POPE: An On-line Partial Evaluator

Peter Ørbæk*
poe@daimi.aau.dk

June 24, 1994

Abstract

In the past, most partial evaluators have been geared towards purely functional and logic languages with no or little support for side-effects and mutable data. Many programs, however, use such imperative features for efficiency or for ease of expression, thus partial evaluators should be able to specialize imperative programs as well. The partial evaluator described in this paper attempts to cope with side-effecting procedures and mutable data while still providing for polyvariant higher-order functions.

1 Introduction

During some experimentation with the partial evaluator Schism [Con88, Con94], trying to specialize an interpreter for a subset of Action Notation [BP93, Mos92], it became clear that Schism was not up to the task and was spending way too much time doing polyvariant binding-time analysis (on the order of hours on a SparcStation 20!).

I therefore decided to try to make an online partial evaluator [Ruf93] for Scheme [CR+91b] that should be able to handle the kind of polyvariance needed for the action interpreter, plus be able to cope with side-effecting functions and mutable data, something that Schism and many other partial evaluators do not handle well.

The focus was on making a flexible partial evaluator sacrificing efficiency both in the partial evaluation phase as well as in the run-time phase. No provisions for self-application are made whatsoever.

Another reason for writing POPE (Peter's Online Partial Evaluator) was that Schism runs on T and the commercial Chez-Scheme™ compiler, whereas POPE runs on the freely available `scm` interpreter and thus it will even run on a PC[1].

2 Discussion

Why did Schism spend so much time in the binding-time phase? The main interpreter function in the action interpreter has 11 arguments, where only one can be expected to be

*BRICS, Center of the DNRF, Aarhus University, Denmark

[1]The scheme code is available via anonymous ftp in `ftp://ftp.daimi.aau.dk/pub/empl/poe/`

everywhere static. The rest of the arguments may occur both statically and dynamically. Because of the *polyvariant* BTA that essentially computes separate binding-times for each combination of static and dynamic arguments, there are 2^{10} combinations just for this one function.

A naïvely coded BTA would traverse the function body once for each combination of binding-times. I don't know how Schism is implemented, but it seems that even with Schism we get the exponential blow-up in BTA time.

In the on-line approach to partial evaluation, one does not typically employ a binding-time analysis at all, and functions are only considered in the context of their actual application, ie. not all possible combinations of binding-times for the arguments need to be considered.

The drawback of the on-line approach is that specialization time may be longer than in an off-line partial evaluator, due to the precomputation a BTA may do.

Another option would be to employ a *monovariant* off-line partial evaluator like Similix [Bon90, BP93], where the BTA only considers the body of functions with respect to the most conservative (dynamic) binding-times for the arguments. However, experience shows that in many cases, polyvariance is a necessity in order to achieve good specialized programs.

Both kinds of off-line techniques achieve extra specialization speed by being less accurate with respect to binding-times than an on-line specializer can. Since no actual values are present in the binding-time analysis phase, a conservative approach need to be taken for `if` statements even in the case of a static condition. In an on-line specializer the value of the condition will be known at specialization time, and only the taken branch of the `if` need to be considered.

For all the deficiencies of the off-line approach it should be noted that good self-application of an off-line partial evaluator is more easily achieved than with an on-line partial evaluator. For this work self-application was a non-issue, thus an on-line approach was taken.

3 An example

As an introduction to POPE I'll give the traditional `append` example. Given the definition of a list appender function:

```
(define (myappend x y)
  (if (null? x)
      y
      (cons (car x) (myappend (cdr x) y))))
```

we wish to partially evaluate this function with respect to a static first argument, say `'(1 2 3)`. In POPE the process would look like (assuming the function definition is on the file "`append.scm`"):

```
% pope
> (pe-file "append.scm" "append-res.scm" '(lambda (w) (myappend '(1 2 3) w)))
; reading append.scm...
; mutation analysis...
```

```
; parsing expression...
; partially evaluating...
; residual PE'ing...
; unparsing...
; writing cache...
; writing residual program
>
```

The residual program is placed on the file "`append-res.scm`", and looks like this:

```
(define pope:main
  (lambda (w) (cons '1 (cons '2 (cons '3 w)))))
```

4 To filter or not to filter

In order to control and avoid infinite unfoldings of recursive calls Schism provides the notion of a *filter*. Filters are attached to `lambda` constructs, and control the propagation of binding-time information, as well as it is possible to conditionally unfold or residualize a specialized version of the function. This is a very powerful feature for an off-line partial evaluator, but a little awkward to use in my opinion. Often it is the case that one knows which *call* makes the partial evaluator or binding-time analyzer unfold too aggressively. With filters one has to move the call to an auxiliary function making sure to get all the necessary arguments included, and put a filter on the auxiliary function. With the DYN construct or a `generalize` construct as in Similix one can add information to the call locally that will prevent the partial evaluator from unfolding too much.

