ c\ c_b\ c_i\ k_i\ k_{ub}\ k_{ui}\ k_{bb}\ k_{bi}))$
          $s\ F)$
```

Figure 4.21: Source of P^C, part 2

Chapter 5

Concrete Syntax

We have shown in the previous chapters how strategies can add a level of control to an online partial evaluator. While maintaining much of the simplicity of the underlying partial evaluator, strategies themselves can be arbitrarily complex. Part of the expressive power comes from the ability to compose strategies, so that we can choose the specific algorithms we want to use and combine them.

This ability of strategies to be composed suggests a system where different kinds of functions are composed to perform the partial evaluation. As we saw with the *expand second* strategy, we can think of strategies as an implementation of the β-rule, where the role of the strategy is to determine whether or not the β-rule should be performed, and to perform the appropriate code transformation.

In a larger language, it is common to implement a reduction via a transformation from expressions to expressions. In this chapter we explore what would happen if we implemented strategies this way, manipulating a concrete syntax rather than relying on the underlying reducer to perform reductions for us (such as β-reduction). This also gives us the opportunity to add other kinds of reductions to the system.

These new strategies will be called *transformers* to emphasize their role. Each transformer implements a reduction in the language. These transformers are then composed to make one general transformer. By allowing transformers to annotate the expressions they process, these transformers can communicate with each other and make use of statically determined information such as binding-times and types.

To test this idea, we made a small implementation, written in OCaml. Our target language is a small functional language, which has if, anonymous functions, arithmetic, and let. The grammar for the language is in figure 5.1.

$$\begin{array}{rcl} P & ::= & \texttt{let}\ v\ =\ E \\ E & ::= & \texttt{fun}\ v\ \rightarrow\ E \mid (E\ E) \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\ & & \mid \texttt{let}\ v\ =\ E\ \texttt{in}\ E \mid c \mid E\ \oplus\ E \end{array}$$

where v represents variables, c represents constants, and $\oplus$ represents arithmetic operations. A program is a series of definitions P.

Figure 5.1: Grammar for the target language

A *transformer* is a function which takes an expression as an input, and returns an expression as its output. In this system, each transformer corresponds to and implements a code transformation needed for reduction, normalization, or partial evaluation. The pattern these transformers will follow is to examine an expression to see if the transformer knows how to process it. If so, the transformer will apply its transformation and return the result. If not, the input expression is returned unchanged. All the transformers in this section apply local transformations.

5.1 Reduction

We can use transformers to perform reduction operations. Each transformer will handle one rule (or a few related rules) of the transition semantics for the language. For example, our language has two `if` rules:

$$\texttt{if}\ \mathit{true}\ \texttt{then}\ x\ \texttt{else}\ y\ \rightarrow\ x$$

$$\texttt{if}\ \mathit{false}\ \texttt{then}\ x\ \texttt{else}\ y\ \rightarrow\ y$$

These can be represented by the following transformer $T_{\texttt{if}}$:

$$\begin{array}{l} T_{\texttt{if}}(e) = \texttt{match}\ e\ \texttt{with} \\ \quad \mid \texttt{if}\ \mathit{true}\ \texttt{then}\ x\ \texttt{else}\ y\ \ \rightarrow x \\ \quad \mid \texttt{if}\ \mathit{false}\ \texttt{then}\ x\ \texttt{else}\ y \rightarrow y \end{array}$$

The `match`/`with` syntax compares e to the two patterns, and returns the value of the right side of the arrow. If no match is found, we return the original expression.

Addition is represented by the arithmetic transformer.

$$\begin{aligned}&T_+(e) = \texttt{match}\ e\ \texttt{with}\\&\quad \mid i\ \texttt{+}\ j \rightarrow i + j, \text{when } i \text{ and } j \text{ are integers}\end{aligned}$$

The T_+ transformer only operates on addition operations, when both of the arguments are integers. Additions where one or both of the arguments are expressions are ignored. We can add optimizations to the transformer by having it watch for special cases, such as when one of the arguments is a zero.

$$\begin{aligned}&T_+(e) = \texttt{match}\ e\ \texttt{with}\\&\quad \mid i\ \texttt{+}\ j \quad \rightarrow i + j, \text{when } i \text{ and } j \text{ are integers}\\&\quad \mid 0\ \texttt{+}\ e' \rightarrow e'\\&\quad \mid e'\ \texttt{+}\ 0 \rightarrow e'\end{aligned}$$

Similar transformers are defined for function application and `let`.

$$\begin{aligned}&T_{\texttt{let}}(e) = \texttt{match}\ e\ \texttt{with}\\&\quad \mid \texttt{let}\ v\ =\ e'\ \texttt{in}\ e'' \rightarrow [e'/v]e''\end{aligned}$$

Where $[e'/v]e''$ is substitution of e' for v in e'', defined in the usual, variable hygienic way.

5.2 Composition

To use these transformers to perform interpretation, we take two steps. First, we compose the local transforms to create a single interpreter transformer T_{int}:

$$T_{int} = T_+ \circ T_{\texttt{if}} \circ \cdots \circ T_{\texttt{let}}$$

Next, we need a function I that performs a recursive descent over input terms, using the fix-point of T_{int}.

```
$I(e) =$ match $fix\ T_{int}(e)$ with
    | (fun $x \rightarrow e'$) $\rightarrow e$
    | $e_1$ then $e_2$ else $e_3$ $\rightarrow$
        $T_{int}$(if $I(e_1)$ then $I(e_2)$ else $I(e_3)$)
    | let $v = e_1$ in $e_2 \rightarrow T_{int}$(let $v = I(e_1)$ in $I(e_2)$)
    | $e_1 \oplus e_2 \rightarrow T_{int}(I(e_1) \oplus I(e_2))$
    | $e_1\ e_2 \rightarrow$ match $I(e_1)$ with
                | (fun $x \rightarrow e'$) $\rightarrow [e2/x]e'$
                | $e'_1 \rightarrow e'_1\ I(e_2)$
    | $c \rightarrow c$
```

We can think of I as a kind of transformer, parameterized over transformer T_{int}.

5.3 Normalization

To perform normalization, we need to create a transformer that will descend into a function's body. To prevent variable name capture, we need to rename the variable before descending. The normalization function N is parameterized over I, which is meant to be an interpreter of the kind described above.

```
$N(I, e) =$ match $I(e)$ with
    | (fun $x \rightarrow e'$) $\rightarrow$ (fun $x^* \rightarrow N([x^*/x]e')$)
    | if $e_1$ then $e_2$ else $e_3$ $\rightarrow$
        $T_{int}$(if $N(e_1)$ then $N(e_2)$ else $N(e_3)$)
    | let $v = e_1$ in $e_2 \rightarrow T_{int}$(let $v = N(e_1)$ in $N(e_2)$)
    | $e_1 \oplus e_2 \rightarrow T_{int}(N(e_1) \oplus N(e_2))$
    | $c \rightarrow c$
```

5.4 Partial Evaluation

Specialization of an expression can produce the same result as normalization, but we often want to stop short of normalization. In order to limit the actions of the normalizer, we introduce staging annotations. Following convention, we will use overline to represent the compile-time stage, and underline to represent the runtime stage. The normalizer and interpreter functions then will be modified to rewrite terms only when they are in the compile-time stage.

