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)

January 18, 1995

Abstract

Partial evaluation aims at specializing programs; it is based on call unfolding, constant propagation, and constant folding along the control and data flows of a source program. A continuation represents control during the execution of a program. Continuation-based partial evaluation improves partial evaluation by enhancing the static control flow of the source program.

An offline partial evaluator is staged into binding-time analysis and program specialization. The binding-time analysis classifies source expressions to be static or dynamic. Program specialization amounts to reducing the static expressions and reconstructing the dynamic expressions to form the residual program. In this context, a control-based binding-time improvement affects both the binding-time analysis and the corresponding specializer.

We address continuation-based specialization for Gomard and Jones's offline partial evaluator λ-Mix. We integrate a control-based binding-time improvement into λ-Mix's binding-time analysis and we present two specializers that exploit the increased static information. One uses an explicit representation of control, and is due to Bondorf. The other uses an implicit representation of control and two control operators to delimit and to abstract control. We show that the first is the continuation-passing style (CPS) counterpart of the second. We compare their relative efficiency.

The CPS transformation makes it possible to integrate the binding-time improvement into a partial evaluator by bootstrapping an existing partial evaluator. We show that this relation between implicit and explicit representations of control also scales up to self-applicable partial evaluation and partial-evaluation compilers.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definition and Theory—*syntax*; D.3.2 [**Programming Languages**]: Language Classification—*applicative languages*; D.3.3 [**Programming Languages**]: Languages constructs—*control structures*; *procedures, functions and subroutines*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*functional constructs*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*lambda calculus and related systems*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation*

Additional keywords and phrases: continuation-passing style transformation, control operators, partial evaluation.

*Technical Report CS-95-178, Computer Science Department, Brandeis University, Waltham, Massachusetts. An earlier version of this paper appeared in the proceedings of the ACM SIGPLAN Symposium on LISP and Functional Programming, June 27-29 1994, Orlando, Florida.

†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 and during a 10-days visit to Aarhus 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. This work is supported by the DART project (Design, Analysis and Reasoning about Tools) of the Danish Research Councils.

Contents

1 Introduction **4**
- 1.1 Partial evaluation . . . 4
- 1.2 Binding times . . . 4
- 1.3 Binding-time improvements . . . 5
- 1.4 Control-based binding-time improvements . . . 5
- 1.5 Integration of control-based binding-time improvements in a partial evaluator . . . 6
- 1.6 Summary . . . 8
- 1.7 Offline partial evaluation . . . 9
- 1.8 Overview . . . 10

2 The starting point: λ-mix **10**

3 Binding-time analysis for continuation-based partial evaluation **12**

4 Continuation-style, continuation-based specialization **12**
- 4.1 Integrating the binding-time improvement into the specializer . . . 12
- 4.2 Well-behaved continuations . . . 13
- 4.3 A "well-behaved" continuation-style specializer . . . 14

5 Direct-style, continuation-based specialization **14**
- 5.1 The control operators shift and reset . . . 14
  - 5.1.1 Example . . . 15
  - 5.1.2 Implementations of shift and reset . . . 15
- 5.2 Continuation-based specialization using shift and reset . . . 16
- 5.3 Correctness of the direct-style, continuation-based specializer . . . 16

6 Other control-based binding-time improvements **17**
- 6.1 Dynamic applications . . . 17
- 6.2 Dynamic conditional expressions . . . 18

7 Implementation issues **19**
- 7.1 Bootstrapping . . . 19
- 7.2 Measures . . . 20
  - 7.2.1 An experiment . . . 20
  - 7.2.2 Comparing the direct-style and the continuation-style specializers . . . 20
  - 7.2.3 Assessing the cost of the improvement . . . 20
  - 7.2.4 Iterative programs . . . 22
  - 7.2.5 An alternative implementation . . . 22
  - 7.2.6 Side issues . . . 22
- 7.3 Conclusion . . . 22

8 Self-applicable partial evaluation **23**
- 8.1 Background . . . 23
- 8.2 Offline partial evaluation . . . 24
- 8.3 Handwriting pecom . . . 24
- 8.4 Continuation-style, continuation-based partial-evaluation compilers . . . 25
- 8.5 Direct-style, continuation-based partial-evaluation compilers . . . 25
- 8.6 Summary . . . 27

9 Related work **27**

10 Conclusion **27**

A Continuation-based partial evaluation in Standard ML **28**

List of Figures

1 The source and target language of λ-terms . . . 11
2 The language of annotated λ-terms . . . 11
3 Binding-time analysis for the λ-calculus with let . . . 11
4 Direct-style specializer . . . 11
5 Improved binding-time analysis rule for dynamic let expressions . . . 11
6 Continuation-style specializer . . . 13
7 Continuation-style, continuation-based specializer . . . 14
8 The CPS transformation . . . 15
9 Direct-style, continuation-based specializer . . . 16
10 Improved binding-time analysis and specialization for beta-redexes . . . 18
11 Binding-time analysis and specialization for conditional expressions . . . 19
12 Improved binding-time analysis and specialization for conditional expressions . . . 19
13 Relative costs of specializing direct-style and CPS pattern-matching programs with respect to several patterns . . . 21
14 Continuation-style, continuation-based pecom . . . 25
15 Direct-style, continuation-based pecom . . . 26
16 The source and target language of λ-terms and the language of annotated λ-terms in Standard ML (Figures 1 and 2) . . . 29
17 The example, in the source language . . . 30
18 The example, annotated by $\mathcal{A}$ — `exp` . . . 30
19 The result of specializing `exp` . . . 30
20 The example, annotated by $\mathcal{A}'$ — `exp'` . . . 31
21 The result of specializing `exp'` . . . 31
22 Intermediate values during specialization . . . 32
23 Environment module . . . 32
24 Generator of fresh variables . . . 32
25 Direct-style specializer in Standard ML (Figure 4) . . . 33
26 Continuation-style specializer in Standard ML (Figure 6) . . . 34
27 Direct-style, continuation-based specializer in Standard ML (Figure 9) . . . 35
28 Continuation-passing, continuation-based specializer in Standard ML (Figure 7) . . . 36
29 Direct-style, continuation-based pecom in Standard ML (Figure 15) . . . 37
30 Continuation-style, continuation-based pecom in Standard ML (Figure 14) . . . 38
31 The CPS transformation in Standard ML (Figure 8) . . . 39
32 Shift and reset in Standard ML of New Jersey . . . 40

1 Introduction

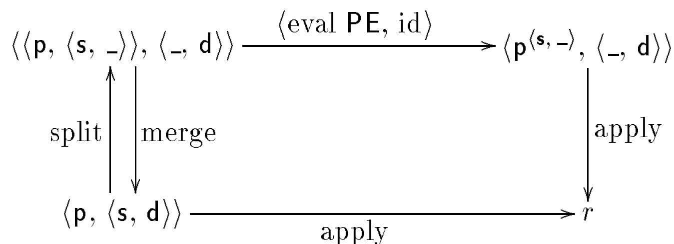
Partial evaluation is a program-transformation technique for specializing programs [17, 38]. It was developed in the sixties and seventies [4, 44], and drastically simplified in the eighties for purposes of self-application [39]. Today partial evaluation is evolving both quantitatively and qualitatively. Quantitatively, partial evaluators handle more and more programming-language features — data structures, higher-order procedures, evaluation orders, types, and so on. Qualitatively, these features need to be handled effectively.