Needless to say, the approach taken in POPE is to add information to the call, not the function definition. There are three constructs for this:

- Wrapping an expression, usually a variable, in `(DYN expr)`. This will completely prevent POPE from looking at `expr`, and treat the result as dynamic as well as always residualizing the body expression.

- Transforming a call `(func expr...)` into `(CACHE` *flag* `func expr...)`. This will make POPE put a specialized version of the function into its cache, and that specialized function will appear in the residual code. This can be used to avoid infinite unfoldings of recursive calls that should really be residualized. If the *flag* is `#tt` then the call will be unfolded provided all arguments are statically known. Otherwise the call will always be made into a call to the residualized function.

- Transforming an expression into `(generalize` *expression*`)`. This will make POPE partially evaluate the expression but leave the call as residual code, and treat the result as dynamic. At run-time `generalize` will be the identity macro.

In the case of infinite (or near infinite `:-)`) unfoldings (in the absence of appropriate CACHE directives), POPE optimistically assumes the presence at run-time of the original program, and in some cases will generate calls to the original functions instead of producing a residual function with all parameters assumed dynamic.

5 Mutable data

One of the problems with Schism was that it was unable to handle mutable data and side-effecting functions, so one of the goals was to be able to handle such things. Since a partial evaluator may not execute the code in the same order as a true evaluator would, it is mandatory to avoid changing the order of evaluation of expressions that may side-effect or depend of mutable data.

POPE knows about the special forms `set!` and `set-vector!`, and the first argument to these special forms will be left unevaluated by POPE. POPE also knows about all the input/output functions defined in [CR+91b], and will always residualize such calls.

Calls to the special functions `set-car!` and `set-cdr!` are allowed but will produce a warning, as the mutation analysis is not able to guarantee safe handling of them. Consider the following scenario:

```
(letrec ((f (lambda (x) (begin (set-car! (cdr x) 7) x))))
  (f (cons 1 (cons 2 3))))
```

The result ought to be `'(1 7 . 3)`, but the partial evaluator will specialize the function with respect to the static argument, and will propagate the argument in the function body thus giving the following in the first instance:

```
(letrec ((f (lambda (x) (begin (set-car! (cdr x) '7) x))))
  (let* ((pope:*foo*
          (begin (set-car! '(2 . 3) '7) '(1 2 . 3))))
    '(1 2 . 3)))
```

which is clearly wrong. The mutation analysis is only capable of dealing with mutable *variables*, not mutable structures.

POPE also has special code for `car, cdr, cons, list, list?, pair?, equal?, or, and, length, null?, begin` and `apply`, mostly in order to take advantage of the partially static data while safely handling mutable variables.

In order to deal with the "no reserved words" property of Scheme, the above special-cased functions are first looked up in the environment in order to allow overloading them.

When an application of an unknown but named function is encountered, the function will be `eval`'ed provided all the arguments are wholly static and can be transformed to native Scheme representation. Specifically, lambda-abstractions can not be transformed in this fashion.

6 Partially static data

POPE is able to deal with partially static structures, ie. `cons`'es of both static and dynamic data. For instance assuming `d` dynamic, the structure `(cons 1 (cons (+ 3 d) 5))` would be treated as partially static. The advantages of this are obvious, as for example the `car` of the above structure may be computed statically.

Problems arise with the combination of partially static data and mutable variables, as expressions inside a partially static structure may contain side-effecting procedure

calls, thus a partially static structure may not be indiscriminately inlined in an unfolded procedure body as that might lead to duplicate execution of the side-effecting code.

Also, taking the static `car` of a partially static structure, may forget some side-effecting computation in the `cdr` of the original structure.

The simplest way to circumvent these problems is to never inline partially static data, and only inline wholly static data. This was the approach taken by the first versions of POPE. For some tasks this yields reasonable results, but better results can be achieved in the case where the dynamic parts of a partially static structure is made up entirely of immutable variables, as such a structure may be safely inlined. The mutation analysis run before partial evaluation marks all variables that may conceivably be mutated during program execution.

