Towards a New Perspective on Partial Evaluation: Results, New Ideas, and Future Directions

Morry Katz*
Computer Systems Laboratory
MJH 416A
Stanford University
Stanford, CA 94305
(katz@cs.stanford.edu)

1 Introduction

This paper is an update to the ideas presented in "Towards a New Perspective on Partial Evaluation"[1]. The reader is assumed to be familiar with the previous paper, as the ideas presented therein are not repeated in this document. The major advances since the publication of that paper that will be discussed in this document fall into three categories: changes to the use lattice to improve termination, design of a lazier analysis, and addition of a new mechanism to the termination strategy based on searching for base cases of recursions.

This paper consists of three sections. The first section presents a discussion of the results obtained from implementing the RCP partial evaluator proposed in the previous paper. The next section introduces those new ideas that have allowed for the implementation of a partial evaluator that is closer to the fully lazy ideal. The final section discusses some unimplemented ideas for improving our current partial evaluator and other areas requiring further research.

*Supported in part by fellowships from NCR Corp. and Microsoft Inc.

[1] Published as FUSE-MEMO-92-12 and [1]. Available for anonymous FTP from quilty.stanford.edu.

2 Results

Our original partial evaluator operated as expected on the sample programs presented in our previous paper. Partial evaluation of `iota` applied to an unknown value for `n` terminated. Partial evaluation of `iota` applied to a known integer (e.g., `5`) resulted in symbolic execution that proceeded until the complete result had been computed, yielding a straight line program that just consed the desired output. Similarly, partial evaluation of the regular expression matcher applied to the known pattern a^* and an unknown input string produced a fully inlined decision tree with a single loop for kleene star matching. Unfortunately, this partial evaluator did not work as well as desired on some other types of programs.

Due to the RCP nature of our original partial evaluator, it was susceptible to induced divergence. In all cases, divergence resulted from overrecording the amount of information that was actually used, causing inherently equivalent application to appear to be distinct. The overrecording of use can be traced to two

causes: the inability of the chosen information lattice to record precisely the information utilized by some primitives and insufficient laziness in the use computing mechanism.

No information lattice can precisely capture every aspect of the information that is needed about a set of values in order to perform all computations. The information lattice presented in our previous paper was stated to be unable to represent exactly the information utilized by the `<` function in comparing a known integer to the number `5`. That information lattice could only record use of the integer argument's type, as $\perp_{Int}$, or its value. The actual information used in deriving the result of the computation is that the known integer is either a member of the subrange of the integers less than `5` or the subrange greater than or equal to `5`. Since subranges were not included amongst the lattice elements, either an underestimate or an overestimate of use has to be made. Since we were designing an RCP partial evaluator, overestimation was required in order to insure that no two distinct applications would ever erroneously appear equivalent. Of course, this can lead to induced divergence when two equivalent applications are annotated with non-equivalent use information. In practice, we found that overrecording of information usage due to limitations of our information lattice and consequent induced divergences happened more often than we desired. We considered patching the most common source of this problem by adding subranges to the information domain; however, the addition of arbitrary unions of subranges seemed to be required in order to solve effectively the problems resulting from integer comparisons. Furthermore, we believed that similar problems were likely to appear for other types, leading us to desire a more universal solution to the problem.

Lazy use-analysis as proposed in our previous paper proved insufficiently lazy to avoid induced divergence due to overrecording of use information. Lazy use-analysis was based on delaying asserting that information had been used until it actually contributed to making a control flow decision. The application of this concept to conditionals worked as expected. Conditionals asserted use of the boolean value of their predicates since these values are required in deciding whether control should proceed with execution of the consequent or the alternative branch of the conditional. This can be an overrecording of information usage since the information used about the result returned by a conditional does not always depend on the branch of the conditional that was executed. Induced divergence resulting from this source of overrecording of information usage was no more common than anticipated.