Binding-time improvements [38, Chap. 12] aim at making a partial evaluator yield better results. At first, binding-time improvements are achieved mostly by hand-transforming source programs prior to partial evaluation. As they are better understood, these hand-transformations are progressively integrated into partial evaluators, thereby alleviating the need for source-level binding-time improvements. This paper addresses control-based binding-time improvements.

1.1 Partial evaluation

Given a source program p, let us (arbitrarily) split its input into a pair $\langle \mathsf{s}, \mathsf{d} \rangle$. Partial evaluation of p with respect to the static input $\langle \mathsf{s}, _ \rangle$ yields the specialized program $\mathsf{p}^{\langle \mathsf{s}, _ \rangle}$, under a definitional condition that paraphrases Kleene's S^m_n-theorem [40]: applying this specialized program to the remaining input $\langle _, \mathsf{d} \rangle$ should yield the same result as applying p to the complete input $\langle \mathsf{s}, \mathsf{d} \rangle$ — provided that the source program p, the partial evaluator PE, and the residual program $\mathsf{p}^{\langle \mathsf{s}, _ \rangle}$ all terminate.

This definitional condition is captured in the following diagram:



where eval is a curried version of apply.

$$\begin{aligned} \text{eval} &: \text{Program} \rightarrow \text{Input} \rightarrow \text{Output} \\ \text{apply} &: \text{Program} \times \text{Input} \rightarrow \text{Output} \end{aligned}$$

These two functions make it possible to distinguish between a function and the program implementing the function. (eval p) denotes the function implemented by the program p. (apply p i) applies the program p to the input i.

1.2 Binding times

The notion of binding time arises naturally in partial evaluation since source programs are evaluated in two stages: at partial-evaluation time (*i.e.*, in the top arrow of the diagram) and at run time (*i.e.*, in the right arrow of the diagram). Both values and expressions have binding times. A value that is available to the partial evaluator has binding time "static". A value that is not available until run time has binding time "dynamic". An expression that can be reduced at partial evaluation time, given the available static values, is referred to as a "static expression". An expression that must be evaluated at run time, because it

depends on dynamic values, is referred to as a "dynamic expression". A partial evaluator reduces the static expressions and reconstructs the dynamic expressions to produce a residual program.

1.3 Binding-time improvements

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

$$x + (y - 1)$$

where x has a static value and y has a dynamic value. A naïve partial evaluator 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 partial evaluator would now consider the subtraction to be static (since x is known at partial-evaluation time and 1 is an immediate constant) and the addition to be dynamic (since y is not known until run time). By rewriting the expression, we achieve a binding-time improvement: the same partial evaluator considers more expressions as static and thus yields a more specialized program. Overall, the same partial evaluator yields a better result.

1.4 Control-based binding-time improvements

A few years ago, Consel and Danvy identified a "binding-time bottleneck" in source programs, leading to poor binding-time properties and thus to poor specialization [16]. The bottleneck occurs when a subexpression of a dynamic expression yields a value that is not dynamic, and this value is the value of the entire expression. Consider

$$1 + (\text{let } x = d \text{ in } 3)$$

where d has a dynamic value. The let expression must be residualized.[1] 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. Propagating the addition of 1 inward:

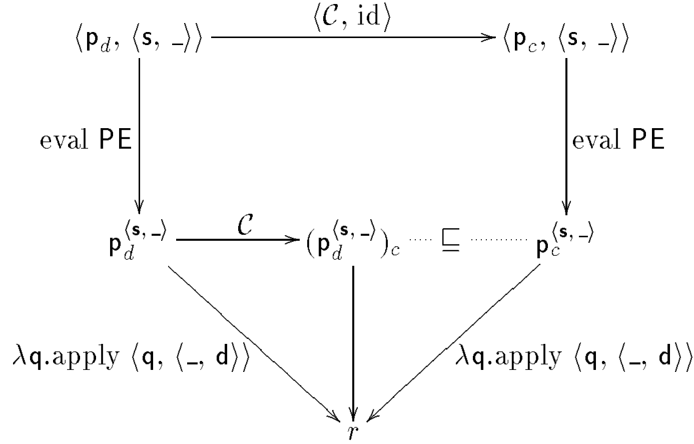
$$\text{let } x = d \text{ in } (1 + 3)$$

allows more specialization.

Consel and Danvy also observe that this bottleneck is absent in continuation-passing style (CPS) programs [57]. Thus it is simple to get better result with a (higher-order) partial evaluator: one should just CPS-transform the source program.

The situation is summarized in the following diagram.

[1] 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). Following practice [7], we assume that let expressions have been inserted during preprocessing so that dynamic expressions are not duplicated or discarded when λ-abstractions are applied statically.



p_d denotes a direct-style source program, PE denotes the source code of a partial evaluator, $\mathcal{C}$ denotes the CPS transformation, and $\sqsubseteq$ denotes the relation "is less specialized than". The diagram can be read as follows.

- Specializing p_d with respect to $\langle \mathsf{s}, _ \rangle$ yields $\mathsf{p}_d^{\langle \mathsf{s}, _ \rangle}$. CPS-transforming this residual program yields $(\mathsf{p}_d^{\langle \mathsf{s}, _ \rangle})_c$.
- CPS-transforming p_d yields p_c. Specializing p_c with respect to $\langle \mathsf{s}, _ \rangle$ yields $\mathsf{p}_c^{\langle \mathsf{s}, _ \rangle}$.
- $(\mathsf{p}_d^{\langle \mathsf{s}, _ \rangle})_c$ is less specialized than $\mathsf{p}_c^{\langle \mathsf{s}, _ \rangle}$.
- Applying $\mathsf{p}_d^{\langle \mathsf{s}, _ \rangle}$, $(\mathsf{p}_d^{\langle \mathsf{s}, _ \rangle})_c$, and $\mathsf{p}_c^{\langle \mathsf{s}, _ \rangle}$ to the remaining input yields the same result as applying p to the complete input $\langle \mathsf{s}, \mathsf{d} \rangle$.

Consel and Danvy also note that if a source program is tail-recursive (*e.g.*, in CPS), CPS-transforming it does not provide new opportunities for control-based binding-time improvements [16, Prop. 1].

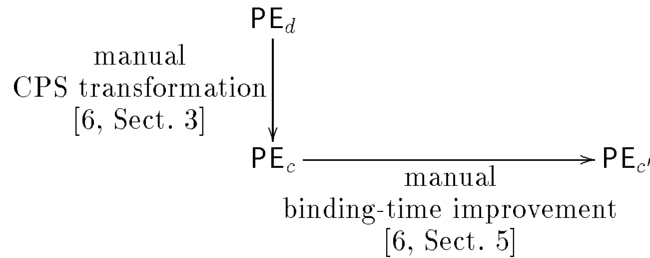
In summary, source programs often require a control-based binding-time improvement: relocating the context of a let expression to the subexpression that determines the value of this let expression. The CPS transformation enables this improvement by propagating the continuation of a let expression to the body of this expression. It also provides an inter-procedural improvement when a procedure is applied to a static continuation.