7 Exposed binds

Suppose we have the following piece of code:

```
(+ 67
   (let ((x 2))
     (cdr (cons (begin (set! x 7) x) 45))))
```

The `cdr` can be computed statically, but we cannot afford to lose the dynamic mutation of the `x` variable. One option would be to residualize the entire piece of code, and label the result dynamic, but this loses the statically known result. POPE transforms the above code to

```
(let* ((x.24 (generalize '2))
       (pope:*foo* (begin (set! x.24 '7) x.24)))
  '112)
```

The `let*` construct is internally called an exposed bind. Since the order of parameter evaluation is left unspecified in the Scheme standard it is safe to move the application of + inside the exposed bind construct. As variables are renamed for each `let` and each inlined application, applications of user-defined functions can be moved inside exposed binds as well.

As there is no detection of dead variables, POPE cannot see that the value of the mutated variable is never used, and thus it can't eliminate the `let*` entirely.

Care is taken not to convert a `let` around a `lambda` abstraction into an exposed bind, as such a conversion might cause variables captured by the `lambda` to sift out of their static scope.

8 Inner workings

POPE treats the input program and static data in several stages. First the program is read from one or more files and macro expanded by way of the "macros that work" package by Clinger and Rees [CR91a] (a part of SLIB). The read definitions and the given start-expression are then parsed into an internal de-sugared representation.

Just after macro-expansion and parsing, POPE performs a mutation analysis on the program to be partially evaluated. The mutation analysis runs in two phases: First all identifiers are associated with a globally unique number (as the program is not α-converted), calls to `set!` and `set-vector!` are recognized and the number of the mutated variable is associated with a bit indicating the mutation.

The second phase transforms the program such that all occurrences of a possibly mutated variable are marked to this effect. During specialization all such marked variables are treated as holding dynamic data.

After mutation analysis, the partial evaluator proper is invoked. The partial evaluator takes advantage of an a function-call cache. Calls marked with the `(CACHE flag func ...)` construct are cached, keyed by the function definition and the static parameters. When a matching call is encountered, a call to a suitably specialized version of the function is reconstructed instead of unfolding the call. This is used to avoid unfolding the (usually few) calls that should not be unfolded at specialization time.

Theoretically one could construct a cache entry for each and every call encountered, and use some heuristic or some form of recursion detection to decide when the call should be unfolded. This approach was not taken since the cache in such a case was judged to grow too big, and by far the most calls to statically known functions should be unfolded anyway, and thus a cache entry for such calls would be unneeded.

Calls to globally defined functions are recognized, such that the residualized, specialized versions of such functions can be made global. Non-global functions that are residualized will be put in their proper scope via a `letrec`. This may unfortunately result in more than one version of essentially the same function in several places. No attempt is made to avoid this.

After partial evaluation an internal "value" is obtained. A codification procedure is invoked to produce the appropriate residual code from this value. This may in turn invoke the partial evaluator on bodies of residual `lambda` expressions. In the proper partial evaluation process, the bodies on non-applied `lambda` constructs are not (pre-)specialized.

The codification process leaves an internal representation of the residual program. A final unparsing phase is invoked to write proper Scheme code to a file. The cached residual functions are written to the file as well. In fact, the unparser may invoke the codifier which may in turn invoke the partial evaluator in order to deal with the values bound in the internal "exposed bind" construct.

9 Manual

The partial evaluator is started from the Unix command line by the command:

```
pope
```

Once the partial evaluator is loaded, one may run the primary function, `pe-file`, which can be called in one of two ways:

```
(pe-file <schemefile> <residualfile> <s-exp>)
(pe-file '(<schemefile1> <schemefile2>....) <residualfile> <s-exp>)
```

depending on whether the program consists of one or more files.

The definitions in the *schemefiles* will be read and macro-expanded, and the *s-exp* will will be partially evaluated in the scope of those definitions. The residual program will be placed on the *residualfile*.

For small tests, one may also use the form:

```
(pe <expression>)
```

to partially evaluate a single expression.

There are a couple of global variables that one can tweak (with `set!`) to improve performance. `*call-depth*` limits the number of recursive function calls that POPE will perform. This is used to avoid infinite loops. Initially it is 100. `*speculation-depth*` limits the number of recursive speculative evaluations that POPE will perform. Speculative evaluation occurs when POPE meets a conditional expression with a dynamic test. Initially this is 3.

