Continuation-Based Partial Evaluation

Julia L. Lawall
Computer Science Department
Brandeis University [*]
(jll@cs.brandeis.edu)

Olivier Danvy
Computer Science Department
Aarhus University [**]
(danvy@daimi.aau.dk)

Abstract

Binding-time improvements aim at making partial evaluation (a.k.a. program specialization) yield a better result. They have been achieved so far mostly by hand-transforming the source program. We observe that as they are better understood, these hand-transformations are progressively integrated into partial evaluators, thereby alleviating the need for source-level binding-time improvements.

Control-based binding-time improvements, for example, follow this pattern: they have evolved from ad-hoc source-level rewrites to a systematic source-level transformation into continuation-passing style (CPS). Recently, Bondorf has explicitly integrated the CPS transformation into the specializer, thus partly alleviating the need for source-level CPS transformation. This CPS integration is remarkably effective but very complex and goes beyond a simple CPS transformation. We show that it can be achieved directly by using the control operators shift and reset, which provide access to the current continuation as a composable procedure.

We automate, reproduce, and extend Bondorf's results, and describe how this approach scales up to hand-writing partial-evaluation compilers. The first author has used this method to bootstrap the new release of Consel's partial evaluator Schism. The control operators not only allow the partial evaluator to remain in direct style, but also can speed up partial evaluation significantly.

1 Introduction

Partial evaluation is a program-transformation technique for specializing programs [11, 24]. It was developed in the sixties and seventies [1, 26], drastically simplified in the eighties for purposes of self-application [25], and is now evolving both quantitatively and qualitatively. Quantitatively, partial evaluators handle more and more programming-language features — types, higher-order procedures, data structures, and so on. Qualitatively, these features need to be handled effectively. This is where *binding-time improvements* intervene [24, Chap. 12].

1.1 Binding-time improvements

The notion of binding time arises naturally in partial evaluation since source programs are evaluated in two stages: at partial-evaluation time (statically) and at run time (dynamically). The parts of the source program that can be evaluated statically are referred to as "static" and the others as "dynamic".

Obviously, the more static parts there are in a source program, the better it gets specialized. A binding-time improvement is a source-level transformation that enables more parts to be classified as static. Say that we want to partially evaluate the expression

$$x + (y - 1)$$

where we know that x is bound statically and y is bound dynamically. A naïve binding-time analysis would classify both the subtraction and the addition to be dynamic, since in each case one of the operands is dynamic. Using the associativity and commutativity laws of arithmetic, we can rewrite the expression as follows.

$$y + (x - 1)$$

The same naïve binding-time analysis would now classify the subtraction to be static (since x will be known at partial-evaluation time and 1 is an immediate constant) and the addition to be dynamic (since y will not be known until run time). By rewriting the expression, we have achieved a binding-time improvement: the same binding-time analysis classifies more expressions as static, thus enabling the same specializer to do a better job. Overall, the same partial evaluator yields a better result.

1.2 Evaluation and partial evaluation

Partial evaluation mimics evaluation — computing values of static expressions, but residualizing (*i.e.*, reconstructing) dynamic expressions, to produce the specialized program. When an expression is residualized, the continuation of the partial evaluation of its components may differ from the continuation of their evaluation. This can cause a loss of static information. Consider the Scheme expression[1]

---

[*] Waltham, Massachusetts 02254, USA. This work was initiated at the Oregon Graduate Institute of Science & Technology in summer 1993; continued at Indiana University in fall 1993; and was completed at Brandeis University. It was partially supported by NSF under grant CCR-9224375 and by ONR under grant N00014-93-1-1015.

[**] Ny Munkegade, 8000 Aarhus C, Denmark.

[1] The square brackets can be read as parentheses.

```
(+ 1 (let ([x D]) 3))
```

where `D` is a dynamic expression. The let expression must be residualized.[2] The value of the body, 3, is thus returned to the reconstructor of the let expression rather than to the outer addition, as would happen under ordinary evaluation.

Continuations themselves provide a solution to this problem. Propagating the result of specializing 3 directly to the continuation of the let expression allows the outer addition to take place at specialization time.

A series of approaches have been suggested to take advantage of continuations during the process of partial evaluation. Consel and Danvy [10] make the continuations explicit, by rewriting the source program into continuation-passing style. This transformation provides an inter-procedural improvement when the continuation is static. Writing the source program in CPS allows the user to control the binding-time value of the continuation using *e.g.*, Schism's filters [8]. Furthermore continuations that are explicit in the source program can benefit from polyvariant binding-time analysis, if available [7].

Holst and Gomard [22], on the other hand, propose rewriting each source procedure to relocate the context of every let expression into its body. Because only the context syntactically apparent from the body of the source procedure is relocated, this is a purely intra-procedural improvement. The transformation is a reflection of a transformation on continuations back into the direct-style world. As a consequence, the source program remains in the familiar direct style.

Alternatively, this binding-time improvement can be integrated into the specializer. Then the source program is completely unchanged. Bondorf develops such a specializer in two steps [3]. First, the specializer (PE_d) is rewritten in CPS by hand (PE_c), to make its continuations explicitly accessible. Then, the continuations are manipulated in non-standard ways (PE'_c) so that, in effect, static values reach their consumer statically, *i.e.*, at partial-evaluation time. Bondorf's approach is characterized by the following diagram:

$$\begin{tikzcd}
\mathrm{PE}_d \arrow[d, "\text{manual CPS transformation [3, Sect. 3]}"'] & \\
\mathrm{PE}_c \arrow[r, "\text{manual binding-time improvement [3, Sect. 5]}"'] & \mathrm{PE}'_c
\end{tikzcd}$$

This approach is documented further in Jones, Gomard, and Sestoft's textbook on partial evaluation [24, Chap. 10].