The unexpected source of induced divergence was due to another type of control flow decision in Scheme. Every time a function is applied, there is a control flow decision that is made when the function position of the application is evaluated and the resultant closure is applied to the arguments. The code that is executed after the application is dependent on the body of the closure selected. The result generated by the closure application may also depend on the values over which the function body has been closed. The lazy use-analysis rule that control flow decisions should immediately assert information usage implies that symbolic execution should assert that it has used the information contained in a closure each time one is applied. This presents a serious problem for a language like Scheme with first-class, higher-order functions. Code written in a continuation passing style (CPS) explicitly passes a continuation function to each closure. When this function is applied, it is used in making a control flow decision, so use of the identity of the closure is immediately asserted. The result is that for almost any CPS program, the continuation arguments of

no two function applications will ever appear to be equivalent. This means that partial evaluation of almost all CPS programs using the lazy use-analysis based partial evaluator presented in our previous paper result in induced divergences.

Part of the solution to apparent lack of equivalence amongst closure uses is to redefine the information lattice elements used to represent uses of closures. The function domain for Scheme values in Figure 4 of our previous paper should be replaced by a closure domain defined as $Clos = Code \times Sval^{\star}$. This definition captures that a closure is the conjunction of a piece of code, a lambda expression, and a set of lexical values over which the lambda expression is closed. A corresponding domain of values for representing closures that have been applied, which should replace the function domain in the information lattice in Figure 5 of that paper, is $Clos = Code \times PEval^{\star}$. This domain represents that uses of a closure can include uses of its code and uses of the values over which it is closed and the arguments to which it is applied. Two applications of different closures of the same lambda expression are equivalent with respect use if equivalent uses are made of the values over which they are closed and the arguments to which they are applied. The domains presented in our previous paper could not capture this equivalence.

Unfortunately, the improved closure domains suggested do not solve the problem of induced divergence when performing partial evaluation of CPS programs using lazy-use analysis. The continuation arguments that are passed from function to function essentially encode the set of stack frames that would have been present at the time of the application if the program had been written in a direct style. Each closure representing a continuation is composed of a piece of code to be executed and is closed over a value (a closure) representing the rest of the continuation. Essentially, each closure acts as a single stack frame that contains in its environment another closure representing the next stack frame, etc. As a continuation is executed, it successively pops frames from the pseudo stack and executes them one by one. Each time a frame is executed, the corresponding closure is applied causing use of its code to be immediately asserted. As successive frames are executed, use of the code of the current frame implies use of the code portion of the continuation over which a previous frame was closed. At any point in time, the use profile of a given closure representing a continuation will include use of all the frames of the stack that have been used so far.

The problem arises because the continuations for two successive non-tail recursive iterations of a recursion will always have continuations that differ. The later iteration's continuation always includes a piece of code to complete the former iteration before executing the portion of the continuations they have in common. When the uses of these two continuations are compared, they will always differ because execution of the tails of the recursive procedures have always made use of the differences in their two continuations (i.e., the differences in their effective stacks). The only apparent solution to the problem of partial evaluation of CPS code was to become lazier in recording use of information about closures.

3 New Ideas

3.1 A Lazier Form of Analysis

3.1.1 Previous Problem

The goal of fully lazy use-analysis is to determine that information that contributes to the return result of a program. Lazy use-analysis builds use dependencies for all operations other than those that cause control flow

```
(define iota
  (lambda (n)
    (define loop
      (lambda (i)
        (if (> i n)
            '()
            (cons
             i
             (loop (1+ i))))))
    (loop 1)))
```

Figure 1: Iota function

changes. In lazy use-analysis, control flow operators are the only source of use assertions. The asserted use information propagates over the use dependencies in order to create use categorizations for various values in the system. If control flow operators are no longer going to make use assertions, what should be the source of use assertions?

The obvious answer appears to be that use should be asserted about the results that are returned by a program. This seems to capture the spirit of the stated goal of a fully lazy analysis. The simplest solution would appear to be to build a partial evaluator that simply asserts use of all aspects of each result that is returned by a program. This solution fails for many simple and important programs.