1.5 Integration of control-based binding-time improvements in a partial evaluator

Bondorf observes that the partial evaluation of a CPS program costs more than the partial evaluation of the corresponding direct-style program [6]. Furthermore, when the source program is CPS-transformed before partial evaluation, both the residual program and any annotated versions of the source program presented to the user as a debugging aid would have to be transformed back to direct style. Thus, to benefit from the control-based binding-time improvement while keeping partial evaluation efficient and tractable, Bondorf proposes to integrate the binding-time improvement into the partial evaluator. This integration is done by implementing the partial evaluator using explicit continuations.

Bondorf develops such a partial evaluator in two steps. First, the direct-style partial evaluator (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 to bypass the bottleneck, thus allowing static values to reach their

consumer statically ($\mathsf{PE}_{c'}$). Bondorf's approach is characterized by the following diagram, and documented further in Chapter 10 of Jones, Gomard, and Sestoft's textbook on partial evaluation [38].



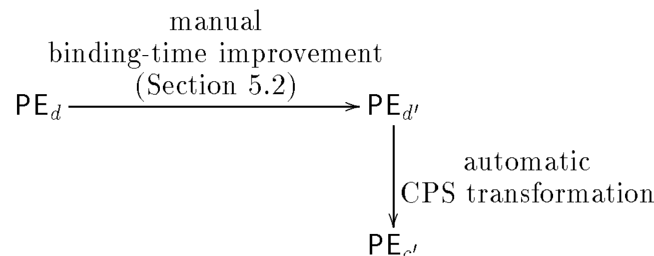
The resulting partial evaluator achieves inter-procedural improvements when a procedure call is unfolded statically. Because, however, access to continuations is built into the partial evaluator, significant strategic changes are required to achieve all the improvements enabled by source CPS transformation [6, Sect. 6.4]. This aspect is analyzed elsewhere [45]. Also, as mentioned above, Consel and Danvy observe that transforming a tail-recursive program (such as a CPS program) into CPS gives no extra binding-time improvement. Since $\mathsf{PE}_{c'}$ results from integrating Consel and Danvy's bottleneck elimination into the partial evaluator, it naturally follows that both the improved ($\mathsf{PE}_{c'}$) and the unimproved (PE_d and PE_c) partial evaluators give the same result on tail-recursive programs.

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 1). It is even necessary in Similix [6, Sect. 7], where it enables the treatment of partially static values [38].

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

- the partial evaluator is CPS-transformed by hand;
- its continuations need skillful massage — the more complicated the partial evaluator, the more skill is required; and
- the resulting partial evaluator 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-style, continuation-based partial evaluator $\mathsf{PE}_{c'}$ precisely correspond to the effect of Danvy and Filinski's control operators shift and reset [18, 19] in direct style, and that inserting shift and reset at a few selected places in a direct-style partial evaluator ($\mathsf{PE}_{d'}$) and then converting $\mathsf{PE}_{d'}$ into CPS automatically yields a continuation-style, continuation-based partial evaluator ($\mathsf{PE}_{c'}$). Both the source program and the partial evaluator remain in the familiar direct style. Our approach is characterized by the following diagram:

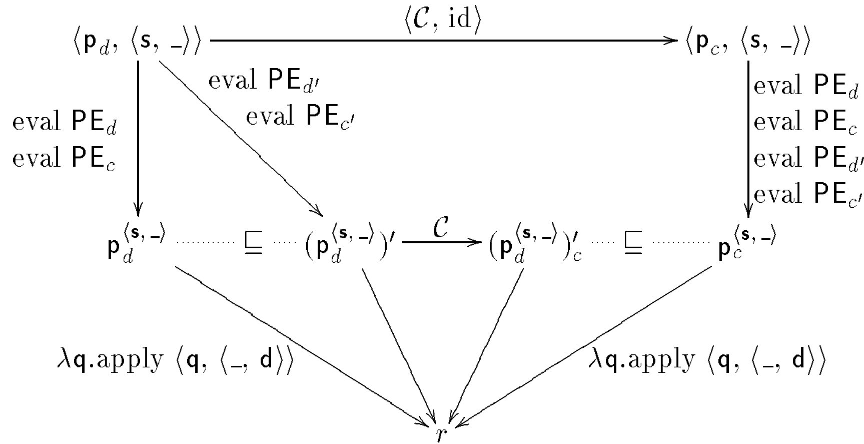


The transformation from $\mathsf{PE}_{d'}$ to $\mathsf{PE}_{c'}$ is not an issue here, since it can be carried out automatically, as indeed the transformation from PE_d to PE_c in Bondorf's work could have been. We express the control-based binding-time improvements at the source (direct-style) level, in essence "improving binding times without explicit CPS-conversion", neither of the source programs nor of the partial evaluator. 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 direct implementation of shift and reset [26].

We also observed that if a source program is already in CPS, there are no opportunities for control-based binding-time improvements and thus PE_d, PE_c, $\mathsf{PE}_{c'}$, and $\mathsf{PE}_{d'}$ all yield the same residual program [43].

1.6 Summary

The relationship among the partial evaluators is summarized in the following diagram.



Again, p_d denotes a direct-style source program and p_c denotes its CPS counterpart. $\mathcal{C}$ denotes the CPS transformation. PE_d denotes the source code of a direct-style partial evaluator and PE_c denotes its CPS counterpart. $\mathsf{PE}_{d'}$ denotes the source code of a direct-style, continuation-based partial evaluator and $\mathsf{PE}_{c'}$ denotes its CPS counterpart. Finally, $\sqsubseteq$ denotes the relation "is less specialized than". The diagram can be read as follows.