The resulting specializer achieves inter-procedural improvements as well. Because, however, access to continuations is built into the specializer, significant strategic changes to the partial evaluator are required to achieve all the improvements possible with hand CPS transformation [3, Sect. 6.4], as analyzed elsewhere [27].

Nevertheless, in most situations, continuation-based specialization yields good results. It improves the specialization of both source let expressions and let expressions introduced to preserve the linearity of dynamic computation (see Footnote 2). It is even necessary in Similix, where it enabled the treatment of partially-static values [3, Sec. 7]

Unfortunately, in Bondorf's paper and in Jones *et al.*'s textbook,

- the specializer is CPS-transformed by hand;
- its continuations need skillful massage — the more complicated the specializer, the more skillful the massage; and
- the resulting specializer is written in something almost like CPS, which requires even more sophistication to develop further, let alone maintain.

In this paper, we show that the non-standard uses of the continuation in a continuation-based specializer PE'_c precisely correspond to the effect of the control operators shift and reset [13, 14], and that inserting shift and reset at a few selected places (PE'_d) and then converting the specializer into CPS (PE'_c) automatically yields Similix's continuation-based specializer [3]. This makes it possible to keep both the source program and the specializer in the familiar direct style. Our approach is characterized by the following diagram:

$$\begin{tikzcd}
\mathrm{PE}_d \arrow[r, "\text{manual binding-time improvement (Section 2)}"] & \mathrm{PE}'_d \arrow[d, "\text{automatic CPS transformation}"] \\
& \mathrm{PE}'_c
\end{tikzcd}$$

The transformation from PE'_d to PE'_c is not an issue here, since it can be carried out automatically, as indeed the transformation from PE_d to PE_c in Bondorf's work could have been. We express the control-based binding-time improvements at the source (direct-style) level, in essence "improving binding times without explicit CPS-conversion", neither of the source programs nor of the specializer. At no point do we CPS-transform by hand. In fact it is not necessary to CPS transform at all, if we use Filinski's implementation of shift and reset using call/cc and set! [18]. The resulting direct-style specializer appears to be more efficient than its CPS counterpart.

2 Continuation-Based Program Specialization

2.1 Similix's continuation-based specializer

We start from a simple specializer for Scheme programs (see Figure 2). It is adapted from Bondorf's original specializer for Similix [3, Fig. 3]. The source language is a side-effect free subset of Scheme extended with a `Lift` operator

---

[2] In general, dynamic let-expressions are residualized to preserve the termination properties of the source program (as is the case in this example), to preserve the sequencing order of the source program (should `D` contain side-effects), or to avoid rampant duplication in the specialization phase (should the let-bound variable occur several times).

```
(define-type Annotated-Expression
  (Over Expression)
  (Under Expression))

(define-type Expression
  (Lift Annotated-Expression)
  (Constant Literal)
  (Identifier Symbol)
  (Conditional Annotated-Expression Annotated-Expression Annotated-Expression)
  (Let-Block Symbol Annotated-Expression Annotated-Expression)
  (Operation Symbol Annotated-Expressions)
  (First-Order-Application Symbol Annotated-Expressions)
  (Higher-Order-Application Annotated-Expression Annotated-Expressions)
  (Tagged-Abstraction Unique-Tag))
```

Figure 1: BNF of the source language

that converts a value into a residual expression. The specializer is implemented in Scheme extended with some Schism syntactic-sugar — destructuring `let-type` expressions and `case-type` expressions to pattern-match structured data [8].

A program to be specialized (*i.e.*, the source program) is first annotated by a binding-time analysis. An expression to be evaluated during specialization is marked `Over`, while an expression to be residualized is marked `Under` (see Figure 1). The correctness of the binding-time analysis ensures that the annotations are consistent.

Figure 3 displays a fragment of the continuation-based specializer of Similix, `specialize-c` [3]. The treatment of dynamic let expressions deserves comment. The specializer processes a dynamic let expression with some continuation `k`. After its header is processed, the let expression is residualized, and its body is specialized *with the same continuation* `k`, thus achieving the control-based binding-time improvement described in Section 1.2.

Such a continuation-based specializer is deliberately designed *not* to be in CPS, as illustrated by the following examples.

- The call to `specialize-c` is not a tail call in the treatment of the body of a dynamic let expression.
- The call to `specialize-c` is not a tail call in the branches of a dynamic conditional expression.
- In the treatment of dynamic conditional branches, the continuation is reset to be the identity procedure.

In the direct-style world, these manipulations over continuations are precisely captured by the following two control operators.

2.2 The control operators shift and reset

Shift and reset were introduced to capture composition and identity over continuations [13, 14]. Reset delimits a context, and is identical to Felleisen's prompt; shift abstracts a delimited context into a procedure, and is similar (though not in general equivalent) to Felleisen's control [16]. Unlike call/cc, which captures the *whole* context of a computation [5], shift captures a *delimited* context. Thus first-class continuations abstracted by shift can be composed, whereas first-class continuations abstracted by call/cc cannot. Unlike call/cc, the value of the body of a shift expression is not returned to the context, unless the continuation is explicitly applied. Using the continuation linearly enables one to relocate a context without duplicating or discarding computation — which is our motivation here.