The iota program of our previous paper (corrected to correspond to the APL definition of this function as shown in figure 1) can potentially return any of the following results: `()`, `(1)`, `(1 2)`, `(1 2 3)`, ... This family of return values forms a sort of stream. It is programs whose possible return values form this type of stream that cause problems for a lazier analysis that asserts use of all aspects of each return value. Lets consider what happens during partial evaluation of `iota` on an unknown integer. `Loop` is initially called on the value `1`. Since the predicate of the conditional is undecidable due to `n` being an unknown integer, both the consequent and the alternative will be evaluated. The consequent returns `()` as a result from the program. The alternative applies `loop` to `2`. The predicate is again undecidable for this iteration. The consequent returns `()` from this iteration of `loop`, causing the previous iteration to return `(1)`. The alternative of the second iteration then applies `loop` to `3`. This is a second recursive call to `loop` so it must be determined whether the argument vectors of the two iterations are used equivalently in order to determine whether symbolic execution should be continued, or whether a fixed point has been reached.

Since this algorithm is lazy, use values can change over time, invalidating previous comparisons. Leaving aside the complexities of invalidated use comparisons and the possible need to restart symbolic execution, lets consider the simplest order of operations that manifest the problem with this algorithm. Assume that as soon as a result is returned from the program, full use of every aspect of that value is immediately asserted and the results of those assertions are propagated over all use dependencies. In other words, As soon as `(1)` is returned, it is asserted that use has been made of the cons cell, the identity of its car, `1`, and the identity of its cdr, `()`. The use of the identity of `1` implies use of the oneness of `i` in the first iteration of `loop`. This use is incompatible with the lack of use of `i`'s value of `2` in the second iteration and the value `3` passed to the third iteration. As a result, symbolic execution continues. The third iteration initiates the return of `(1 2)` from the program. Asserting complete use of this value will assert use of the twoness of `i` in the second iteration, preventing symbolic execution from terminating when the third recursive call is made to `loop`. Symbolic execution diverges as every possible

member of the infinite stream of return values is computed.

Clearly, something other than asserting use of all aspects of every return result is needed for an effective fully lazy use-analysis. At the time our previous paper was written, we settled on lazy use-analysis as a seemingly workable, albeit less than perfect approximation to the desired ideal. Since that time, we believe that improved insight into the problem has enabled us to develop something close to a fully lazy use-analysis.

3.1.2 The Apparent Solution

Asserting complete use of all aspects of every possible return result from a program implies more utilization of these values than may actually take place. As has been discussed in the past for return values of functions, it is often the case that only certain aspects of the information contained in a value will be utilized. For a function call it is possible to determine what information was, or could be, used based on the context from which it was called. For a program, there is no such context. However, it is evident that asserting that all aspects of the potential return results are needed is an overstatement of use that can result in divergence. We propose that what must be asserted is that the return values are needed, but their actual values are unimportant. In other words, the computations that produce the return values must be performed, but no specific use need be asserted about idiosyncratic characteristics of those values.

Use-analysis based on this philosophy can be implemented by adding to the information lattice one additional value. In the new set of domains shown in figure 2, $\perp$ represents an unused value and $\perp_{PEval}$ represents a value that must be computed, but whose precise value is unimportant. (It should be noted that $\perp$ has effectively taken the place of the old definition of $\perp_{PEval}$ and that $\perp_{PEval}$ has taken on the meaning "this value is needed".) The information lattice in figure 3 shows that there is more information content in needing a value than having it be completely unused. (It also contains some relations amongst closure values that were not present for the function domains of the previous paper.)

Use dependences involving the new use annotation are fairly straight forward. Needing the result of a computation implies needing the values used to compute the result. For example, the computation `(+ 1 2)` would create use dependences linking needing the result, `3`, to needing both `1` and `2`. Function applications also form a use dependence between needing their result and needing the code of the applied closure. This represents that the result of a function application depends on the function that has been applied. For the example of `(+ 1 2)`, a use dependence would be formed between `3` and the closure bound to the symbol `+`. Finally, use dependences are formed between needing the value returned by a conditional whose predicate is decidable and the boolean value of the predicate.