5.4.1 Binding Time Information

There are two ways we can make use of binding time analysis in our partial evaluator. The first way is to have the binding time analyzer output two-level code, marking the dynamic code as runtime. This will cause the partial evaluator to reduce only the code marked as static, much as a standard offline partial evaluator.

The second way is to introduce binding-time annotations to the expressions. To implement offline partial evaluation, we have a transformer check the annotations. If the annotation indicates that the expression is dynamic, the code's staging level is raised to runtime.

$$
\begin{aligned}
&T_{bta}(e) = \texttt{match}\ e\ \texttt{with} \\
&\quad \mid (\texttt{fun}_d\ x\ \rightarrow\ e')\ \rightarrow\ (\overline{\texttt{fun}}\ x \rightarrow\ e') \\
&\quad \mid \texttt{if}_d\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \rightarrow \\
&\qquad\quad \overline{\texttt{if}}\ N(e_1)\ \overline{\texttt{then}}\ N(e_2)\ \overline{\texttt{else}}\ N(e_3) \\
&\quad \mid \texttt{let}_d\ v\ =\ e_1\ \texttt{in}_d\ e_2\ \rightarrow\ \overline{\texttt{let}}\ v\ =\ e_1\ \overline{\texttt{in}}\ e_2 \\
&\quad \mid e_1\ \oplus_d\ e_2\ \rightarrow\ e_1\ \overline{\oplus}\ e_2
\end{aligned}
$$

The order in which the transformers are composed is significant. Any transformers that come before T_{bta} will ignore binding-time annotations. The transformers coming afterward will be forced to use them, unless they ignore the staging levels. This allows us the possibility of making partial evaluators that use a hybrid of online and offline methods.

5.4.2 Heuristics

If a statically known function is applied to a dynamic conditional with static branches, we can often create more opportunities for specialization by moving the function into the branches. For example:

$$\begin{aligned}\overline{\texttt{let}}\ f\ x\ &=\ x \mathbin{\overline{+}} 10\ \overline{\texttt{in}}\\ f\ (\underline{\texttt{if}}\ \underline{y}\ &\underline{\texttt{then}}\ 10\ \underline{\texttt{else}}\ 20)\end{aligned}$$

can be rewritten as

$$\begin{aligned}\overline{\texttt{let}}\ f\ x\ &=\ x \mathbin{\overline{+}} 10\ \overline{\texttt{in}}\\ \underline{\texttt{if}}\ \underline{y}\ &\underline{\texttt{then}}\ f(10)\ \underline{\texttt{else}}\ f(20)\end{aligned}$$

and finally reduced to

$$\underline{\texttt{if}}\ \underline{y}\ \underline{\texttt{then}}\ 20\ \underline{\texttt{else}}\ 30$$

This heuristic can be implemented as follows:

$$\begin{aligned}&T_{dycon}(e) = \texttt{match}\ e\ \texttt{with}\\ &\quad \mid \overline{e_1}\ (\texttt{if}\ \underline{e_2}\ \texttt{then}\ e_3\ \texttt{else}\ e_4)\ \rightarrow\\ &\qquad \texttt{if}\ \underline{e_2}\ \texttt{then}\ \overline{e_1}\ e_3\ \texttt{else}\ \overline{e_1}\ e_4\end{aligned}$$

There would be similar cases for operators, and `match/with` (if the language had them). This transformer corresponds to the static application "code motion" rule of [9].

5.4.3 Bounded Static Variation

A more complex version of this makes use of the knowledge gained by taking a branch, and is commonly known as "The Trick"[17, 9]. If the conditional is simply a variable, then the value of the variable will be known in the branches, even if it is marked dynamic.

$$\begin{aligned}&T_{trick}(e) = \texttt{match}\ e\ \texttt{with}\\ &\quad \mid \overline{e_1}\ (\texttt{if}\ \underline{x}\ \texttt{then}\ e_3\ \texttt{else}\ e_4)\ \rightarrow\\ &\qquad \texttt{if}\ \underline{x}\ \texttt{then}\ \overline{e_1}([true/x]e_3)\ \texttt{else}\ \overline{e_1}([false/x]e_4)\end{aligned}$$

5.5 Global Transformations

Local transformations require access only to the immediate expression being processed. Transformations like function unrolling and specialization require access to the program source.

5.5.1 Function Inlining

If the transformers are allowed read access to the source of the program, we can perform function inlining.

$$\begin{aligned} T_{inline}(f, e) &= \texttt{match } e \texttt{ with} \\ \mid f\ e \ &\rightarrow\ [body(f)/arg(f)]e \end{aligned}$$

5.5.2 Specialization

If transformers are allowed write access to the source of the program, then we can perform specialization.

For example, if we wanted to specialize the two-argument function g with respect to its first argument if the argument is a constant, we could use the transformer:

$$\begin{aligned} T_{specialize}(g, e) &= \texttt{match } e \texttt{ with} \\ \mid g\ c\ e' \rightarrow\ & \texttt{set } body(g_c) = [c/var_1(g)]body(g_c); \\ & \texttt{set } var_1(g_c) = var_2(g_c); \\ & \texttt{set } arity(g_c) = 1; \\ & g_c\ e' \end{aligned}$$

This is presented here in an imperative style for brevity. In a functional style we would use standard “plumbing” techniques to thread the source of the program.

We can also write transformers that force the residualization of certain functions, and even work on functions that we anticipate being created.

5.6 Usage

The compositional framework allows us to write this as a list of instructions to the evaluator. For example, if we wanted to specialize g with respect to the second argument, unfold f, and residualize calls to h, the code for the evaluator would look like:

$$pe\ (T_{int}\ o\ T_{specialize_{g,2}}\ o\ T_{unfold_f}\ o\ T_{residualize_h})$$

In our implementation, each transformer registers itself with a central registry, and reports to it each time it takes an action. When the evaluator is done, it prints a summary of how many times each transformer was used. This is useful for understanding which transformers are having an effect.

5.7 Conclusions and Related Work

Transformers are a related to strategies in that they provide an implementation for particular reductions in the system. Rather than using the underlying reducer to perform the reduction, transformers manipulate an abstract syntax tree. We can build new transformers by composing transformers we have already written, in the same way that we can build new strategies by composing old ones. Composing transformers has the the effect of pipelining the transformations.

This idea is similar to DyC[13], and the Staged Compilation Framework (SCF)[25] developed at the University of Washington. In their framwork a pipeline of transformations is used to perform stages of a partially evaluating compiler, targeted toward the C language. The SCF stages are more coarse-grained than the transformers of this chapter, in that they process entire expression trees rather than individual nodes. Further, these stages are meant to be run once each, while transformers potentially run multiple times over the same nodes.