2.2.1 Example

$$
\begin{aligned}
(2 \times (3 \times 4)) + 1 &= \\
&= 25 \\
\mathit{reset}(2 \times \mathit{shift}\ k\ \mathit{in}\ 3 \times 4) + 1 &= (3 \times 4) + 1 \\
&= 13 \\
\mathit{reset}(2 \times \mathit{shift}\ k\ \mathit{in}\ k(3 \times 4)) + 1 &= (2 \times (3 \times 4)) + 1 \\
&= 25 \\
\mathit{reset}(2 \times \mathit{shift}\ k\ \mathit{in}\ 3 \times k(4)) + 1 &= (3 \times (2 \times 4)) + 1 \\
&= 25 \\
\mathit{reset}(2 \times \mathit{shift}\ k\ \mathit{in}\ 3 \times k(k(4))) + 1 &= (3 \times (2 \times (2 \times 4))) \\
&\quad + 1 \\
&= 49
\end{aligned}
$$

In the first term, where there is no shift expression, the computations are a multiplication by 2, a multiplication of 4 by 3, and an increment by 1. In the remaining terms, k is bound to a procedural abstraction of the delimited context $[2 \times [\ ]]$. In the second term, k is not used and thus the context $[2 \times [\ ]]$ is abandoned. In the third term, the context is relocated on site. In the fourth term, the context is relocated inside the multiplication by 3. The continuation is used linearly and thus the same computations occur as in the first and third terms, albeit in a different order. In the last term, the context is relocated and duplicated inside the multiplication by 3.

In the rest of this paper, we do not use the full power of shift, in that we do not discard contexts and we do not duplicate them. Instead, we only use continuations linearly to relocate contexts.

```
(define specialize
  (lambda (e env)
    (case-type e
      [(Over e)
       (case-type e
         [(Constant c)
          c]
         [(Identifier i)
          (lookup-local i env)]
         [(Conditional test consequent alternative)
          (if (specialize test env)
              (specialize consequent env)
              (specialize alternative env))]
         [(Let-Block formal actual body)
          (specialize body (extend-one formal (specialize actual env) env))]
         [(Operation operator actuals)
          (call operator (map (lambda (e) (specialize e env)) actuals))]
         [(First-Order-Application procedure actuals)
          (let-type ([(First-Order-Closure formals body) (lookup-global procedure)])
            (specialize body (extend formals
                                     (map (lambda (e) (specialize e env)) actuals)
                                     empty-env)))]
         [(Higher-Order-Application procedure actuals)
          (let-type ([(Higher-Order-Closure tag free-vals) (specialize procedure env)])
            (let-type ([(Annotated-Abstraction formals body free-vars) (lookup-lambda tag)])
              (specialize body (extend formals
                                       (map (lambda (e) (specialize e env)) actuals)
                                       (extend free-vars free-vals empty-env)))))]
         [(Tagged-Abstraction tag)
          (let-type ([(Annotated-Abstraction formals body free-vars) (lookup-lambda tag)])
            (Higher-Order-Closure tag (map (lambda (x) (lookup-local x env)) free-vars)))])]
      [(Under e)
       (case-type e
         [(Lift e)
          (Constant (specialize e env))]
         [(Conditional test consequent alternative)
          (Conditional (specialize test env)
                       (specialize consequent env)
                       (specialize alternative env))]
         [(Let-Block formal actual body)
          (let ([new-formal (gensym! formal)])
            (Let-Block new-formal
                       (specialize actual env)
                       (specialize body (extend-one formal new-formal env))))]
         [(Operation operator actuals)
          (Operation operator (map (lambda (e) (specialize e env)) actuals))]
         [(First-Order-Application procedure actuals)
          (let-type ([(Memoized new-procedure new-actuals)
                      (memoize! procedure (map (lambda (e) (specialize e env)) actuals))])
            (First-Order-Application new-procedure new-actuals))]
         [(Higher-Order-Application procedure actuals)
          (Higher-Order-Application (specialize procedure env)
                                    (map (lambda (e) (specialize e env)) actuals))]
         [(Tagged-Abstraction tag)
          (let-type ([(Annotated-Abstraction formals body free-vars) (lookup-lambda tag)])
            (let ([new-formals (map gensym! formals)])
              (Abstraction new-formals
                           (specialize body (extend formals new-formals env)))))])])))
```

Figure 2: Direct-style specializer for Similix

```
(define specialize-c
  (lambda (e env k)
    (case-type e
      [(Over e)
       (case-type e
         ...
         [(Identifier i)
          (k (lookup-local i env))]
         [(Conditional test consequent alternative)
          (specialize-c test env (lambda (b)
                                   (if b
                                       (specialize-c consequent env k)
                                       (specialize-c alternative env k))))]
         ...)]
      [(Under e)
       (case-type e
         [(Lift e)
          (specialize-c e env (lambda (v) (k (Constant v))))]
         [(Conditional test consequent alternative)
          (specialize-c test env (lambda (b)
                                   (k (Conditional b
                                                   (specialize-c consequent env (lambda (a) a))
                                                   (specialize-c alternative env (lambda (a) a))))))]
         [(Let-Block formal actual body)
          (let ([new-formal (gensym! formal)])
            (specialize-c actual
                          env
                          (lambda (v)
                            (Let-Block new-formal v (specialize-c body
                                                                  (extend-one formal new-formal env)
                                                                  k)))))]   ;;; <---***---
         ...
         [(Tagged-Abstraction tag)
          (let-type ([(Annotated-Abstraction formals body free-vars) (lookup-lambda tag)])
            (let ([new-formals (map gensym! formals)])
              (k (Abstraction new-formals
                              (specialize-c body (extend formals new-formals env) (lambda (a) a))))))])])))
```

Figure 3: Purely functional, continuation-based specializer for Similix (excerpts)

2.2.2 CPS transformation of shift and reset expressions

A shift expression is naturally CPS-transformed by abstracting the current (delimited) continuation into a procedure. This continuation is composed with the new current continuation at any point where the procedure is applied. A reset expression is naturally CPS-transformed by supplying the identity procedure as a continuation. The following equations summarize Plotkin's call-by-value CPS transformation for the λ-calculus extended with shift and reset. This CPS transformation is documented further in the literature [13, 14].