Using this new analysis for symbolic execution of `loop` applied to an unknown integer differs primarily in what happens when use is asserted of the return values. After 3 iterations of `loop` have been executed, the values `()`, `(1)`, and `(1 2)` have all been returned. Asserting the need to compute all these values, asserts the need to compute the `1`, the `2`, and the cons cells. Asserting need for computing the integers asserts the need for the values associated with the parameter `i` in both the second and third iterations of the recursion. Since these two uses are equivalent, symbolic execution is terminated when the third recursive call to `loop` (the one applied to the argument `3`) is performed during the third iteration. Divergence no longer results for stream style computations like iota.

Int	$=$	$\mathtt{0}, \pm\mathtt{1}, \pm\mathtt{2}, \cdots$	integers
		$\bot_{Int}$	unspecified integer
$Bool$	$=$	$\mathtt{true} + \mathtt{false}$	booleans
		$\bot_{Bool}$	unspecified boolean
Nil	$=$	$\mathtt{nil}$	empty list
$Pair$	$=$	$PEval \times PEval$	pairs
		$\bot_{Pair} = \bot \times \bot$	unspecified pair
$Clos$	$=$	$Code \times PEval^{\star}$	closure values
		$\bot_{Clos}$	unspecified closure
$Kval$	$=$	$Int + Bool + Nil + Pair + Clos$	known values
$Bots$	$=$	$\bot_{Int} + \bot_{Bool} + \bot_{Pair} + \bot_{Clos}$	bottom values
$PEval$	$=$	$Kval + Bots + \bot_{PEval} + \bot$	partial evaluation values
		$\bot_{PEval}$	unspecified (needed)value
		$\bot$	unused value

Figure 2: Value domains for information

$\bot \prec \bot_{PEval}$
$\forall x \in (Bots \cup Nil).(\bot_{PEval} \prec x)$
$\forall i \in Int.(\bot_{Int} \prec i)$
$\forall b \in Bool.(\bot_{Bool} \prec b)$
$\forall c \in Clos.(\bot_{Clos} \prec c)$
$\forall < code, s > \in Clos.(\forall s_{i'} \prec s_i.(\forall j \neq i \Rightarrow s_{j'} = s_j.(< code, s' > \prec\prec < code, s >)))$
$\forall < x, y > \in Pair.(\forall x' \prec x.(< x', y > \prec\prec < x, y >))$
$\forall < x, y > \in Pair.(\forall y' \prec y.(< x, y' > \prec\prec < x, y >))$
$\forall k \in (Kval \cup Bots).(k \prec \top)$

Figure 3: Information lattice description

3.2 Computing Return Values

When a recursion is terminated due to use equivalence, some value must be returned as the result of that recursion. Most partial evaluators have chosen the simple solution of returning a completely unspecified value. Ruf in [2] proposed computing a more accurate return value through computation of a fixed point over the generalization of all values returned by the recursively called function. This fixed point is computed by initially returning an unspecified, top value, for the result of the recursive calls. As other values are returned from the function, these are generalized with the current generalized return value to generate a new generalized return value. If this value differs from the old generalization, the new generalized return value is injected as the return value at each of the recursive call sites and computation proceeds using the new generalization. Eventually, this process reaches a fixed point, yielding a valid generalized return value.

We have adopted a method based on Ruf's for computing our return values. In our approach, when symbolic execution of a recursion is terminated, the analysis is continued by initiating symbolic execution of the application of the code of the two use equivalent applications to a generalization of the two equivalent argument vectors. Symbolic execution proceeds as usual from this special application until a recursion is terminated with the special application being one of the two equivalent applications. At this point, the generalized return value of the special application is returned as the result of the terminated recursion and symbolic execution of the continuation of the original recursive function call proceeds. If the generalized return value changes at some point due to a contribution from another return value, all the computations performed in the continuation of the original recursive function application must be thrown away and symbolic execution of this continuation must be reinitiated using the new generalized return value.