Chapter 6

The λ-Calculus Reducer

During these tests, the third projection was very difficult to achieve. The combinatorial nature of the problem, and the fact that the strategy version of the evaluator is only slightly larger than Mogensen's, lead to the question: what if non-termination is caused by inefficiencies in the underlying reducer?

The main cause for this concern is the fact that our λ-calculus reducer uses a graph-based representation of the λ-terms. An expression e is represented as shared information when two or more other expressions have pointers to it. The advantage of this representation is clear: if two expressions M and N share a subexpression e, then if M causes e to be evaluated, N is able to use the result of that evaluation for free. But, part of the β-reduction algorithm involves copying elements of the graph. This can cause problems in that, for example, if M is copied to another expression M', it is possible that the subexpression e will also be copied, resulting in the computation being done twice.

In this chapter we discuss the reducer and the optimizations made to it. In section 6.1 we discuss the design of Peyton-Jones' interpreter[23], which was the starting point for our reducer. In section 6.2 we discuss how we modified the interpreter to make it a reducer and the efficiency gains we were able to realize.

6.1 Standard λ Interpretation

We used [23] as a starting point. It is an interpreter for a lazy λ-calculus. Rather than using an environment, closures, and thunks, this interpreter represents λ-terms completely using an abstract syntax *graph*, which makes explicit the computations that are shared.

The key computational step in λ-calculus is β-reduction, which is the λ-calculus implementation of a function call. In β-reduction, a *redex* (reducible expression) consisting of an abstraction applied to an argument is rewritten. The argument is substituted into the body of the abstraction wherever the variable bound by the abstraction appears. An example of a β-reduction is in figure 6.1. The argument $(\lambda q.q\ \lambda r.r)$ is substituted for x.

$$\begin{tikzcd}
 & & & @ \arrow[dll] \arrow[drr] & & & & \Rightarrow & & & & @ \arrow[dll] \arrow[drr] & & & \\
 & \lambda x \arrow[d] & & & & @ \arrow[dl] \arrow[dr] & & & & @ \arrow[dl] \arrow[dr] & & & & @ \arrow[dl] \arrow[dr] & \\
 & @ \arrow[dl] \arrow[dr] & & & \lambda q \arrow[d] & & \lambda r \arrow[d] & & \lambda q \arrow[d] & & \lambda r \arrow[d] & & \lambda q \arrow[d] & & \lambda r \arrow[d] \\
x & & x & & q & & r & & q & & r & & q & & r
\end{tikzcd}$$

Figure 6.1: Simple β-reduction, using trees

From the figure we can also see why graphs are used instead of trees: the $(\lambda q.q\ \lambda r.r)$ computation has been duplicated, and will now be performed twice. Figure 6.2 shows the same computation using graphs instead of trees. The idea of using pointers to share computations goes back to 1971[31].

$$\begin{tikzcd}
 & @ \arrow[d, bend right] \arrow[d, bend left] & \\
 & @ \arrow[dl] \arrow[dr] & \\
\lambda q \arrow[d] & & \lambda r \arrow[d] \\
q & & r
\end{tikzcd}$$

Figure 6.2: Simple β-reduction, using graphs

Once we start using a graph, it becomes important that we do not destructively update the function when performing a substitution, because it may be shared by other applications. For example, if the $\lambda x.(x\ x)$ in the figure were applied to a different argument at some other point, we would get an incorrect result if we simply replaced the occurrences of x with pointers to $(\lambda q.q\ \lambda r.r)$. Instead, we need to make a copy of the function, substituting the argument for the variable as we go.

6.1.1 Instantiation

This process of copying the function is called *instantiation*, which is discussed in chapter 12 of [23]. The instantiation algorithm implements substitution in the context of a graph representation of an expression.

The algorithm has three inputs: a reference to the body of the function being applied, the name of the variable being substituted, and a reference to the argument of the function. The instantiation algorithm then needs to do three things. First, it needs to create a copy of the body of the function that has just been applied. Again, we cannot simply update the body in place because this function may be shared by other parts of the graph; to do so would cause subsequent applications to be performed incorrectly. Second, it needs to replace the bound variables by references to the argument to which the function was applied. By referring to the parameter instead of making copies, the results of reducing the argument are made available to the other parts of the graph that use it. This is a standard call-by-need implementation of graph-based λ-calculus interpreters. Finally, the algorithm needs to replace the application node that was just reduced with the result; this also turns out to require some subtlety.

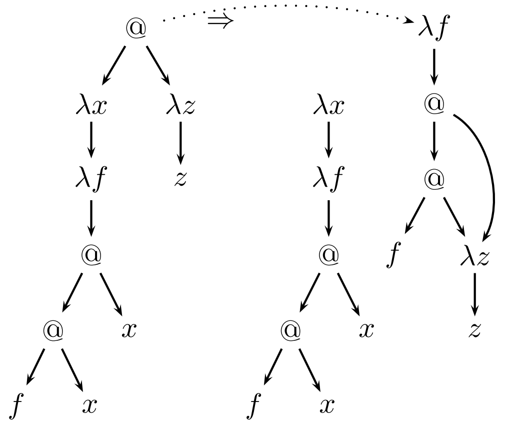

Figure 6.3: Instantiation

Figure 6.3 illustrates a β-reduction. The function $\lambda xf.((f\ x)\ x)$ is applies to $\lambda z.z$. The result is $\lambda f.((f\ \lambda z.z)\ \lambda z.z)$. As can be seen in the graph, the $\lambda z.z$ is shared. The original function is still in memory, disconnected from this computation. It is kept in case it is applied in some other context. The dotted line represents that the application node has been overwritten by the λf abstraction.

6.1.2 Indirection Nodes

When an instantiation occurs, we need to replace the application node with the result node to make the result available to the rest of the computation. But this must be done with some care, because it is easy to lose many opportunities for sharing. In figure 6.3 the result of the instantiation was an abstraction. But suppose instead that the result of the function application was to return another application node that was shared by some other expression, as in figure 6.4.

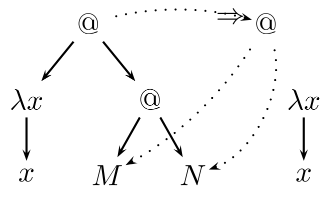

Figure 6.4: Instantiation with inappropriate copying

We can see that there are now two nodes that will cause an application of $(M\ N)$; one was the argument to the function, and the other is the new root of the expression after the β-reduction is complete. If there is a link to the original application node in some other part of the expression, then it is possible for $(M\ N)$ to be performed twice.

The solution is to introduce a new kind of node, called an *indirection node.* It is emitted whenever the result of a function call is an application node that was not generated by the instantiation itself. This indirection node acts as a reference, and allows the computation to be shared. This precaution must also be taken whenever a computation-producing node (such as `if`) is returned from a function. Figure 6.5 gives the same example as figure 6.4 using indirection nodes.