$$\begin{array}{rcl}
[\![x]\!] & = & \lambda k.k\ x \\
[\![\lambda x.e]\!] & = & \lambda k.k(\lambda x.[\![e]\!]) \\
[\![e_0\ e_1]\!] & = & \lambda k.[\![e_0]\!](\lambda v_0.[\![e_1]\!](\lambda v_1.v_0\ v_1\ k)) \\
[\![\mathit{shift}\ j\ \mathit{in}\ e]\!] & = & \lambda k.(\lambda j.[\![e]\!](\lambda a.a))(\lambda v.\lambda k'.k'(k\ v)) \\
[\![\mathit{reset}(e)]\!] & = & \lambda k.k([\![e]\!](\lambda a.a))
\end{array}$$

2.2.3 Direct implementation of shift and reset

Implementing shift and reset does not require a CPS transformation. Their expressive power lies in a single-threaded "meta-continuation" [13, 35] that in effect is structured like a push-down stack, and thus can be globalized in a register [32] (see also [29]). This led Filinski to a direct-style implementation using only call/cc and side-effects over the meta-continuation register [18]. The Scheme implementation is shown in Figure 7, at the end of the paper.

2.3 Continuation-based specialization in direct style

We now show how shift and reset can implement the nonstandard uses of continuations in Similix. To determine where control operators should be introduced, we use the intuition from the beginning of Section 1.2 that static information is hidden when the continuation of the partial evaluator differs from that of the evaluator.

2.3.1 A relation over continuations

As pointed out at the beginning of Section 1.2, a partial evaluator mimics an evaluator: it computes static expressions but residualizes dynamic expressions. Let us rephrase this statement in terms of continuations.[3]

Essentially, a continuation κ is either the initial continuation or is constructed by composing a function f and a continuation κ':

$$\kappa = \kappa' \circ f.$$

Every function in the continuation of the evaluator performs an evaluation step. The functions comprising the continuation of the partial evaluator can be classified as follows.

- Evaluating functions, which perform an evaluation step.
- Residualizing functions, which residualize an evaluation step.
- Bookkeeping functions, which do not correspond to an evaluation step.

As an example of the relationship between the continuation of the evaluator and the continuation of the partial evaluator of Figure 2, consider the expression

```
(Operation op2 (Lift (Over (Operation op1 e1))) e2)
```

The expression `e1` is evaluated with a continuation beginning with a function that applies `op1`. The continuation of its partial evaluation begins with the same function, which is thus an evaluating function. The expression

```
(Lift (Over (Operation op1 e1)))
```

is evaluated with a continuation beginning with a function that evaluates `e2` followed by a function that applies `op2`.[4] It is partially evaluated with a continuation beginning with two functions, of which the first specializes `e2` and the second residualizes `op2`. The first of these functions is an evaluating function while the second is a residualizing function.

As another example, consider a conditional expression. The continuation of the evaluation of either branch is the continuation of the entire conditional expression. If the test part is dynamic, the consequent branch is partially evaluated with a continuation beginning with two functions: the first partially evaluates the alternative branch, and the second reconstructs the entire conditional expression. Both of these are bookkeeping functions.

We now use the classification to achieve a control-based binding-time improvement. Consider an expression whose evaluation reduces to the evaluation of a subexpression. The value of the subexpression is sent to the continuation of the expression. If, during partial evaluation, the continuation starts with a bookkeeping function, the value of the subexpression is lost to further partial evaluation. We can achieve a control-based binding-time improvement by bypassing the bookkeeping function and relocating the continuation of the expression from the residualized expression to the subexpression. For example, the following semantic equality (modulo renaming) illustrates this relocation for let expressions in the CPS world.

```
(k (let ([i1 e1] ...) e))
== (let ([i1 e1] ...) (k e))
```

To preserve the semantics of the entire program, we can only apply such a transformation when the continuation of the partial evaluator contains only residualizing or evaluating functions. The correctness of the transformation then follows by compositionality.

We call residualizing and evaluating functions *safe functions*. They correspond to Bondorf's safe contexts [3, Def. 11].

In general, κ, the continuation of the partial evaluator, may also contain bookkeeping functions. There is, however, always a prefix of the sequence of functions comprising κ that contains only safe functions (in the minimal case, $\lambda a.a$). Let us divide κ as $\kappa' \circ \delta$, where δ is a composition only of safe functions. We call δ a *safe prefix*.

Of course, there may be many such divisions of the continuation. Achieving the greatest improvement depends on choosing the largest possible safe prefix. By the compositionality of both the evaluator and the partial evaluator, we may then use semantic equalities to transform the application of δ. κ' is applied, as before, to the result of the transformation.

The problem of implementing the binding-time improvement on let expressions then reduces to the problem of identifying the safe prefix. Bondorf keeps track of it as a separate argument to the partial evaluator (`k` in Figure 3; thus, his "well-behaved continuations" [3, Def. 14] *are* our safe prefixes.). We propose instead to use the control operator `reset` to separate δ from κ'. The control operator `shift` then provides access to δ when necessary.

2.3.2 Delimiting a safe prefix with reset

Let us locate where the continuation of the specializer is extended with a bookkeeping function. In such cases we set the safe prefix to the identity function, $\lambda a.a$. This is precisely the effect of a reset operation (see Section 2.2.2). Throughout our analysis, we assume the following invariant: in the continuation of the current call to the specializer, the safe prefix has been separated from the rest of the continuation.

The specialization of a static expression follows its evaluation, so each extension to the continuation is a safe function. Thus no control operators are needed.

Dynamic expressions are residualized. The continuation of the specialization of a subexpression might thus begin with a bookkeeping function. Let us proceed case by case.

In the case of the argument of a lift expression, the test in a conditional expression, the header of a let expression, and all the arguments of each kind of application, the continuation of the specialization residualizes the action performed by the continuation of their evaluation. Thus no control operators are needed.

In the specialization of the following subexpressions, however, the continuation begins with a bookkeeping function.