This approach is very similar to Ruf's. The major difference is that symbolic execution of the continuation of a recursive call to a function is allowed to proceed possibly before the calculation of the generalized return value has reached a fixed point. When it is found that an incorrect generalized return value has been used, all computations that potentially utilized that value must be repeated using the correct generalized return value. We have selected this approach because it allows us to use our existing symbolic execution mechanism for computing the generalized return values, rather than having to implement a completely new mechanism for this task.

The special application to the generalized argument vector was introduced for two reasons. First, the two equivalent iterations that terminated the original recursion might not investigate all the control flow paths investigate by the application to the generalized arguments. This means that utilization of those two iterations to compute the generalized return value might result in an incorrect generalized return value. Second, it is important that the use dependences created for computing the generalized return value not effect a determination of whether the original two iterations are use equivalent. Utilization of the original two iterations for computation of the generalized return value could lead to two equivalent iterations appearing to be distinct. Finally, it should be noted that the residual code resulting from the computation of the generalized return value is precisely the code that should be used for the residual recursive definition.

3.3 Improving Termination

Two sources of induced divergence remain in a RCP partial evaluator using fully lazy use-analysis and including computation of generalized return values. The first is divergence due to inability of the use annotations available from a specific information lattice to reflect precisely that information that is used in performing the delta reductions of individual primitives (e.g., `>`). The second is divergence due to failure of the analysis scheme to capture certain types of equivalence. For example, the result returned by both branches of a conditional may always be guaranteed to be equivalent so that the return result does not really depend on the value of the predicate. In general, a property like the above may be unprovable, even if it is true. Similarly, a recursion might be formed by passing through two conditionals whose predicates are guaranteed to always return the same result. If the values of the predicates are unknown during partial evaluation, execution paths will be investigated that could never take place in a runtime execution of the program. Symbolic execution of this type of impossible execution path might diverge, leading to an induced divergence. Again, in general proving code equivalence is impossible, so this type of induced divergence is an inherent liability of the lazy use-analysis based RCP partial evaluation scheme that we chose to investigate.

It is possible to move in the direction of TP partial evaluation by removing the first class of induced divergences. This can be done by changing the way use is approximated for primitives such as `>` so that whenever it is impossible to precisely reflect the information that is used in performing a delta reduction, an underestimate, rather than an overestimate, of use is utilized. The resultant partial evaluator is neither TP nor RCP, rather a hybrid of the two. It continues to diverge on some programs due to the second cause of induced divergence explained above. It fails to produce as good residual code as many RCP partial evaluators because of premature termination.

For example, partial evaluation of `iota` applied to a known integer like `5` will fail to produce the residual code that just conses together the desired list of numbers. It will instead result in a recursive program to compute the list. This is because the application of `>` only records use of the integer property of the constant value of `i` in each iteration, the correct underestimate of use. This makes the return values of the different iterations of the loop all dependent on only the integer property of the loop parameter, and therefore equivalent.

We have implemented a prototype of the hybrid TP/RCP partial evaluator described above. It is currently operational and works as expected on many simple programs. Some unexplained behavior has been observed, and it remains to be determined whether it results from unanticipated or misunderstood properties of the analysis algorithms, or bugs in the implementation.

4 Work In Progress

4.1 Improving Residual Code

We believe that we have a good method for improving the quality of the residual code produced by our hybrid TP/RCP partial evaluator. It is based on adding a secondary consideration to our termination algorithm that will cause certain recursions to reinitiate symbolic execution of a recursion despite having detected two use equivalent iterations. The concept behind the additional mechanism is that every recursion must have a base case that returns some value in order to terminate. If symbolic execution of all execution paths starting