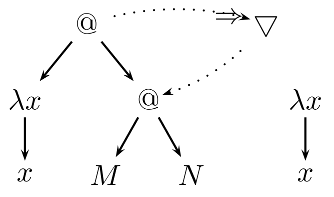

Figure 6.5: Instantiation with an indirection node

6.1.3 Improving the Instantiation Algorithm

Aggressive Sharing

The instantiation routine given in [23] specifies that all parts of the function being instantiated should be copied. This is not what we want, however. Every time an application node is copied, there is the risk that a computation will now be duplicated unless it is certain that no other links to the application node exist. So it is important that the instantiation routine copy as little of the function body as possible. Instead, using techniques similar to those published by Okasaki [21, 22], terms from the original function should be shared, so that any results of computations performed inside the function body are shared. Figure 6.6 shows a function body that contains a sub-term $(\lambda q.q\ \lambda z.z)$. This sub-term does not depend on the argument passed to the function, which means that this computation could be shared among all the calls made by this function. In the example, the function is applied twice, once to M, and later to N. The figure shows what happens in the first application: the body of the function is instantiated, and the $(\lambda q.q\ \lambda z.z)$ part is shared with the original function. The node of the original application to M is overwritten by the result $((\lambda q.q\ \lambda z.z)\ M)$. The next step reduces $(\lambda q.q\ \lambda z.z)$ to $\lambda z.z$. Both the currently active version of the function and the future application of this function to N are able to benefit from this computation. Had we copied everything in the function body, the term $(\lambda q.q\ \lambda z.z)$ would have been reduced twice.

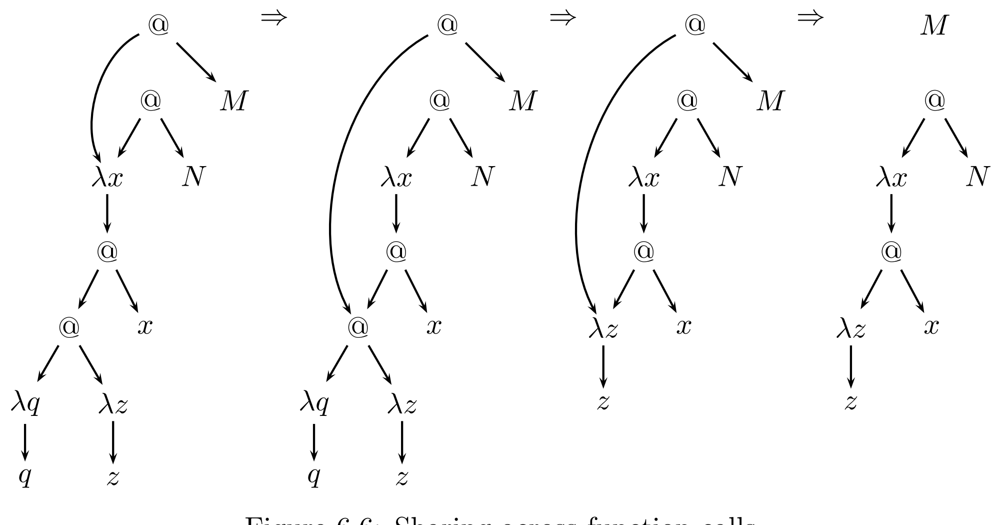

Figure 6.6: Sharing across function calls

One might argue that the function should not have been written with an instance of $(\lambda q.q\ \lambda z.z)$ in it, but this is not always under our control. The $(\lambda q.q\ \lambda z.z)$ could have been placed there by yet another function call occurring earlier in the computation. Since we do not evaluate the arguments of functions before substituting, we must assume that this kind of situation will occur.

Fortunately, it is fairly easy to tell which parts of the term should be shared. When instantiating a node that has sub-nodes, we first instantiate the child nodes, and then check the results. If they are returned back to us unchanged (which we can tell because they will have the same memory location as the original arguments) then there is no need to recreate the node, we simply return the one that was given to us. For example, if we instantiate an application node $(M\ N)$, we first need to instantiate M and N. Supposing we save the results in variables M' and N', we check to see if $M{==}M'$ and $N{==}N'$, where `==` denotes memory equality. If both equalities hold, we return the original application node; otherwise we create a new application node $(M'\ N')$ and return it instead.

Caching

A second optimization is suggested by noticing that if a node is reachable by $n > 1$ paths in the body of an abstraction before instantiation, then the corresponding n paths in the body of the result should also all reach the same corresponding node. An example of this is given in figure 6.7. The variable x will be instantiated and the traversal into M will be performed twice, once for each branch of the application above it. Because of the aggressive sharing optimization outlined above, this will not result in a loss of sharing, but we still have to process all the nodes in M (at least) twice.

To solve this problem we added an annotation field to the nodes to keep track of the results of previous traversals. When the instantiation function reaches a node, it first checks to see if the node had been checked before. If so, it simply returns the result from last time. Otherwise, it puts a copy of the result into the annotation field of the node for future reference, before returning it to the caller. Of course, it is very important to reset these annotations after the instantiation is finished, or else future instantiations will produce inaccurate results.

To see the effect of these changes, we ran some of our experiments with both the old and the

$$\begin{tikzcd}
 & \lambda x \arrow[d] & \\
 & \lambda y \arrow[d] & \\
 & @ \arrow[dl] \arrow[dr] & \\
\lambda z \arrow[d] & & x \\
@ \arrow[d, bend right] \arrow[d, bend left] & & \\
M & &
\end{tikzcd}$$

Figure 6.7: Sharing in an instantiated abstraction

new versions of the instantiation function. The results are in figure 6.8. For applications of the partial evaluators, we saw speedups in the range of 5–11%.

Experiment	Old β	New β	Speedup
$M\ Ack\ C_2$	532	492	8.1%
$M\ Ack\ C_3$	1080	971	11.2%
$P\ Ack\ C_2\ \Sigma_{all}$	746	709	5.2%
$P\ Ack\ C_3\ \Sigma_{all}$	1506	1401	7.5%

Figure 6.8: Comparison of instantiation methods

6.2 From Interpretation to Reduction

The algorithm described above is meant for reducing λ-terms to weak head normal form, hereafter WHNF. A term is in WHNF if the term is an abstraction or a variable; or if the term is an application whose function argument is neither an abstraction, nor an application that can be reduced to an abstraction. Figure 6.9 shows four examples of terms in WHNF. The first and second are in WHNF because they are abstractions, and the third and fourth are in WHNF because the function side of the root-level application is not an abstraction, and cannot be reduced to an abstraction.

The functions $\mathcal{W}$ and $\mathcal{B}$ describe the algorithm to convert a term to WHNF: The $\mathcal{B}$ (for "beta") function checks to see if an application can be β-reduced or not. As is standard in a lazy language, arguments to functions are not evaluated until they are used.