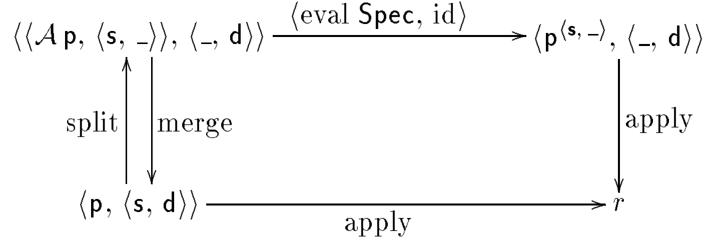
- Specializing p_d with respect to $\langle \mathsf{s}, _ \rangle$ using PE_d or using PE_c yields $\mathsf{p}_d^{\langle \mathsf{s}, _ \rangle}$.
- Specializing p_d with respect to $\langle \mathsf{s}, _ \rangle$ using $\mathsf{PE}_{d'}$ or using $\mathsf{PE}_{c'}$ yields $(\mathsf{p}_d^{\langle \mathsf{s}, _ \rangle})'$.
- $\mathsf{p}_d^{\langle \mathsf{s}, _ \rangle}$ is less specialized than $(\mathsf{p}_d^{\langle \mathsf{s}, _ \rangle})'$.
- CPS-transforming p_d yields p_c and CPS-transforming $(\mathsf{p}_d^{\langle \mathsf{s}, _ \rangle})'$ yields $(\mathsf{p}_d^{\langle \mathsf{s}, _ \rangle})'_c$.
- Specializing p_c with respect to $\langle \mathsf{s}, _ \rangle$ yields $\mathsf{p}_c^{\langle \mathsf{s}, _ \rangle}$, no matter which partial evaluator is used.
- $(\mathsf{p}_d^{\langle \mathsf{s}, _ \rangle})'_c$ is less specialized than $\mathsf{p}_c^{\langle \mathsf{s}, _ \rangle}$.
- Applying $\mathsf{p}_d^{\langle \mathsf{s}, _ \rangle}$, $(\mathsf{p}_d^{\langle \mathsf{s}, _ \rangle})'$, $(\mathsf{p}_d^{\langle \mathsf{s}, _ \rangle})'_c$, and $\mathsf{p}_c^{\langle \mathsf{s}, _ \rangle}$ to the remaining input yields the same result as applying p to the complete input $\langle \mathsf{s}, \mathsf{d} \rangle$.

1.7 Offline partial evaluation

In practice, distinguishing between static and dynamic values incurs a definite interpretive overhead in a partial evaluator. This overhead is limited by analyzing the binding times of a source program prior to its actual specialization. This strategy is known as *offline* partial evaluation [17, 38]. An offline partial evaluator is staged into a binding-time analysis $\mathcal{A}$ and a specializer Spec (implementing a function $\mathcal{S}$):

$$\begin{aligned} \text{eval } \mathsf{PE} &= (\text{eval } \mathsf{Spec}) \circ \mathcal{A} \\ &= \mathcal{S} \circ \mathcal{A} \end{aligned}$$

The binding-time analysis identifies whether each term is static or dynamic. The specializer processes each term following these annotations. The situation is summarized in the following diagram.

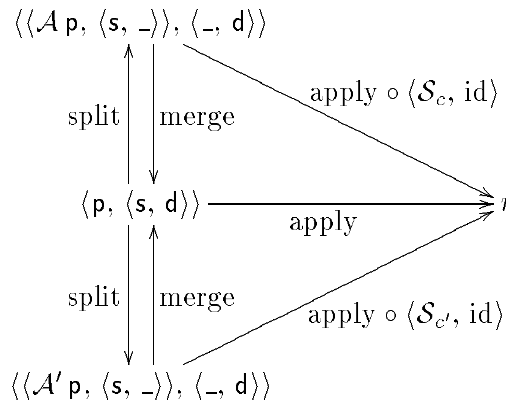


Because offline partial evaluation is staged into a binding-time analysis $\mathcal{A}$ and a specializer $\mathcal{S}$, the integration of a binding-time improvement into an offline partial evaluator must be staged as well. Continuation-based partial evaluation relies on an improved binding-time analysis $\mathcal{A}'$ that classifies more values to be static. The corresponding specializer uses continuations to bypass the binding-time bottleneck, thus allowing these values that are newly typed static to reach their consumer statically. The specializer may represent control explicitly ($\mathsf{PE}_{c'}$) or implicitly ($\mathsf{PE}_{d'}$).

$$\begin{cases} \mathsf{PE}_d &= (\text{eval } \mathsf{Spec}_d) \circ \mathcal{A} &= \mathcal{S}_d \circ \mathcal{A} \\ \mathsf{PE}_c &= (\text{eval } \mathsf{Spec}_c) \circ \mathcal{A} &= \mathcal{S}_c \circ \mathcal{A} \\ \mathsf{PE}_{c'} &= (\text{eval } \mathsf{Spec}_{c'}) \circ \mathcal{A}' &= \mathcal{S}_{c'} \circ \mathcal{A}' \\ \mathsf{PE}_{d'} &= (\text{eval } \mathsf{Spec}_{d'}) \circ \mathcal{A}' &= \mathcal{S}_{d'} \circ \mathcal{A}' \end{cases}$$

where Spec_c and $\mathsf{Spec}_{c'}$ are the CPS counterparts of Spec_d and $\mathsf{Spec}_{d'}$, respectively.

Again, the naïve partial evaluator and the improved one must construct specialized programs that yield the same result on the dynamic input



1.8 Overview

A control-based binding-time improvement amounts to relocating contexts. This relocation is conveniently achieved with continuations, since continuations represent contexts (as functions). The binding-time improvement can either be achieved in the source program (Section 1.4) or be integrated in the partial evaluator (Section 1.5). In an offline partial evaluator, it is integrated into the binding-time analysis and the specializer (Section 1.7). The specializer may use either an explicit ($\mathsf{Spec}_{c'}$) or an implicit ($\mathsf{Spec}_{d'}$) representation of control.

The rest of this paper is organized along these lines. We start with a simplified λ-mix, an offline partial evaluator for the lambda-calculus extended with let expressions (Section 2). We present a binding-time analysis and a corresponding specialization function. We integrate a control-based binding-time improvement in the binding-time analysis (Section 3). We present a first specializer with an explicit representation of control (Section 4) and a second specializer with an implicit representation of control, using the control operators shift and reset (Section 5). The first specializer is due to Bondorf [6]. The second can be automatically CPS-transformed into the first. Thus the binding-time improvement is orthogonal to the choice of whether to express the specializer in direct or continuation-passing style.

The first author used this method to bootstrap a new release of Consel's partial evaluator, Schism [14]. Using Filinski's direct implementation of shift and reset [26], we have observed that the second specializer is actually more efficient than the first (Section 7).

The methodology scales up to self-applicable partial evaluation (Section 8). The result of applying a partial evaluator to another partial evaluator is a partial-evaluation compiler, *i.e.*, a specialized partial evaluator generating specialized partial evaluators. Since direct style and continuation-passing style are orthogonal to the binding-time improvement, there are 16 partial-evaluation compilers. We exhibit a direct-style, continuation-based partial-evaluation compiler: composing it with the CPS transformation and CPS-transforming the resulting transformation automatically yields Bondorf and Dussart's continuation-style, continuation-based partial-evaluation compiler [8].[2] After a comparison with related work (Section 9), we conclude (Section 10). The Standard ML code of all the figures is presented in an appendix.

2 The starting point: λ-mix

Our starting point is the core of λ-mix, a simple offline partial evaluator for the language of call-by-value λ-terms, extended with let expressions [31, 32, 37]. We focus on this simple language for conciseness, however the techniques extend straightforwardly to a more realistic source language [6, 38, 43].