with the first call to a recursive function have never returned a value, but have instead been terminated due to reaching an equivalent recursive call, then no base case has been found. Assuming the input program does not diverge, one of the recursive calls that has not been performed must lead to the base case and return a value. Without risking induced divergence, it is safe to force symbolic execution of the terminated recursions until at least one base case is reached. a For a simple program like `iota`, there is only a single recursive call in the body of the recursive function. This means it is simple to understand how application of the above mechanism would operate. Partial evaluation of `iota` applied to `5` would progress until two iterations of `loop` had been performed. At this point, the recursion would be terminated, a generalized return value would be computed, the tails of the loops would be executed, and the result (`1 2` $\perp_{Int}$ $\perp_{Int}$. $\perp_{PEval}$) would be returned. Asserting $\perp_{PEval}$ use of this return value would not effect the equivalence of the two iterations of `loop`, so the analysis without the new rule would be concluded. However, no base case for `loop` would have been reached. As a result, the recursion of `loop` would be forced to start one more iteration. This iteration would make another recursive call to `loop` that would be found to be equivalent to an earlier one, leading to a second termination of the recursion. A generalized return value would again be computed, the tails of all iterations of `loop` would be executed, and a new value would be returned from `iota`. Assertion of $\perp_{PEval}$ use of this new return value also would not effect the equivalences that led to termination. However, a base case for `loop` would still not have been reached. Consequently, the entire process would be repeated by forcing symbolic iteration of another iteration of `loop`. This process would continue until the final iteration of `iota` applied to `5` was performed. This iteration is the base case. The final recursive call returns `()` and the entire program returns `(1 2 3 4 5)`. At this point the optimal residual program for `iota` applied to `5` has been produced.

```
(define (fact n)
  (if (zero? n)
      1
      (mult 1 n)))

(define (mult i j)
  (if (= i j)
      i
      (let ((mid
             (truncate
              (/ (+ i j) 2))))
        (* (mult i mid)
           (mult (1+ mid) j)))))
```

Figure 4: Factorial function

The operation of our new mechanism is slightly more complex for a program with multiple recursive calls. In the factorial program shown in figure 4, `mult` is called recursively from two points in its body. Partial evaluation of `fact` applied to `100` would lead to four different recursive calls to `mult` all being terminated due to use equivalence[2]. Generalized return values would be computed for all the calls to `mult` and symbolic execution would proceed with execution of the function tails of all four flows of control. In this case, each of the four control flows returns a value from `fact`, although they are all identical. Assertion of use of these return results does not invalidate any of the equivalence decisions our new mechanism comes into play. As with `iota`, no base case for `mult` has been reached, so each ter-

[2] The first iteration makes two recursive calls to `mult`. Each of these iterations makes an additional two recursive calls to `mult`. Each of the recursions is terminated because all four flows of control have passed through two use equivalent iterations.

minated recursion is forced to start one more iteration of its recursion. It is important to note that all recursions of `mult` are forced to start one more iteration. No recursion that again terminates after being forced to execute the one additional iteration will be forced to start another iteration unless all of the recursions again terminate without producing a base case. It is this condition that assures that our mechanism will not add a new source of divergence in searching for base cases. In conclusion, our new mechanism will force partial evaluation of `fact` applied to a known integer to proceed until the correct return value is computed and produce the residual program that just returns this value. In the absence of our new mechanism, our partial evaluator would yield a recursive residual program for calculating that result.

Our new mechanism is similar in its motivation to the undecidable conditional mechanism used in the termination algorithm of Fuse [3]. Fuse only terminated partial evaluation of a recursion when a recursion function application was reached and the recursive call spanned a conditional whose predicate was undecidable. The intention there was that a runtime evaluator can only terminate a recursion by reaching a base case. If control flow has not bifurcated somewhere between an original call and a recursive call, then a runtime evaluator that reached the first call would by definition have to execute the recursive call as well. Only if control has passed through an undecidable conditional is it possible that the branch of the conditional containing the recursive call would not be executed by a runtime evaluator, control instead having passed to the other branch of the conditional and leading to the base case.