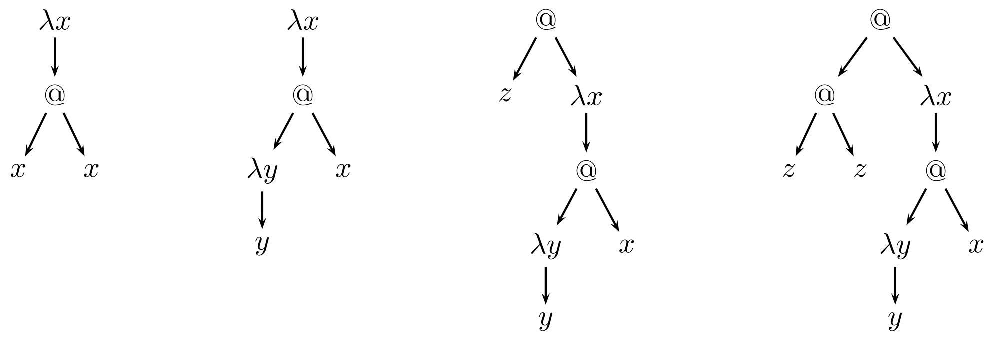

Figure 6.9: Terms in weak head normal form

$$\begin{aligned}
\mathcal{W}[\![x]\!] &\Rightarrow x \\
\mathcal{W}[\![\lambda x.M]\!] &\Rightarrow \lambda x.M \\
\mathcal{W}[\![M@N]\!] &\Rightarrow \mathcal{B}[\![\mathcal{W}[\![M]\!]@N]\!] \\
\mathcal{B}[\![(\lambda x.M)@N]\!] &\Rightarrow \mathcal{W}[\![M[N/x]]\!] \\
\mathcal{B}[\![M]\!] &\Rightarrow M
\end{aligned}$$

6.2.1 Normalization

The terms output by our partial evaluators are represented in HOAS, in which almost all of the structure of the term exists underneath a λ. This means that WHNF will not reduce the terms completely enough to be useful, especially if we want to view the results of our computations. Therefore, we reduce to normal form (NF). A term is in normal form if it is in WHNF and its child nodes are in normal form.

In figure 6.9, the first term is in normal form, but not the following terms. Figure 6.10 shows the result if the reduction to normal form is completed.

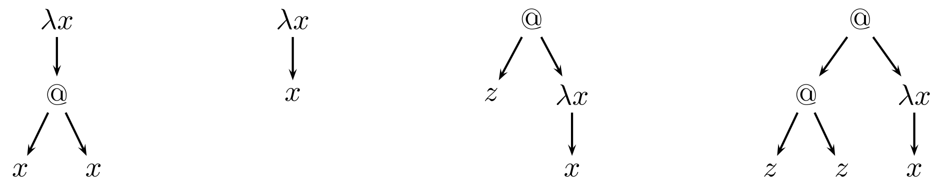

Figure 6.10: Terms in Normal Form

The function $\mathcal{N}$ defines the reduction to normal form. It makes use of $\mathcal{W}$, defined above. To reduce a term to normal form, first reduce it to weak head normal form, and then reduce the children to normal form.

$$\begin{aligned}
\mathcal{N}[\![M]\!] &\Rightarrow \mathcal{N}'[\![\mathcal{W}[\![M]\!]]\!] \\
\mathcal{N}'[\![x]\!] &\Rightarrow x \\
\mathcal{N}'[\![\lambda x.M]\!] &\Rightarrow \lambda x.\mathcal{N}[\![M]\!] \\
\mathcal{N}'[\![M@N]\!] &\Rightarrow \mathcal{N}[\![M]\!]@\mathcal{N}[\![N]\!]
\end{aligned}$$

6.2.2 α-Capture

The normalization algorithm, as given, suffers from α-capture. An α-capture occurs when a variable bound by a certain abstraction later becomes bound by a different abstraction. It also occurs when a variable that is free at one point of the computation becomes bound at a later point of the computation. This represents an incorrect change in the meaning of the variable. This can be especially surprising to a programmer because if the variable is given a different name the capture will not occur.

When reducing to WHNF it is sufficient to be sure that each λ-abstraction binds a uniquely named variable, and that all variables are bound—at that point, no reductions will cause α-capture. This is because a capture occurs when a variable that is free in the body of an application is substituted underneath a λ of the same name. For such a substitution to occur, the variable would either have to be free at the start (contradicting our assumption that all variables are bound) or it would have to be bound by an abstraction further up the graph (contradicting the assumption that we are reducing to WHNF—such reductions do not go underneath λs).

When reducing to normal form this condition is not strong enough. Figure 6.11 shows an example of this. The variables x and w are bound twice each. The last tree shows what happens when we reduce to normal form—the two variables x at the leaves of the term are bound by different abstractions, but there is no way to discover that by examining the graph.

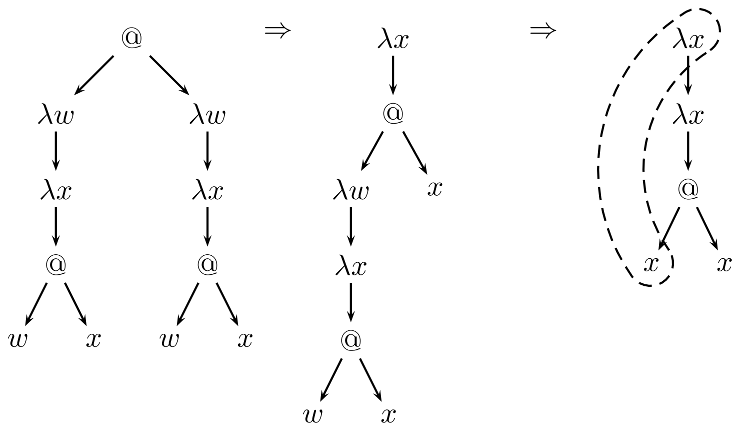

Figure 6.11: Example of α-capture

While the situation in figure 6.11 could be avoided by giving different names to the variables, figure 6.12 illustrates a condition where unique variable naming is not enough. Even if all the bindings start out with unique names, the computation could cause an arbitrary number of copies to be made by means of terms like $\lambda q.(q\ q)$.

α-capture becomes a problem when, during normalization, an abstraction is be instantiated with a variable that is free in the body of the abstraction being instantiated. Such a free variable could be placed deep into the body of the function, below a subexpression containing an abstraction of the same name, thus causing capture.

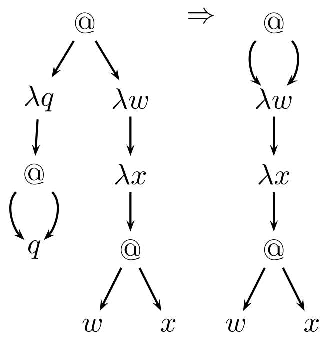

Figure 6.12: Second example of α-capture

The solution turns out to be fairly simple, and is illustrated in figure 6.13. After reducing a term to WHNF, the next step in the normalization process is to descend into the body of the resulting abstraction. It is at this point that the variable bound by the abstraction (*descent variable*) becomes free and is in danger of being captured. If it is found that a potentially capturing abstraction exists in the remainder of the term, then we rename the descent variable. If no such capturing abstraction exists, or if the descent variable does not occur free in the remainder of the term, we can proceed with the reduction without making any changes. (This is the case in figure 6.15.)