The source and target language of λ-mix is shown in Figure 1. The binding-time analysis (Figures 2 and 3) assigns each term an annotated term e' and a type τ. The annotation describes the binding time of the term itself: a static term is unannotated, while a dynamic term is underlined. The type describes the binding time of the value of the term: the type is a function type if the value is static, and d if the value is dynamic. (In a language with a richer set of constants and primitive operators, we would have a richer set of types associated with static values.) The types of subexpressions influence the annotation of enclosing expressions. In particular, an application expression must be annotated dynamic (underlined) when the value of the function position is dynamic, and a let expression must be annotated dynamic when the value of the named expression is dynamic (as described in Footnote 1). All subexpressions of a dynamic expression

[2] A partial-evaluation compiler can be generated by self-application (Section 8) but like Bondorf and Dussart, we wrote ours by hand, directly.

$$e \quad ::= \quad i \mid \lambda\, i\,.\, e \mid e_0\, e_1 \mid \text{let } i = e_1 \text{ in } e_2$$

Figure 1: The source and target language of λ-terms

$$e \quad ::= \quad i \mid \lambda\, i\,.\, e \mid e_0 \,@\, e_1 \mid \text{let } i = e_1 \text{ in } e_2 \mid \underline{\lambda}\, i\,.\, e \mid e_0 \,\underline{@}\, e_1 \mid \underline{\text{let}}\ i = e_1\ \underline{\text{in}}\ e_2$$

Figure 2: The language of annotated λ-terms

$$\tau \quad ::= \quad \tau_1 \rightarrow \tau_2 \mid \mathsf{d}$$

$$\frac{\Gamma(i) = \tau}{\Gamma \vdash i \ \triangleright\ i : \tau}$$

$$\frac{\Gamma, i : \tau_1 \vdash e \ \triangleright\ e' : \tau_2}{\Gamma \vdash \lambda\, i\,.\, e \ \triangleright\ \lambda\, i\,.\, e' : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma, i : \mathsf{d} \vdash e \ \triangleright\ e' : \mathsf{d}}{\Gamma \vdash \lambda\, i\,.\, e \ \triangleright\ \underline{\lambda}\, i\,.\, e' : \mathsf{d}}$$

$$\frac{\Gamma \vdash e_0 \ \triangleright\ e_0' : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_1 \ \triangleright\ e_1' : \tau_1}{\Gamma \vdash e_0\, e_1 \ \triangleright\ e_0' \,@\, e_1' : \tau_2} \qquad \frac{\Gamma \vdash e_0 \ \triangleright\ e_0' : \mathsf{d} \qquad \Gamma \vdash e_1 \ \triangleright\ e_1' : \mathsf{d}}{\Gamma \vdash e_0\, e_1 \ \triangleright\ e_0' \,\underline{@}\, e_1' : \mathsf{d}}$$

$$\frac{\Gamma \vdash e_1 \ \triangleright\ e_1' : \tau_1 \qquad \Gamma, i : \tau_1 \vdash e_2 \ \triangleright\ e_2' : \tau_2}{\Gamma \vdash \text{let } i = e_1 \text{ in } e_2 \ \triangleright\ \text{let } i = e_1' \text{ in } e_2' : \tau_2} \text{if } \tau_1 \neq \mathsf{d} \qquad \frac{\Gamma \vdash e_1 \ \triangleright\ e_1' : \mathsf{d} \qquad \Gamma, i : \mathsf{d} \vdash e_2 \ \triangleright\ e_2' : \mathsf{d}}{\Gamma \vdash \text{let } i = e_1 \text{ in } e_2 \ \triangleright\ \underline{\text{let}}\ i = e_1'\ \underline{\text{in}}\ e_2' : \mathsf{d}}$$

Figure 3: Binding-time analysis for the λ-calculus with let

$$\begin{array}{rcl}
\mathcal{S}_d\,[\![i]\!]\,\rho & = & \rho\, i \\
\mathcal{S}_d\,[\![\lambda\, i\,.\, e]\!]\,\rho & = & \lambda\, w\,.\, \mathcal{S}_d\,[\![e]\!]\,\rho[i \mapsto w] \\
\mathcal{S}_d\,[\![e_0 \,@\, e_1]\!]\,\rho & = & (\mathcal{S}_d\,[\![e_0]\!]\,\rho)\,(\mathcal{S}_d\,[\![e_1]\!]\,\rho) \\
\mathcal{S}_d\,[\![\text{let } i = e_1 \text{ in } e_2]\!]\,\rho & = & \mathcal{S}_d\,[\![e_2]\!]\,\rho[i \mapsto \mathcal{S}_d\,[\![e_1]\!]\,\rho] \\
\mathcal{S}_d\,[\![\underline{\lambda}\, i\,.\, e]\!]\,\rho & = & \text{let } m = \text{fresh}() \text{ in } \underline{\lambda}\, m\,.\, \mathcal{S}_d\,[\![e]\!]\,\rho[i \mapsto \underline{m}] \\
\mathcal{S}_d\,[\![e_0 \,\underline{@}\, e_1]\!]\,\rho & = & (\mathcal{S}_d\,[\![e_0]\!]\,\rho)\,\underline{@}\,(\mathcal{S}_d\,[\![e_1]\!]\,\rho) \\
\mathcal{S}_d\,[\![\underline{\text{let}}\ i = e_1\ \underline{\text{in}}\ e_2]\!]\,\rho & = & \text{let } m = \text{fresh}() \text{ in } \underline{\text{let}}\ m = \mathcal{S}_d\,[\![e_1]\!]\,\rho\ \underline{\text{in}}\ \mathcal{S}_d\,[\![e_2]\!]\,\rho[i \mapsto \underline{m}]
\end{array}$$

Figure 4: Direct-style specializer

$$\frac{\Gamma \vdash e_1 \ \triangleright\ e_1' : \mathsf{d} \qquad \Gamma, i : \mathsf{d} \vdash e_2 \ \triangleright\ e_2' : \tau_2}{\Gamma \vdash \text{let } i = e_1 \text{ in } e_2 \ \triangleright\ \underline{\text{let}}\ i = e_1'\ \underline{\text{in}}\ e_2' : \tau_2}$$

Figure 5: Improved binding-time analysis rule for dynamic let expressions

must have a dynamic value, although they need not be dynamic expressions.

The specializer is applied to the annotated term produced by the binding-time analysis. The correctness of the binding-time analysis ensures that the annotations are consistent [36, 49, 50]. The specializer appears in Figure 4. In the definition of the specializer, fresh is a primitive operator that returns fresh identifiers, to avoid name clashes in residual programs. On the right-hand side of the equations, underlined terms construct residual code.

3 Binding-time analysis for continuation-based partial evaluation

In Figure 3, the rule for let expressions specifies that if a let-bound expression is dynamic, then the whole let expression is dynamic. This classification is excessive when the body of the let expression is not dynamic and yields a value that could be consumed by the context of the let expression. Thus the impact on the binding-time analysis of the binding-time improvement introduced in Section 1.4 can be stated as follows: the binding-time type of the value of a let expression is the binding-time type of the value of its body, regardless of whether the let expression needs to be reconstructed. Figure 5 displays the modified rule for dynamic let expressions. The rest of the binding-time analysis remains the same (Figure 3).