10 Results

POPE has been used to specialize a number of small examples as well as the action notation interpreter and a pattern matcher for regular expressions.

As there was no emphasis on performance of the PE process, and since `scm` is an interpreter, the partial evaluation process may take some time, depending on how one configures the `*call-depth*` and `*speculation-depth*` variables. Specializing the Action Notation interpreter (a 500 line Scheme program) with respect to a relatively small action with a single loop took about 45 seconds on a SparcStation 20, and required about 15 million `cons` operations.

A particular example: Taking a straightforward interpreter for a `while` language using side-effects to implement the store, and specializing it with respect to Euclid's algorithm for finding the greatest common divisor of two natural numbers took about 16 seconds on the SparcStation 20[2]. The specialized program was between 25 and 40 times faster than the purely interpreted program. Since the store in the interpreter was implemented via a Scheme vector, the loops in the program were residualized, but interpretive overhead such as dispatch on syntax and the lookup of variable locations were eliminated at specialization time.

The code for POPE amounts to about 1600 lines of Scheme code in eight files. It should be able to run on any Scheme implementation supporting the R4RS [CR+91b] standard including hygienic macros, as well as the SLIB Scheme library.

[2]See the appendices

A The While Interpreter

```
(defineType Stmt
  (Skip z)
  (Var x)
  (Assign v e)
  (Cnd e s1 s2)
  (Seq s1 s2)
  (Wrt e)
  (While e s))

(defineType Expr
  (Add e1 e2)
  (Sub e1 e2)
  (Lt e1 e2)
  (Eq e1 e2)
  (Neg e1)
  (Cst x)
  (Var v))

(define the-store #(0))

(define (env-lookup v b)
  (if (assq v b)
      (cdr (assq v b))
      #f))

(define (terp s b n)
  (caseType s
     [(Skip z) (cons b n)]
     [(Var x)
      (cons (cons (cons x n) b) (+ n 1))]
     [(Assign v e)
      (begin
        (vector-set! the-store (env-lookup v b) (terp-expr e b n))
        (cons b n))]
     [(Cnd e s1 s2)
      (begin
        (if (terp-expr e b n)
            (terp s1 b n)
            (terp s2 b n))
        (cons b n))]
     [(Seq s1 s2)
      (let ((v1 (terp s1 b n)))
        (terp s2 (car v1) (cdr v1)))]
     [(Wrt e1)
      (begin
        (write (terp-expr e1 b n))
        (newline)
        (cons b n))]
```

```
     [(While e s1)
      (begin
        (CACHE #f terp-while e s1 b n)
        (cons b n))]
     [else 'sdfdsf]))

(define (terp-while cnd body b n)
  (if (terp-expr cnd b n)
      (begin
        (terp body b n)
        (CACHE #f terp-while cnd body b n))
      (cons b n)))

(define (terp-expr e b n)
  (caseType e
     [(Add e1 e2)
      (+ (terp-expr e1 b n) (terp-expr e2 b n))]
     [(Sub e1 e2)
      (- (terp-expr e1 b n) (terp-expr e2 b n))]
     [(Lt e1 e2)
      (< (terp-expr e1 b n) (terp-expr e2 b n))]
     [(Eq e1 e2)
      (= (terp-expr e1 b n) (terp-expr e2 b n))]
     [(Neg e1)
      (not (terp-expr e1 b n))]
     [(Cst c) c]
     [(Var v) (vector-ref the-store (env-lookup v b))]
     [else 'ffo]))
```

B Euclid

```
var count
var x
var y
count := 0
while count < 50
        x := 120
        y := 9
        while not(x = y)
                var foobar
                if x < y then
                        y := y - x
                else
                        x := x - y
                endif
        endwhile
        count := count + 1
        write x
endwhile
skip
```