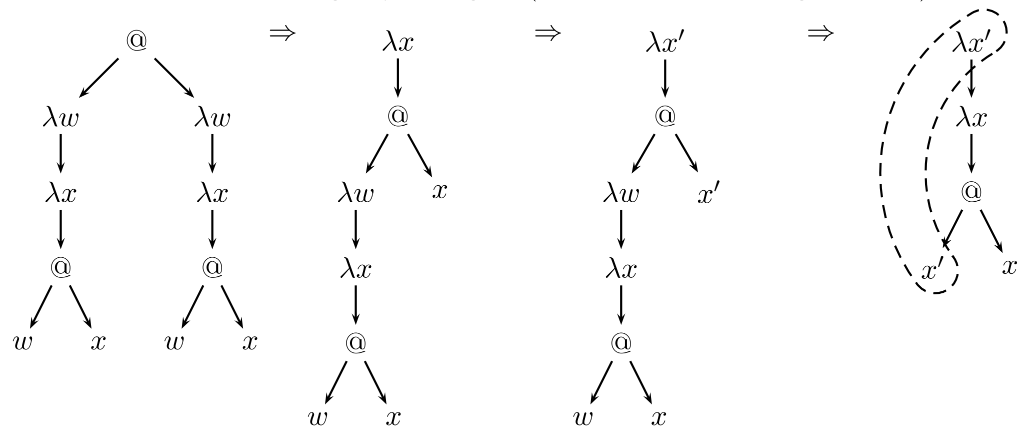

Figure 6.13: Example of α-capture being avoided

The potential for optimization comes in deciding whether or not the descent variable could be captured. In certain cases, such as when the descent variable does not occur, or there are no abstractions in the remainder of the term that bind the same variable, we can say that capture is impossible. The tradeoff tends to be this: that increased optimality reduces the total number of β-reductions performed, while increasing the expense of each reduction. In our implementation, we in fact do not do any checking; we simply rename the abstraction, and instantiate the sub-term with the same instantiation function as before. If none of the variables in the body are bound to that abstraction, then the function will simply reuse the term as-is, and not copy anything.

6.3 Ideal Abstractions

This technique of renaming the variable as we descend into the term brings up some interesting questions. In order to distinguish the variables, we need to rewrite the variable names somehow. The näive way is to rewrite the text of the variable name (e.g., x may become x' after copying), but this quickly becomes cumbersome during long computations, and may make it difficult to read back the results. One technique widely used is to represent variables with pointers to their binders[2, 3]. This allows the textual representation to be less cluttered, while still preserving the uniqueness of the variables.

It turns out that by generalizing this technique, we are able to save memory and reduce the number of β-reductions needed to reduce a term. A λ-calculus expression is represented as a graph which in turn represents a tree with explicit sharing of memory. When the instantiation function is called, the effect is similar to any functional update of a tree-like data-structure. Whatever is on the path between the root node of the function being instantiated and the variables being replaced needs to be copied. In particular, we will need to copy of any abstractions that are on this path. In figure 6.14, we see a similar function call as in figure 6.6. This time, there is a λw along the path between the root and the substitution variable x. As a result, the λw is copied, which has a detrimental effect: since variables must refer to their binders, the w must be copied also, along with everything between it and the root. It is not hard to make an example where a cascade of copying results from this kind of situation. In figure 6.6, the result is that the computation $(\lambda z.(z\ w)\ \lambda q.q)$ is performed twice, even though it has nothing to do with x.

The problem is that having variables refer to their binders is too specific. Examining the two occurrences of λw in the example, we can argue that they are, in some sense, really the same abstraction. Further, notice that there is no possibility that one variable w will be passed into the other variable's abstraction. This situation occurs when abstractions are copied. What we want, then, is to be able to keep track of these related abstractions. Our solution is to represent an abstraction using a pointer to an *Ideal Abstraction*, which is external to the graph. Variables will also point to the ideal abstraction, rather than the abstraction in the graph. (We sometimes call these abstractions *Shadow Abstractions*, á la Plato[6]). The effect will be that if we have to copy a shadow abstraction, the copy will point to the same ideal abstraction as the original, as will the variable bound by it. Thus, the variable can be considered unchanged, and can be shared instead of copied.