For example, when d is dynamic, the let expression must be residualized in the following expression (see Footnote 1):

$$(\text{let}\ x = d\ \text{in}\ \lambda\, a\,.\, a)\,(\lambda\, b\,.\, b)$$

The standard binding-time analysis $\mathcal{A}$ (Figure 3) classifies the body of the let expression, the application, and its argument to be residualized:

$$(\underline{\text{let}}\ x = d\ \underline{\text{in}}\ \underline{\lambda}\, a\,.\, a)\ \underline{@}\ (\underline{\lambda}\, b\,.\, b)$$

Since everything is dynamic in this two-level expression, specializing it yields the original expression. The binding-time analysis for the control-based binding-time improvement, $\mathcal{A}'$ (Figure 5), classifies the body of the let expression and thus the application as static:

$$(\underline{\text{let}}\ x = d\ \underline{\text{in}}\ \lambda\, a\,.\, a)\ @\ (\underline{\lambda}\, b\,.\, b)$$

Specializing this two-level expression yields the following residual expression, which is more specialized than the original expression:

$$\text{let}\ x = d\ \text{in}\ \lambda\, b\,.\, b$$

This example is particularly simple. In practice, such a dynamic let expression with a non-dynamic body may appear after unfolding function calls, *e.g.*, if the actual parameter is dynamic [7].

The following two sections investigate possible integrations of this binding-time improvement into the specializer.

4 Continuation-style, continuation-based specialization

4.1 Integrating the binding-time improvement into the specializer

The binding-time improvement for dynamic let expressions relies on access to the context of the let expression. To make the context explicitly available within the specializer, Bondorf first transforms the specializer into CPS (see Figure 6). The centerpiece of continuation-based specialization lies in the specialization of

$$
\begin{aligned}
\mathcal{S}_c\,[\![i]\!]\,\rho\,\kappa &= \kappa\,(\rho\,i)\\
\mathcal{S}_c\,[\![\lambda\,i\,.\,e]\!]\,\rho\,\kappa &= \kappa\,(\lambda\,w\,.\,\lambda\,k\,.\,\mathcal{S}_c\,[\![e]\!]\,\rho[i\mapsto w]\,k)\\
\mathcal{S}_c\,[\![e_0\,@\,e_1]\!]\,\rho\,\kappa &= \mathcal{S}_c\,[\![e_0]\!]\,\rho\,\lambda\,v_0\,.\,\mathcal{S}_c\,[\![e_1]\!]\,\rho\,\lambda\,v_1\,.\,(v_0\,v_1)\,\kappa\\
\mathcal{S}_c\,[\![\text{let}\ i=e_1\ \text{in}\ e_2]\!]\,\rho\,\kappa &= \mathcal{S}_c\,[\![e_1]\!]\,\rho\,\lambda\,v\,.\,\mathcal{S}_c\,[\![e_2]\!]\,\rho[i\mapsto v]\,\kappa\\
\mathcal{S}_c\,[\![\underline{\lambda}\,i\,.\,e]\!]\,\rho\,\kappa &= \text{let}\ m=\text{fresh}()\ \text{in}\ \mathcal{S}_c\,[\![e]\!]\,\rho[i\mapsto\underline{m}]\,\lambda\,v\,.\,\kappa\,(\underline{\lambda}\,m\,.\,v)\\
\mathcal{S}_c\,[\![e_0\,\underline{@}\,e_1]\!]\,\rho\,\kappa &= \mathcal{S}_c\,[\![e_0]\!]\,\rho\,\lambda\,v_0\,.\,\mathcal{S}_c\,[\![e_1]\!]\,\rho\,\lambda\,v_1\,.\,\kappa\,(v_0\,\underline{@}\,v_1)\\
\mathcal{S}_c\,[\![\underline{\text{let}}\ i=e_1\ \underline{\text{in}}\ e_2]\!]\,\rho\,\kappa &= \text{let}\ m=\text{fresh}()\ \text{in}\ \mathcal{S}_c\,[\![e_1]\!]\,\rho\,\lambda\,v\,.\,\mathcal{S}_c\,[\![e_2]\!]\,\rho[i\mapsto\underline{m}]\,\lambda\,w\,.\,\kappa\,(\underline{\text{let}}\ m=v\ \underline{\text{in}}\ w)
\end{aligned}
$$

Figure 6: Continuation-style specializer

a reconstructed let expression: the continuation is propagated to the specialization of the body of the let expression. The rest of the specializer remains the same (Figure 6).

$$
\mathcal{S}_{c'}\,[\![\underline{\text{let}}\ i=e_1\ \underline{\text{in}}\ e_2]\!]\,\rho\,\kappa \quad=\quad \text{let}\ m=\text{fresh}()\ \text{in}\ \mathcal{S}_{c'}\,[\![e_1]\!]\,\rho\,\lambda\,v\,.\,\underline{\text{let}}\ m=v\ \underline{\text{in}}\ \mathcal{S}_{c'}\,[\![e_2]\!]\,\rho[i\mapsto\underline{m}]\,\kappa
$$

This improvement is correct if the continuation of the specializer corresponds to the run-time continuation.

4.2 Well-behaved continuations

Let us consider some of the continuations created during specialization.

When specializing the function position of a dynamic application, the continuation first specializes the argument and then reconstructs the application. These correspond to the actions taken when evaluating an application. Thus this specializer continuation corresponds to the continuation of the evaluation of the function position.

When specializing the body of a dynamic λ-abstraction, the continuation reconstructs the λ-abstraction. During evaluation, however, the body of a λ-abstraction is evaluated *after* the value of the λ-abstraction is created, if at all. In particular it is not evaluated until the value of the λ-abstraction is applied. Thus this specializer continuation does not correspond to the continuation of the evaluation of the body.

When specializing the body of a dynamic let expression, the continuation first reconstructs the let expression. During evaluation, however, the value of the body is directly sent to the continuation of the let expression. Thus this specializer continuation does not correspond to the continuation of the evaluation of the body.

A continuation in the specializer that corresponds to a continuation in the evaluator is said to be well-behaved [6, 38]:

Definition 1 *A continuation κ in the specializer is* well-behaved *iff it places its argument in a residual context such that when the residual program runs, this argument is evaluated in the current lexical environment before the overall residual expression yields a value.* □

Here is the grammar of these "safe" residual contexts:

$$
s[\,]\quad ::= \quad [\,]\ \mid\ \text{let}\ i=s[\,]\ \text{in}\ e_2\ \mid\ (s[\,])\,(e_1)\ \mid\ e_0\,(s[\,])
$$

Filling the hole of a safe context with an expression e yields an expression e' such that both e and e' will be evaluated but e will yield a value before e' does.