1. The body of a let expression.
2. The branches of a conditional expression (as described in Section 2.3.1).
3. The body of a lambda abstraction.
4. The body of the top-level definition of a memoized procedure (*i.e.*, a specialization point [24]).

---

[3] The fact that continuations are implicit (in a direct-style program) or explicit (in a continuation-passing program) is irrelevant here since programs can be transformed between direct style and CPS automatically.

[4] This scenario assumes left-to-right sequencing order. A corresponding scenario holds for any other sequencing order.

```
(define specialize
  (lambda (e env)
    (case-type e
      [(Over e)    ;;; as in Figure 2
       ...]
      [(Under e)
        (case-type e
         [(Lift e)
          (Constant (specialize e env))]
         [(Conditional test consequent alternative)
          (Conditional (specialize test env)
                        (RESET (specialize consequent env))
                        (RESET (specialize alternative env)))]
         [(Let-Block formal actual body)
          (SHIFT k (let ([new-formal (gensym! formal)])
                      (Let-Block new-formal
                                 (specialize actual env)
                                 (RESET (k (specialize body (extend-one formal new-formal env)))))))]
         ...   ;;; as in Figure 2
         [(Tagged-Abstraction tag)
          (let-type ([(Annotated-Abstraction formals body free-vars) (lookup-lambda tag)])
            (let ([new-formals (map gensym! formals)])
              (Abstraction new-formals
                            (RESET (specialize body (extend formals new-formals env))))))])])))
```

Figure 4: Direct-style, continuation-based specializer for Similix (excerpts)

In each case, the specializer reconstructs the enclosing expression while the evaluator sends the value to some other continuation. Thus we enclose each specialization in a reset expression. In the fourth case, the reset expression occurs in the definition of `memoize!`, which is not shown.

In the resulting specializer, the continuation up to the enclosing reset contains only safe functions.

Fact 1 *Transforming this specializer into CPS automatically yields Bondorf's* $\mathcal{C}^1$ *[3, Fig. 5].*

2.3.3 Improving the safe prefix with shift

Each reset expression delimits a minimal safe prefix. In some cases, however, a longer prefix is locally available. Exploiting it achieves a binding-time improvement. Let us consider where a shift expression can be used to access a longer safe prefix for each of the four cases above.

The body of a let expression is evaluated with the continuation of the entire let expression. Thus, we may rewrite the specialization of a let expression as follows.

```
(shift k
  (let ([new-formal (gensym! formal)])
    (Let-Block
      new-formal
      (specialize actual env)
      (reset (k (specialize body
                             (extend-one formal
                                         new-formal
                                         env)))))))
```

Similarly, each conditional branch is evaluated with the continuation of the entire conditional expression. We could use shift to access a safe prefix here, but doing so would duplicate the context in each branch. Not making the improvement, as in Figure 4, causes a (safe) loss of information and avoids residual-code explosion.

Finally, for the body of a lambda abstraction, the continuation of the evaluation is not available until the lambda abstraction is applied. As for the conditional branches, our specializer is still correct, but it could produce better results if the specialization of the body could interact directly with the continuation of each corresponding call. Specialization points (*i.e.*, memoized procedures), however, should not be specialized with respect to a continuation, to increase sharing in residual programs [3, 27, 30].

Figure 4 shows the direct-style specializer with the control operators inserted as described above. (For readability, the occurrences of shift and reset are capitalized.)

Fact 2 *Transforming this specializer into CPS automatically yields the continuation-based specializer of Similix [3, Fig. 6] (see also Figure 3).*

2.4 Assessment

The specializers in Figures 2 and 4 are almost identical. Only in five places do they differ. In the four cases listed in Section 2.3.2 a reset separates the safe prefix from the rest of the continuation. In the specialization of a dynamic let expression, a shift enables the binding-time improvement.

Superimposing the two diagrams of Section 1.2 yields the following commuting diagram.

```
(define specialize
  (lambda (e env)
    (case-type e
      [(Over e)
       ...]
      [(Under e)
       (case-type e
         ...
         [(Higher-Order-Application procedure actuals)
          (case-type procedure
            [(Tagged-Abstraction tag)
             (let-type ([(Annotated-Abstraction formals body free-vars) (lookup-lambda tag)])
               (SHIFT k (let ([new-formals (map gensym! formals)])
                          (Higher-Order-Application (Abstraction new-formals
                                                                 (RESET (k (specialize body
                                                                                       (extend formals
                                                                                               new-formals
                                                                                               env)))))
                                                    (map (lambda (e) (specialize e env)) actuals)))))]
            [else
             (Higher-Order-Application (specialize procedure env)
                                       (map (lambda (e) (specialize e env)) actuals))])]
         ...)])))
```

Figure 5: Extension of Figure 4 for beta-redexes

$$\begin{tikzcd}
\mathrm{PE}_d \arrow[r] \arrow[d] & \mathrm{PE}'_d \arrow[d] \\
\mathrm{PE}_c \arrow[r] & \mathrm{PE}'_c
\end{tikzcd}$$

In particular, the correctness proof of Bondorf's approach applies here as well.

2.5 Improvements

Beta redexes offer an opportunity for further binding-time improvement. A beta redex is an on-site application of a lambda abstraction. This situation appears frequently after macro expansion. When a beta redex is residualized, its context can still be relocated in the body of the corresponding lambda abstractions, *à la* Sabry and Felleisen [31]. In fact, in early 1990, Schism used continuations to carry out this relocation, thereby already achieving the let optimization (since in Schism, let is syntactic sugar for a beta redex, *à la* Landin).

To extend the specializer to improve over residual beta redexes, we only need to change the clause for higher-order applications to detect a beta redex and then to apply the same treatment as for let expressions (see Figure 5). The simplicity of this change illustrates the flexibility of the direct-style approach.