In figure 6.15 we see an example of this. This is the second term from figure 6.14, with the variable bindings to their abstractions denoted with a dotted arrow. Alongside of it is the same term after a reduction using ideal abstractions. Though the λw is still copied, we are still able to share the variable w. This in turn means we don't have to copy the $(\lambda z.(z\ w)\ \lambda q.q)$ term, so it will only be reduced once.

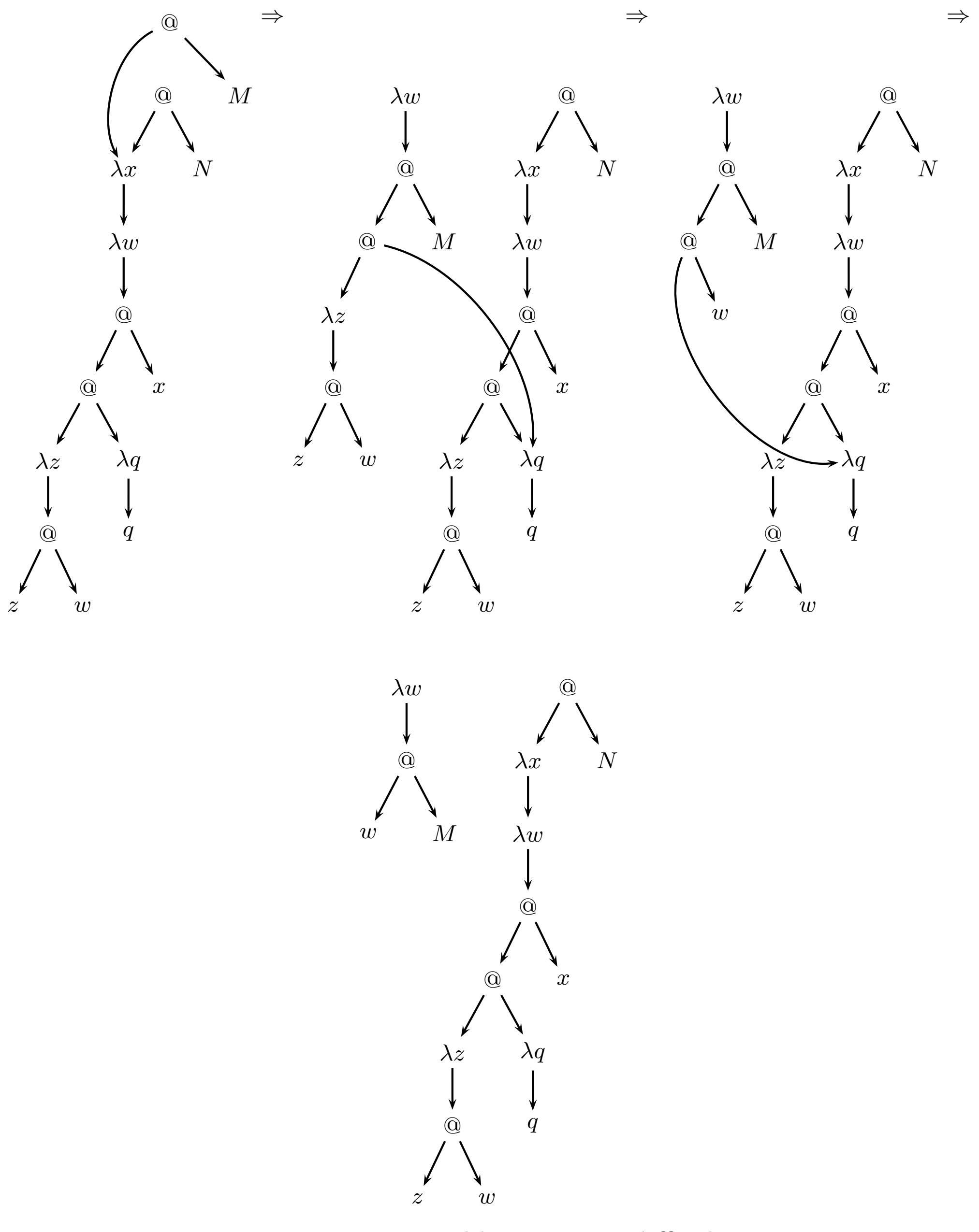

Figure 6.14: Variable renaming difficulty

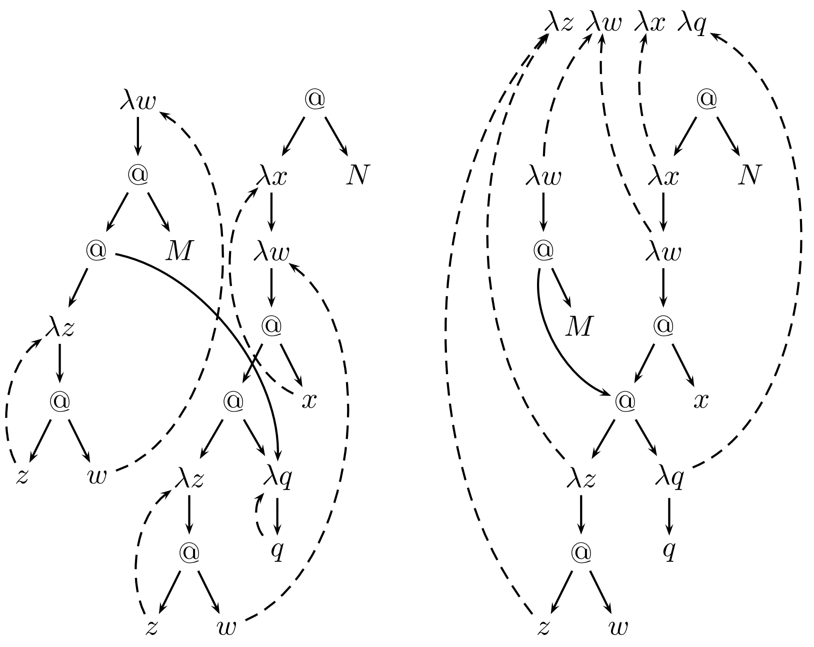

Figure 6.15: Term without and with ideal abstractions

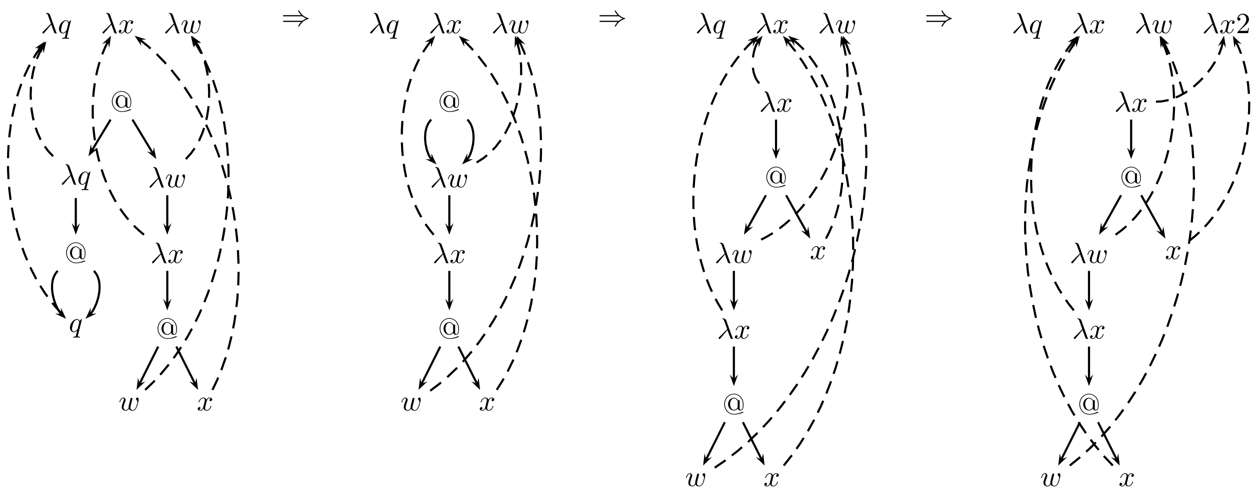

Figure 6.16: Figure 6.12 with ideal abstractions and renaming

6.4 Future Work

The results of this work were encouraging. Mogensen's reducer required 268,481 β-reductions to compute the third projection of M. Using our reducer, we could perform the same computation in only 232,621 reductions, a 13% improvement. Results like this are, of course, highly dependent on the particularities of the code being reduced. This reduction was not enough to make a significant improvement in our ability to perform the third projection. We stopped work on this front when we were able to determine that classic combinatorial explosion was the cause of non-termination for the third projection of P.

An optimal reducer, the Bologna Higher Order Machine (BHOM), has been found [3]. Nevertheless, we feel that there could still be benefit in increasing the power of this system. The BHOM reducer is optimal in the number of β-reductions, but to accomplish that many other kinds of nodes and reductions had to be introduced into the representation. We did not test our reducer alongside BHOM since the downloadable version only reduces to WHNF, and we had already discovered the cause of the third projection's non-termination. However, it would be interesting to know what the tradeoffs are between the relative simplicity of our reducer and the β-optimality of BHOM. It may be that the software metric that really matters—wall-clock time—does not favor one reducer over the other in all cases.

One of the optimizations that we could make is to find heuristics for determining when an abstraction will capture a free variable. If we could know that a variable would always be safe from capture, we would not have to rename it upon descending underneath its binding abstraction.

Another optimization is to mark trees that have no redexes in them. To find the next redex to reduce, the reducer must start at the root of the tree and use a kind of depth-first search. It is possible to check the same expression multiple times, due to a complicated initial expression, or because a term already in normal form is passed into a function. The search time could be reduced by remembering the result of previous searches.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

We have presented a method of controlling an on-line partial evaluator by means of a *strategy*, an extra parameter consisting of a function that acts as an advisor to the underlying partial evaluator. A strategy processes an application node by performing two operations. First, it selects a (potentially different) strategy to pass to the subterms, causing them to be evaluated to some degree. Second, it decides how β-reduction should be performed on that node. By introducing strategies into Mogensen's partial evaluator, we were able to control the degree to which the target programs were normalized.