The problem with the undecidable conditional rule is that the converse is not true. Just because a recursion spans an undecidable conditional does not mean that a base case has been reached through another flow of control. For example, hand CPSed code never returns from any function calls, so many undecidable conditionals separate two iterations of a recursions even though these conditionals would not separate the two iterations in the same program written in a direct style. This means that the undecidable conditional rule does not work effectively for much code written in CPS. Also, programs in which both branches of an undecidable conditional make a recursive call will pass the undecidable conditional test even though no base case has been reached. Our mechanism that seeks to find base cases as opposed to the control operators that might lead to base cases promises to perform better than Fuse on several classes of programs.

We believe addition of the base case rule to the termination mechanism used by our existing hybrid TP/RCP partial evaluator will yield a partial evaluator that has both excellent termination properties and produces very high quality residual code. We are currently discussing different methods for implementing the mechanism. Many of the methods are correct, but appear to be too inefficient to be of practical use. The leading candidate, which is a lazy algorithm for computing the property of whether a base case has been reached, has not been completely developed, but appears to be very promising. We anticipate completing the design of this algorithm and integrating it into our current partial evaluator in the near future.

4.2 Improving Efficiency

Our current prototype partial evaluator was implemented in the most straightforward manner possible in order to maximize the speed of implementation and minimize the probablility of introducing bugs into the use-analysis algorithms. Unfortunately, the resulting prototype is only usable on very small and fairly simple programs due to its large memory consump-

tion. One of our next goals is to improve the efficiency of this implementation somewhat so that we can partial evaluate somewhat larger and more complex programs. We anticipate this requiring the use of more memoization in the implementation in order to avoid repeated symbolic execution of equivalent threads of control. This should not only improve the space requirements of the partial evaluator, but might very well improve its speed as well.

A production version of a lazy use-analysis based partial evaluator would undoubtedly require much more sophisticated implementation techniques than we intend to investigate at this time. In particular, it would most likely be desirable to delay symbolic execution of given execution paths on different sets of data until the propagation of use information indicates that the results of the different executions are going to be used in a nonequivalent manner, mandating separate analysis. In essence this means that efficiency can be gained by sharing pieces of the analysis temporarily, and only splitting analysis of single continuations on multiple values once it is demonstrated that this is needed. Such an implementation is much more complex to program and we are avoiding it for the time being because of the high risk of introducing errors into the analysis due to programming bugs. We have not yet developed sufficient insight into how the analysis works in order to be able to easily determine when unexpected results of the analysis are due to implementation bugs, as opposed to unexpected properties of the algorithms themselves.

4.3 Residual Code Generation

To date our prototype lazy use-analysis based partial evaluator does not generate any residual code. It performs the lazy use-analysis necessary to determine what specializations should be created, but does not generate them. Once we have been successful in finding an analysis that has the combination of termination and residual code quality properties that we desire, we will need to add a code generator to the back end of our prototype partial evaluator. At the present time, we anticipate adding a very simple residual code generator that just produces Scheme code. It will not perform any additional optimizations beyond those that are obviously called for based on the use-analysis.

5 Conclusions

In this paper we have presented the results of implementing the RCP partial evaluator proposed in our previous paper. Based on those results, we went on to investigate changes that could be made in our use domains in order to improve the termination properties of our partial evaluator, but came to the eventual conclusion that a lazier form of analysis would be needed to do the job effectively. We have presented the design of that lazier analysis along with a list of it negative and positive properties as found from a prototype implementation. Finally, we have proposed a secondary termination mechanism that searches for base cases of recursions that we have used to create a hybrid TP/RCP partial evaluator. This hybrid partial evaluator promises to produce high qulity residual code and display good termination properties.

References

[1] Morry Katz and Daniel Weise. Towards a new perspective on partial evaluation. In *Proceedings of the 1992 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipula-*

tion, pages 29–37, San Francisco, CA, June 1992.

[2] Erik Ruf and Daniel Weise. Improving the accuracy of higher-order specialization useing control flow analysis. In *Proceedings of the 1992 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 67–74, San Francisco, CA, June 1992.

[3] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online program specialization. In *Fifth International Conference on Functional Programming Languages and Computer Architecture*, pages 165–191. Springer-Verlag Lecture Notes in Computer Science, August 1991.