As another extension to the continuation-based specializer, consider the case of a let expression whose body yields a dynamic value (a similar argument applies to a beta redex). Sending the residual let-body to the continuation of the let expression does not introduce any extra opportunity for specialization. Since there is no benefit from the reorganization of the continuations in this case, we may leave the continuation as it is in the original specializer (see Figure 2), thus saving a shift instruction. In our experiments, however, this did not improve efficiency (see Section 4).

Further improvements are possible. For example, in Schism, binding-time information is compiled into specialization actions, to remove interpretive overhead in the specializer [9]. Completely static and completely dynamic expressions are factored out, and the specializer can concentrate on specialization proper, with a smaller continuation traffic. As addressed in Section 3.1, the shift/reset method applies there as well.

2.6 Issues for binding-time analysis

Consel and Khoo present binding-time analysis as an abstraction of partial evaluation [12]. Along that line, continuation-based partial evaluation affects the binding-time analysis at least with respect to the two following points. We mention them here only for the sake of completeness.

Since the continuation of a let expression is directly transmitted to the body of the let expression instead of being sent the result of the whole expression, it is overly conservative to "lub" the results of analyzing the let-header and the let-body to compute the result of analyzing the whole expression. Instead, the result of analyzing the let expression is simply the result of analyzing the let-body.

The same point applies for conditional expressions. If the continuation of a conditional expression were duplicated and directly transmitted to the conditional branches instead of being sent the result of the whole expression, the result of analyzing a conditional expression could reduce to "lubbing" the results of analyzing the conditional branches.

3 Applications

3.1 Bootstrapping

Using control operators and the CPS transformation enables the following bootstrapping procedure: freely experiment with control in the direct-style specializer, and automatically generate the corresponding purely functional one (*i.e.*, without shift and reset). This bootstrapping method is not only conceptually interesting but it has practical interest as well. The first author has used this method to bootstrap the specializer in the new version of Schism [8]. The specializer occupies 24 Kb of Scheme code in direct style and 27 Kb after CPS transformation (19 and 22 with no tabulation characters and no pretty-printing).

3.2 Self-applicable partial evaluation

The area of self-applicable partial evaluation is seeing a new trend: writing "pecom" (a.k.a. "cogen" and "cocom" [23, 24]) by hand. "Pecom" is a partial-evaluation compiler that generates dedicated specializers. It is customarily generated by self-application.[5] Recently, Holst and Launchbury have observed that writing pecom by hand makes it possible to bypass the double-encoding problem for self-applicable partial evaluation of typed programs [23]. Birkedal and Welinder have developed such a partial-evaluation compiler, SML-Mix, for Standard ML programs [2].

Writing SML-Mix by hand was hard enough. Yet it is in direct style and the dedicated specializers it produces are in direct style too. Without guidelines, a considerable amount of expertise would be necessary (1) to make it continuation-based and (2) to make it generate continuation-based dedicated specializers. Yet control-based binding-time improvements are crucially needed in this approach as well as in the traditional one.

Fortunately, the pecom approach smoothly meshes with our control-based binding-time improvement in direct style as follows.

- Insert shift and reset in pecom, as outlined in Section 2.3, and automatically CPS-transform the modified pecom. This yields a purely functional continuation-based pecom.
- Modify pecom to insert shift and reset at selected places (as outlined in Section 2.3) in the dedicated specializers it generates, and automatically CPS-transform the output of pecom. This yields purely functional continuation-based dedicated specializers.

In other words, let pecom$'$ denote a continuation-based pecom (with shift and reset) generating continuation-based specializers (with shift and reset); and let $[\![.]\!]$ denote the CPS transformation, as in as in Section 2.2.2.

- $[\![\text{pecom}']\!]$ yields a purely functional continuation-based pecom (without shift and reset).
- $[\![\text{run } [\![\text{pecom}']\!] \text{ p}]\!]$ yields a purely functional continuation-based specializer dedicated to p (without shift and reset).

Et voilà.

Efficiency, however, might dictate that pecom and the compiled partial evaluators be run in direct style. (Then the remaining "negative effects" Bondorf attributes to CPS would disappear [3, Sec. 6.2].) Both experiments remain to be done.

4 Measures

Analyzing the cost of our four partial evaluators PE_d, PE_c, PE'_d, and PE'_c raises two issues. We need to compare the costs of the CPS and DS implementations, and the cost of the continuation-based specializers over the cost of the naïve specializers. These comparisons are of course difficult to make because they depend greatly on the source program.

We consider a simple pattern-matching program, specialized with respect to both small and large patterns. The shape of a pattern determines the size of the continuation and the size of the residual program, independently. Essentially the source program contains both a regular call and a tail call. Increasing the depth of a pattern provokes more regular calls, making the continuation grow. Increasing the number of variables in a pattern (its "length") increases the size of the resulting substitution and thus the size of the residual program.

We consider two versions of the source program. In both there is only one dynamic let expression. In the first, `pm-static`, its body yields a static value and thus offers an opportunity for control-based binding-time improvements. In the second, `pm-dynamic`, its body yields a dynamic value and thus offers no opportunity for control-based binding-time improvements.

The following tables compare the time and space costs of specializing these two programs using PE_d, PE'_d, PE_c, and PE'_c. We consider patterns of varying depth and length.

4.1 Comparing the CPS and DS specializers

Most compilers are written to perform well on what is perceived to be a typical source program. In particular most compilers, even those using a CPS intermediate representation, are targeted toward DS source programs. Scheme goes as far as to leave the sequencing order unspecified to allow the compiler writer further opportunities for optimization. The direct-style specializer can take advantage of any such optimizations. Furthermore, call/cc, which we use to implement both shift and reset, has been highly optimized in Chez Scheme.[6]

These observations are borne out by the experiments above. In all but one case the CPS specializer uses more space than its DS counterpart, and in some cases almost twice as much. The time consumed by each CPS specializer is about the same as that of its DS counterpart.