$$
\begin{array}{rcl}
\mathcal{S}_{c'}\,[\![i]\!]\,\rho\,\kappa & = & \kappa\,(\rho\,i) \\
\mathcal{S}_{c'}\,[\![\lambda\,i\,.\,e]\!]\,\rho\,\kappa & = & \kappa\,(\lambda\,w\,.\,\mathcal{S}_{c'}\,[\![e]\!]\,\rho[i\mapsto w]) \\
\mathcal{S}_{c'}\,[\![e_0\;@\;e_1]\!]\,\rho\,\kappa & = & \mathcal{S}_{c'}\,[\![e_0]\!]\,\rho\,\lambda\,v_0\,.\,\mathcal{S}_{c'}\,[\![e_1]\!]\,\rho\,\lambda\,v_1\,.\,(v_0\,v_1)\,\kappa \\
\mathcal{S}_{c'}\,[\![\text{let}\;i=e_1\;\text{in}\;e_2]\!]\,\rho\,\kappa & = & \mathcal{S}_{c'}\,[\![e_1]\!]\,\rho\,\lambda\,v\,.\,\mathcal{S}_{c'}\,[\![e_2]\!]\,\rho[i\mapsto v]\,\kappa \\
\mathcal{S}_{c'}\,[\![\underline{\lambda}\,i\,.\,e]\!]\,\rho\,\kappa & = & \text{let}\;m=\text{fresh}()\;\text{in}\;\kappa\,(\underline{\lambda}\,m\,.\,\mathcal{S}_{c'}\,[\![e]\!]\,\rho[i\mapsto\underline{m}]\,\lambda\,v\,.\,v) \\
\mathcal{S}_{c'}\,[\![e_0\;\underline{@}\;e_1]\!]\,\rho\,\kappa & = & \mathcal{S}_{c'}\,[\![e_0]\!]\,\rho\,\lambda\,v_0\,.\,\mathcal{S}_{c'}\,[\![e_1]\!]\,\rho\,\lambda\,v_1\,.\,\kappa\,(v_0\;\underline{@}\;v_1) \\
\mathcal{S}_{c'}\,[\![\underline{\text{let}}\;i=e_1\;\underline{\text{in}}\;e_2]\!]\,\rho\,\kappa & = & \text{let}\;m=\text{fresh}()\;\text{in}\;\mathcal{S}_{c'}\,[\![e_1]\!]\,\rho\,\lambda\,v\,.\,\underline{\text{let}}\;m=v\;\underline{\text{in}}\;\mathcal{S}_{c'}\,[\![e_2]\!]\,\rho[i\mapsto\underline{m}]\,\kappa
\end{array}
$$

Figure 7: Continuation-style, continuation-based specializer

Let us reorganize the continuation-style specializer of Figure 6 so that its continuation κ is always well-behaved. This reorganization makes it possible to integrate the binding-time improvement for dynamic let expressions.

4.3 A "well-behaved" continuation-style specializer

Any continuation can be viewed as a composition of a well-behaved continuation κ and some other continuation κ'. In the trivial case, κ is simply the identity continuation $\lambda\,v\,.\,v$ (corresponding to the first grammar production above). Achieving the greatest improvement from the rewritten specialization of dynamic let expressions depends on choosing the largest possible well-behaved continuation κ. To keep track of the well-behaved component of the current continuation, Bondorf changes the specializer so that it only builds κ. The non-well-behaved component κ' is applied to the result of specializing the subexpressions [6, Sect. 4].

As analyzed above, in two cases the continuation of the pure CPS specializer is not well-behaved. No well-behaved continuation is available to the specialization of the body of a dynamic lambda abstraction. Here the argument to the specializer representing the well-behaved component of the continuation is set to $\lambda\,v\,.\,v$. The continuation of the specialization of the body of a dynamic let expression is also not well behaved. The continuation of the specialization of the entire let expression is, however, a well-behaved continuation of the specialization of the body. Thus we achieve the binding-time improvement. The continuation-passing, continuation-based specializer is shown in Figure 7. Bondorf proves this transformation of the specializer correct [6, Sect. 5].

5 Direct-style, continuation-based specialization

A continuation-passing, continuation-based specializer is expressed in an hybrid of direct style and continuation-passing style. In the direct-style world, such manipulations over continuations are precisely captured by the control operators shift and reset.

5.1 The control operators shift and reset

Shift and reset were introduced to capture composition and identity over continuations [18, 19]. 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 [24]. Unlike call/cc, which captures the *whole* context of a computation [11], 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

$$
\begin{array}{rcl}
\mathcal{C}[\![i]\!]\kappa & = & \kappa\, i \\
\mathcal{C}[\![\lambda i.e]\!]\kappa & = & \kappa(\lambda i.\lambda k.\mathcal{C}[\![e]\!]k) \\
\mathcal{C}[\![e_0\, e_1]\!]\kappa & = & \mathcal{C}[\![e_0]\!]\lambda v_0.\mathcal{C}[\![e_1]\!]\lambda v_1.v_0\, v_1\, \kappa \\
\mathcal{C}[\![p_0\, ()]\!]\kappa & = & \kappa(p_0\, ()) \\
\mathcal{C}[\![p_1(e_1)]\!]\kappa & = & \mathcal{C}[\![e_1]\!]\lambda v_1.\kappa(p_1(v_1)) \\
\mathcal{C}[\![p_2\, (e_1,\, e_2)]\!]\kappa & = & \mathcal{C}[\![e_1]\!]\lambda v_1.\mathcal{C}[\![e_2]\!]\lambda v_2.\kappa(p_2\, (v_1,\, v_2)) \\
\mathcal{C}[\![p_3\, (e_1,\, e_2,\, e_3)]\!]\kappa & = & \mathcal{C}[\![e_1]\!]\lambda v_1.\mathcal{C}[\![e_2]\!]\lambda v_2.\mathcal{C}[\![e_3]\!]\lambda v_3.\kappa(p_3\, (v_1,\, v_2,\, v_3)) \\
\mathcal{C}[\![\text{let } i = e_1 \text{ in } e_2]\!]\kappa & = & \mathcal{C}[\![e_1]\!]\lambda v_1.\text{let } i = v_1 \text{ in } \mathcal{C}[\![e_2]\!]\kappa \\
\mathcal{C}[\![\text{shift } c \text{ in } e]\!]\kappa & = & (\lambda c.\mathcal{C}[\![e]\!]\lambda a.a)(\lambda v.\lambda k.k(\kappa\, v)) \\
\mathcal{C}[\![\text{reset } e]\!]\kappa & = & \kappa(\mathcal{C}[\![e]\!]\lambda a.a)
\end{array}
$$

Figure 8: The CPS transformation

call/cc, the value of the body of a shift expression is not returned to the context, unless the continuation is explicitly applied.

5.1.1 Example

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

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.