C Specialization

The While interpreter specialized with respect to the Euclid program.

```
(define pope:func.2
  (lambda ()
    (if (not (= (vector-ref the-store '1)
                (vector-ref the-store '2)))
      (let ((pope:*foo*
              (begin
                (if (< (vector-ref the-store '1)
                       (vector-ref the-store '2))
                  (let* ((pope:*foo*
                           (begin
                             (vector-set! the-store '2
                               (- (vector-ref the-store '2)
                                  (vector-ref the-store '1)))
                             '(((foobar . 3) (y . 2) (x . 1) (count . 0))
                               . 4))))
                    '(((foobar . 3) (y . 2) (x . 1) (count . 0)) . 4))
                  (let* ((pope:*foo*
                           (begin
                             (vector-set! the-store '1
                               (- (vector-ref the-store '1)
                                  (vector-ref the-store '2)))
                             '(((foobar . 3) (y . 2) (x . 1) (count . 0))
                               . 4))))
                    '(((foobar . 3) (y . 2) (x . 1) (count . 0)) . 4)))
                '(((foobar . 3) (y . 2) (x . 1) (count . 0)) . 4))))
        (begin
          '(((foobar . 3) (y . 2) (x . 1) (count . 0)) . 4)
          (pope:func.2)))
      '(((y . 2) (x . 1) (count . 0)) . 3))))
(define pope:func.1
  (lambda ()
    (if (< (vector-ref the-store '0) '50)
      (let ((v1_27.486
              (let* ((v1_27.244
                       (let* ((v1_27.220
                                (let* ((pope:*foo*
                                         (begin
                                           (vector-set! the-store '1 '120)
                                           '(((y . 2) (x . 1) (count . 0))
                                             . 3))))
                                  '(((y . 2) (x . 1) (count . 0)) . 3)))
                              (pope:*foo*
                                (begin (vector-set! the-store '2 '9)
                                  '(((y . 2) (x . 1) (count . 0)) . 3))))
                         '(((y . 2) (x . 1) (count . 0)) . 3)))
                     (pope:*foo*
```

```
                    (begin (pope:func.2)
                      '(((y . 2) (x . 1) (count . 0)) . 3))))
                '(((y . 2) (x . 1) (count . 0)) . 3)))
            (v1_27.543
              (let* ((pope:*foo*
                      (begin
                        (vector-set! the-store '0
                          (+ (vector-ref the-store '0) '1))
                        '(((y . 2) (x . 1) (count . 0)) . 3))))
                '(((y . 2) (x . 1) (count . 0)) . 3)))
            (pope:*foo*
              (begin (write (vector-ref the-store '1)) (newline)
                '(((y . 2) (x . 1) (count . 0)) . 3))))
        (begin
          '(((y . 2) (x . 1) (count . 0)) . 3)
          (pope:func.1)))
      '(((y . 2) (x . 1) (count . 0)) . 3))))
(define pope:main
  (let* ((v1_27.93
          (let* ((pope:*foo*
                  (begin
                    (vector-set! the-store '0 '0)
                    '(((y . 2) (x . 1) (count . 0)) . 3))))
            '(((y . 2) (x . 1) (count . 0)) . 3)))
         (v1_27.575
           (let* ((pope:*foo*
                   (begin (pope:func.1)
                     '(((y . 2) (x . 1) (count . 0)) . 3))))
             '(((y . 2) (x . 1) (count . 0)) . 3))))
    '(((y . 2) (x . 1) (count . 0)) . 3)))
```

References

[Bon90] Anders Bondorf. Automatic Autoprojection of Higher Order Recursive Equations. In N. Jones, editor, *Proceedings of the 3rd European Symposium on Programming (ESOP'90)*, volume 432 of LNCS, pages 70–87, Copenhagen, May 1990. Springer-Verlag.

[BP93] Anders Bondorf and Jens Palsberg. Compiling Actions by Partial Evaluation. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture (FPCA'93)*, 1993.

[Con88] Charles Consel. New Insights into Partial Evaluation: the SCHISM experiment. In *Proceedings of the 2nd European Symposium on Programming (ESOP'88)*, volume 300 of LNCS, pages 236–246, Nancy, France, 1988. Springer-Verlag.

[Con94] Charles Consel. Report on Schism'94. Technical report, Oregon Graduate Inst., 1994.

[CR91a] William Clinger and Jonathan Rees. Macros that Work. In *Proc. of the 18'th annual ACM Symp. on Principles of Programming Languages (POPL 18)*, pages 155–162. ACM Press, 1991.

[CR+91b] William Clinger, Jonathan Rees, et al. Revised^4 Report on the Algorithmic Language Scheme. Technical report, MIT, 1991.

[Mos92] Peter D. Mosses. *Action Semantics*. Cambridge University Press, 1992. Number 26 in the Cambridge Tracts in Theoretical Computer Science series.

[Ruf93] Erik Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, Computer Systems Laboratory, March 1993. Technical Report CSL-TR-93-563.