---

[5] Pecom is obtained by partially evaluating the partial evaluator with respect to itself. Running pecom on a program p yields the same result (but faster) as partially evaluating the partial evaluator with respect to p, *i.e.*, it yields a partial evaluator specialized with respect to p. In turn, running this dedicated partial evaluator on an input s yields the same result (but faster) as partially evaluating p with respect to s, *i.e.*, it yields a version of p specialized with respect to s. Finally, running this specialized version of p on the remaining input d yields the same result (but faster) as running the program p on the complete input s and d. A concise presentation is available elsewhere [11, Apdx.]. Self-applicable partial evaluation is extensively treated in Jones *et al.*'s book [24].

[6] The Chez Scheme compiler is a DS compiler with an efficient implementation of first-class continuations [21]. We plan to repeat these measures using a CPS compiler.

`pm-static`									
		$\mathrm{PE}_c/\mathrm{PE}_d$		$\mathrm{PE}'_c/\mathrm{PE}'_d$		$\mathrm{PE}'_d/\mathrm{PE}_d$		$\mathrm{PE}'_c/\mathrm{PE}_c$	
depth	length	time	space	time	space	time	space	time	space
small	small	1.18	1.58	1.04	1.48	1.18	1.13	1.04	1.06
small	large	0.97	1.84	1.00	1.00	2.89	259.23	2.99	141.51
large	small	0.87	1.95	1.11	1.97	0.04	0.92	0.04	0.93
large	large	1.02	1.90	1.00	1.01	1.90	137.68	1.86	72.85

`pm-dynamic`									
		$\mathrm{PE}_c/\mathrm{PE}_d$		$\mathrm{PE}'_c/\mathrm{PE}'_d$		$\mathrm{PE}'_d/\mathrm{PE}_d$		$\mathrm{PE}'_c/\mathrm{PE}_c$	
depth	length	time	space	time	space	time	space	time	space
small	small	1.17	1.58	1.58	1.63	1.04	1.04	1.41	1.08
small	large	0.96	1.84	0.99	1.71	0.99	1.08	1.02	1.00
large	small	0.88	1.95	0.95	1.76	0.84	1.11	0.90	1.00
large	large	0.98	1.90	1.08	1.73	0.94	1.10	1.03	1.00

Figure 6: Relative costs of specializing two pattern-matching programs with respect to several patterns

4.2 Assessing the cost of the improvement

Continuation-based specialization introduces extra opportunities for specialization in `pm-static` because the body of the single dynamic let expression yields a static value. Tests on this value can be reduced statically. In contrast, these tests are residualized by the naïve specializer. Thus this specializer performs much less static computation and thus constructs many fewer data structures.

The binding-time improvement does not introduce any extra opportunities for specialization in `pm-dynamic`, where the body of the dynamic let expression yields a dynamic value. Thus the residual programs produced by the improved and naïve specializers are similar. The time and space costs are, as would be hoped, quite similar as well. In Section 2.5, we suggested that when the body of a dynamic let expression returns a dynamic value, it is useless to reorganize the continuations. In our experiments we observed that this "optimization" does not affect performance, on this source program.

4.3 Further experiments

We have noted that when the body of a dynamic let expression has a dynamic value, continuation-based specialization does not produce a better specialized program. Dually, when the continuation accessed by shift is always the minimal safe prefix, continuation-based specialization does not introduce more opportunities for binding-time improvement either (*i.e.*, it is always the identity function that is propagated to the body of the let expression). This case arises in iterative programs, *e.g.*, CPS programs. Therefore the runtime and runspace of the continuation-based specializer should be identical to that of the naïve specializer. As noted in Section 1.2, all four specializers should produce the maximally specialized residual program as well. Conversely, when a tail-recursive program is specialized, the control stack of PE_d and the continuation of PE_c do not grow (see [10, Prop. 1]). Therefore their runtime and their runspace should not diverge. We are currently experimenting to test this analysis.

As a final experiment, we measured the time and space costs of PE'_d when shift and reset are replaced by control and prompt.[7] Both costs are virtually unchanged.

4.4 Conclusion and side-issues

Independently of partial evaluation, the measures above compare running DS code, possibly in the presence of first-class continuations, and running naïve CPS code — where by "naïve" we mean that no special provision is made for the continuation [19]. PE'_d yields a situation where large parts of the same continuation are repeatedly captured and restored.[8] The common parts are naturally shared in CPS, but in DS, some initiative is needed to avoid duplicating entire copies of the control stack in the heap to accommodate first-class continuations, a serious concern for their efficient implementation.

5 Related work

Independently of our work, Friedman and Ashley observed that shift and reset could be used to develop a self-applicable partial evaluator [20]. Felleisen also observed that Bondorf's key step corresponds to delimiting and abstracting control [17].

Like Similix and Schism, the ML partial evaluator of Malmkjær, Heintze, and Danvy also uses continuation-based specialization [28].

Other continuation-based program transformations can be expressed in direct style with shift and reset. One example is the CPS transformation itself [13, 14]. Another is the nqCPS transformation.[9]

The present work is also a part of our investigation of the direct-style transformation in the presence of first-class continuations [15].

---

[7] In Figure 4, each continuation application is surrounded by a reset. Thus Felleisen's control (a.k.a. $\mathcal{F}$) and prompt operators [16] can be used here as well, even though they go beyond CPS (*i.e.*, in general, they do not have any CPS counterpart [14]). We used Sitaram and Felleisen's implementation [34].

[8] At the 1988 Lisp conference, Clinger, Hartheimer, and Ost reported that typically 80% of captured continuations are shared in MacScheme [6].