The expressive power of strategies comes from the fact that they select the strategy to be used in processing a sub-term. This ability to compose strategies allows us to describe their behavior in terms of the actions taken on the tree as a whole (e.g., "expand the first n levels"). By using η-expansion, we are also able to write strategies such as *Expand Second*, which specializes a function with respect to its second argument, but not its first. A different use of η-expansion allowed us to write *Expand Breadth* n, which works by selectively telling nodes to ignore strategies given to them.

We have also shown that it is possible to use strategies in a larger language. One technique involved adding new kinds of terms to the language (e.g., arithmetic, `if`), which are not processed by strategies, but instead are handled by the underlying evaluator. The second technique, shown in chapter 5, was to expand the role of strategies to that of transformers. In addition to handling β-reduction, transformers are responsible for the other kinds of reductions.

Strategies have several limitations to their expressive ability. One limitation is that strategies are only consulted at application nodes. An abstraction node will simply pass its strategy into the sub-term. As a result, we cannot use strategies in their current form to express a computation that involves recognizing an abstraction node. An example would be a hypothetical WHNF strategy, which instructs the partial evaluator to reduce its term to WHNF. To reduce to WHNF it is necessary for a strategy to stop reducing once it has been passed below an abstraction, but currently there is no mechanism to communicate that information to the strategy. A second limitation involves the structure of a PEV. In order to examine the contents of a node in concrete form, it is necessary first to reduce it to a residual form, making that particular computation more expensive.

Another limitation we encountered occurs during self-application of the partial evaluator, due to the combinatorial explosion characteristic of the loss of binding-time information. While the Mogensen evaluator was small enough that the combinatorial explosion was manageable, the additional code required to enable the use of strategies made our partial evaluator too large for a third projection using the more aggressive strategies.

In addition to the work on strategies, we have presented a λ-calculus normalizer that, while not optimal in the number of β-reductions needed to normalize a term, is very aggressive taking opportunities to share the results of computations. To build this normalizer, we made two major improvements. First, we improved the efficiency of the standard instantiation algorithm by enabling it to detect which parts of the expression were not changed by the instantiation. This allowed more subexpressions to be shared than in the standard algorithm. Second, we changed the representation of abstraction nodes to allow them to be copied without automatically requiring the body of the abstraction to be copied as well. These changes gave us a 13% speedup over the reducer used in Mogensen's experiments.

7.2 Future Work

7.2.1 Strategies

One question raised by strategies is how they can be used in larger languages than λ-calculus. Chapter 5 details one possible answer, having strategies process abstract syntax trees directly.

More work can be done to understand the interactions between the different transformers during partial evaluation. Furthermore, the effect of partially evaluating the transformers themselves has not been explored.

Another area for future work involves the use of strategies in other domains, such as recursive descent algorithms. For example, a strategy-based search algorithm might be able to use a fast evaluation strategy to select a candidate subtree, and then switch to a slower but more precise strategy to make the final selection. The ability to compose strategies would allow us to build a new strategy quickly out of strategies we already have.

7.2.2 Reducer

There are several areas to add optimizations to the reducer. One of the optimizations that we could make is to find heuristics for determining when an abstraction will capture a free variable. If we could know that a variable would always be safe from capture, we would not have to rename it upon descending underneath its binding abstraction.

Another optimization is to mark trees that have no redexes in them. To find the next redex to reduce, the reducer must start at the root of the tree and use a kind of depth-first search. It is possible to check the same expression multiple times, due to a complicated initial expression, or because a term already in normal form is passed into a function. The search time could be reduced by remembering the result of previous searches.

We would also like to know the tradeoffs between our techniques and the ones used in the BHOM interpreter. One question is how much the increased cost of performing a β-reduction offsets the gains made by needing fewer β-reductions.

References

[1] et al Abelson. Revised^5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998.

[2] Luigia Aiello and Gianfranco Prini. An efficient interpret for the lambda calculus. *Journal of Computer and System Sciences*, 23:383–424, 1981.

[3] Andrea Asperti and Stefano Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge Tracts in Theoretical Computer Science*. Cambidge, 1998.

[4] H. P. Barendgregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[5] Mattox Beckman and Samuel N. Kamin. Controlled self-applicable on-line partial evaluation, using strategies. In *International Conference on Computer Languages*, pages 143–152, 1998.

[6] Allan Bloom. *The Republic of Plato*, chapter 7. Basic Books, 1986.

[7] Olivier Danvy. Type directed partial evaluation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 242–257, 1996.

[8] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–227, 1995.

[9] Olivier Danvy, Karoline Malmkjr, and Jens Palsberg. Eta-expansion does the trick. Technical Report RS-95-41, 1995.

[10] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, Inc, 1972.

[11] Y. Futamura. Partial evauation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[12] Robert Glück. Towards multiple self-application. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation 1991*, pages 309–320. ACM SIGPLAN, 1991.

[13] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An evaluation of staged, run-time optimizations in dyc. In *Conference on Programming Language Design and Implementation*, pages 293–304, 1999.

[14] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–541, 1997.

[15] N.D. Jones, P. Sestoft, and H. Sondergaard. An experiment in partial evaluation: The generation of a compiler generator. In Jean-Pierre Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 141–157. Springer-Verlag, May 1985.

[16] Neil D. Jones. Mix ten years later. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation 1995*, pages 24–38, 1995.

[17] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[18] Torben Æ. Mogensen. Efficient self-interpretations in lambda calculus. *Journal of Functional Programming*, 2(3):345–363, 1992.

[19] Torben Æ. Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In *Proceedings of the ACM SIGPLAN Symposium on Partial evaluation and semantics-based program manipulation*, pages 39–44. ACM Press, 1995.

[20] F. Nielson and H.R. Nielson. *Two Level Functional Languages.* Cambridge University Press, 1992.

[21] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, 1995.

[22] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[23] Simon L. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[24] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.

[25] Matthai Philipose, Craig Chambers, and Susan J. Eggers. Towards automatic construction of staged compilers. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 113–125. ACM Press, 2002.

[26] Eric Ruf. Topics in online partial evaluation. Technical Report CSL-TR-93-563, Stanford University, March 1993.

[27] Colin Runciman. Binding-time improvement and fold/unfold transformation.

[28] Robert Skeel. Roundoff error and the patriot missile. *SIAM News*, 25(4):11, 1992.

[29] J. Steensgaard-Madsen. Typed representation of objects by functions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(1):67–89, 1989.

[30] Peter Thiemann. Cogen in six lines. In *ACM SIGPLAN International Conference on Functional Programming 1996*, pages 180–189, 1996.

[31] C.P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus.* PhD thesis, Oxford, 1971.

Vita

Mattox Beckman was born in Cleveland, Ohio, in 1970, and lived in five different states before moving to Urbana in 1989. Mattox received his B.A. in Mathematics and Computer Science in December, 1992. He now lives in Chicago where he teaches at the Illinois Institute of Technology.