[9] nqCPS stands for "not quite CPS" (Lee), and is also known as A-normal forms (Sabry & Felleisen), monadic normal forms (Hatcliff & Danvy), and, maybe more plainly, as (higher-order) three-address code.

6 Conclusion

Continuation-based partial evaluation improves over naïve partial evaluation, but it requires skill: skill to program with continuations; and skill to figure out where to use continuations to achieve control-based binding-time improvements. With the shift/reset approach, one puts shift and reset at selected places in a direct-style partial evaluator to get a continuation-based one. CPS transformation automatically yields a purely-functional version of this continuation-based program transformer. This approach is simpler than handwriting large quantities of CPS code, and enables one to experiment with control-based binding-time improvements quickly and decisively.

The operations over continuations to achieve control-based binding-time improvements are not haphazard — they correspond to a precise pattern that can be expressed and reasoned about in direct style, using the control operators shift and reset. These operators are high-level programming constructs. They are used only where necessary, thus preventing low-level programming constructs to occur everywhere.

In summary, we do obtain Similix's continuation-based specializer in the end, automatically (Section 2). This approach suggests a strategy for constructing a continuation-based partial-evaluation compiler (Section 3.2), smoothly. Our partial evaluator illustrates a realistic use of the continuation operators shift and reset (Section 3.1). These control operators enable a more efficient implementation of continuation-based partial evaluation (Section 4).

Acknowledgements

Andrzej Filinski and Karoline Malmkjær provided valuable criticism and encouraging comments. Thanks are also due to Charles Consel for support, and to the referees for perceptive comments. The diagrams of Sections 1.2 and 2.4 were drawn with Kristoffer Rose's XY-pic package.

References

[1] Lennart Beckman, Anders Haraldsson, Östen Oskarsson, and Erik Sandewall. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7(4):319–357, 1976.

[2] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, August 1993.

[3] Anders Bondorf. Improving binding times without explicit CPS-conversion. In Clinger [4], pages 1–10.

[4] William Clinger, editor. *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, San Francisco, California, June 1992. ACM Press.

[5] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[6] William Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for continuations. In Robert (Corky) Cartwright, editor, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 124–131, Snowbird, Utah, July 1988.

[7] Charles Consel. Polyvariant binding-time analysis for applicative languages. In Schmidt [33], pages 66–77.

[8] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In Schmidt [33], pages 145–154.

[9] Charles Consel and Olivier Danvy. From interpreting to compiling binding times. In Neil D. Jones, editor, *Proceedings of the Third European Symposium on Programming*, number 432 in Lecture Notes in Computer Science, pages 88–105, Copenhagen, Denmark, May 1990.

[10] Charles Consel and Olivier Danvy. For a better support of static data flow. In John Hughes, editor, *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 496–519, Cambridge, Massachusetts, August 1991.

[11] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[12] Charles Consel and Siau-Cheng Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993.

[13] Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

[14] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[15] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In Clinger [4], pages 299–310.

[16] Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and Peter Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, January 1988.

[17] Matthias Felleisen. Personal communication, Pittsburgh, Pennsylvania, July 1993.

[18] Andrzej Filinski. Representing monads. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994. ACM Press.

[19] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.

```
(define-syntax reset (syntax-rules () [(_ ?e) (reset-thunk (lambda () ?e))]))

(define-syntax shift (syntax-rules () [(_ ?k ?e) (call/ct (lambda (?k) ?e))]))

(define *meta-continuation* (lambda (v) (error "You forgot the top-level reset...")))

(define abort (lambda (v) (*meta-continuation* v)))

(define reset-thunk (lambda (t)
                      (let ([mc *meta-continuation*])
                        (call/cc (lambda (k)
                                    (begin (set! *meta-continuation* (lambda (v)
                                                                       (begin (set! *meta-continuation* mc)
                                                                              (k v))))
                                           (abort (t)))))))))

(define call/ct (lambda (f)
                  (call/cc (lambda (k)
                             (abort (f (lambda (v)
                                         (reset (k v)))))))))
```

Figure 7: Shift and reset in Scheme

[20] Daniel P. Friedman. Personal communication, e-mail 199309061544.aa14562@daimi.aau.dk, September 1993.

[21] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In Bernard Lang, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 25, No 6, pages 66–77, White Plains, New York, June 1990. ACM Press.

[22] Carsten K. Holst and Carsten K. Gomard. Partial evaluation is fuller laziness. In Paul Hudak and Neil D. Jones, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, pages 223–233, New Haven, Connecticut, June 1991. ACM Press.

[23] Carsten K. Holst and John Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218. Glasgow University, 1991.

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

[25] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[26] Lionello A. Lombardi and Bertram Raphael. Lisp as the language for an incremental computer. In Edmund C. Berkeley and Daniel G. Bobrow, editors, *The Programming Language Lisp: Its Operation and Applications*, pages 204–219, Cambridge, Massachusetts, 1964. The MIT Press.

[27] Karoline Malmkjær. Towards efficient partial evaluation. In Schmidt [33], pages 33–43.

[28] Karoline Malmkjær, Nevin Heintze, and Olivier Danvy. ML partial evaluation using set-based analysis. Technical report CMU-CS-94-129, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1994.

[29] Chethan R. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In Olivier Danvy and Carolyn L. Talcott, editors, *Proceedings of the ACM SIGPLAN Workshop on Continuations*, Technical report STAN-CS-92-1426, Stanford University, pages 49–72, San Francisco, California, June 1992.

[30] Erik Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, Stanford, California, February 1993. Technical report CSL-TR-93-563.

[31] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In Clinger [4], pages 288–298.

[32] David A. Schmidt. Detecting global variables in denotational definitions. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[33] David A. Schmidt, editor. *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, Denmark, June 1993. ACM Press.

[34] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, January 1990.

[35] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1):11–38, May 1988.