5.1.2 Implementations of shift and reset

Shift and reset can be eliminated by a CPS transformation. 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 equations in Figure 8 extend Plotkin's call-by-value CPS transformation for the λ-calculus with pure primitive operators of arity zero through three, let, shift, and reset. This CPS transformation is documented further in the literature [18, 19, 29, 51, 53, 57].

$$
\begin{array}{rcl}
\mathcal{S}_{d'}\,[\![i]\!]\,\rho & = & \rho\, i \\
\mathcal{S}_{d'}\,[\![\lambda\, i\,.\, e]\!]\,\rho & = & \lambda\, w\,.\,\mathcal{S}_{d'}\,[\![e]\!]\,\rho[i \mapsto w] \\
\mathcal{S}_{d'}\,[\![e_0 \mathbin{@} e_1]\!]\,\rho & = & (\mathcal{S}_{d'}\,[\![e_0]\!]\,\rho)\,(\mathcal{S}_{d'}\,[\![e_1]\!]\,\rho) \\
\mathcal{S}_{d'}\,[\![\mathrm{let}\ i = e_1\ \mathrm{in}\ e_2]\!]\,\rho & = & \mathcal{S}_{d'}\,[\![e_2]\!]\,\rho[i \mapsto \mathcal{S}_{d'}\,[\![e_1]\!]\,\rho] \\
\mathcal{S}_{d'}\,[\![\underline{\lambda}\, i\,.\, e]\!]\,\rho & = & \mathrm{let}\ m = \mathrm{fresh}()\ \mathrm{in}\ \underline{\lambda}\, m\,.\,\mathrm{reset}\,(\mathcal{S}_{d'}\,[\![e]\!]\,\rho[i \mapsto \underline{m}]) \\
\mathcal{S}_{d'}\,[\![e_0 \mathbin{\underline{@}} e_1]\!]\,\rho & = & (\mathcal{S}_{d'}\,[\![e_0]\!]\,\rho) \mathbin{\underline{@}} (\mathcal{S}_{d'}\,[\![e_1]\!]\,\rho) \\
\mathcal{S}_{d'}\,[\![\underline{\mathrm{let}}\ i = e_1\ \underline{\mathrm{in}}\ e_2]\!]\,\rho & = & \mathrm{shift}\ \kappa\ \mathrm{in}\ \mathrm{let}\ m = \mathrm{fresh}()\ \mathrm{in}\ \underline{\mathrm{let}}\ m = \mathcal{S}_{d'}\,[\![e_1]\!]\,\rho\ \underline{\mathrm{in}}\ \mathrm{reset}\,(\kappa\,(\mathcal{S}_{d'}\,[\![e_2]\!]\,\rho[i \mapsto \underline{m}]))
\end{array}
$$

Figure 9: Direct-style, continuation-based specializer

Alternatively, shift and reset can be implemented directly. Their expressive power lies in a single-threaded "meta-continuation" [18, 21, 58] that in effect is structured like a push-down stack, and thus can be globalized in a register [54] (see also Murthy's approach with abstract machines [48]). This observation led Filinski to a direct-style implementation using only call/cc and side-effects over the meta-continuation register [26]. Implementations in Standard ML and in Scheme are available in the literature [26, 43].

5.2 Continuation-based specialization using shift and reset

Continuation-based specialization is obtained from a CPS specializer by rewriting the specialization of dynamic λ-abstractions and of dynamic let expressions. We now see how these effects can be achieved directly using shift and reset.

First, we relocate the continuation of the specialization of a dynamic let expression to the specialization of the body of the let expression. This relocation is achieved by using a shift expression to name this continuation, and then applying the named continuation to the result of specializing the body of the dynamic let expression.

We must then ensure that the continuation accessed by the shift expression is well-behaved. In direct style the continuation is well-behaved if each call to the specializer that occurs in a residual context, occurs in a safe residual context. The result of specializing the body of a λ-abstraction is returned to the constructor of residual λ-abstractions. Thus this specialization does not occur in a safe context. Similarly, in the ordinary direct-style specializer, the specialization of the body of a dynamic let expression does not occur in safe context. When we introduce continuation-based specialization, the result of the specialization of the body is immediately passed to a well-behaved continuation. The result of applying this continuation, however, is returned to a context that is not safe. Thus, in both cases we delimit the safe portion of the context by inserting a reset expression.

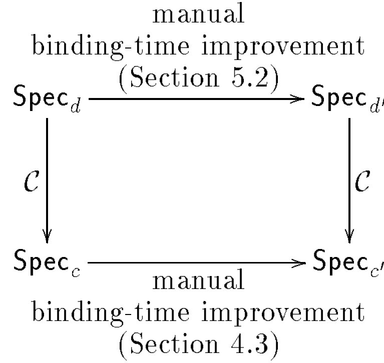
No other modifications to the DS specializer are required. The direct-style, continuation-based specializer is shown in Figure 9. We delimit safe contexts with reset and we relocate them with shift. This transformation achieves continuation-based specialization by only a minor rewriting of two productions of an ordinary direct-style specializer. The new specializer retains the structure of the original one.

5.3 Correctness of the direct-style, continuation-based specializer

Proposition 1 *Automatically transforming the direct-style, continuation-based specializer of Figure 9 into CPS gives the continuation-style, continuation-based specializer of Figure 7.*

The CPS transformation can be carried out using Figure 8, considering the constructors of residual code ($\underline{\lambda}$, $\underline{@}$, and $\underline{\text{let}} \ldots = \ldots \underline{\text{in}} \ldots$) as primitive operators. Since the transformation does not duplicate applications of fresh, and since fresh is insensitive to sequencing order, it can be safely treated as a pure primitive operator.

Proposition 1 shows that superimposing the two diagrams of Section 1.5 yields the following commuting diagram.



The correctness of the direct-style, continuation-based specializer follows directly from the correctness of the CPS transformation and from the correctness proof for continuation-style, continuation-based specialization.

6 Other control-based binding-time improvements

Thus far we have concentrated on let expressions. dynamic applications and dynamic conditional expressions can also benefit from control-based binding-time improvements. Here, however, the result is less satisfactory.

6.1 Dynamic applications

In the binding-time analysis presented for λ-mix (Figure 3), an application is only reconstructed when the value of the function position is dynamic. In this case there is no subexpression whose value is the value of the entire expression. In a partial evaluator such as Similix or Schism [7, 14], however, dynamic applications can occur either as the macro expansion of a let expression naming a dynamic value, or to promote termination when a top-level function is applied.

In Schism, a let expression is macro expanded into the application of a λ-abstraction to the named expression.

$$\text{let } i = e_1 \text{ in } e_2 \quad \equiv \quad (\lambda i.e_2)\, e_1$$

To mirror the binding-time analysis of let expressions naming dynamic values, the binding-time analysis of applications must be modified so that an application is classified as dynamic if either the value of the function position or the value of the argument position is dynamic. When the argument position is dynamic, but the function position is an explicit lambda abstraction, the value of the entire expression is the value of the body of this lambda abstraction. We achieve a control-based binding-time improvement by propagating the context of the entire expression to the body of the λ-abstraction. The new binding-time analysis and specialization rules are shown in Figure 10. This technique was part of an early release of Schism, in 1990.

In the absence of side-effects, top-level functions are constant. Thus every reference to a top-level function has a static value. Consequently, every application applying a top-level function should, by the binding-time analysis of Figure 3, be reduced. Because in a functional language, top-level functions are typically recursive,

$$\frac{\Gamma \vdash e_1 \ \triangleright\ e_1' : \mathsf{d} \qquad \Gamma, i : \mathsf{d} \vdash e_2 \ \triangleright\ e_2' : \tau_2}{\Gamma \vdash (\lambda i. e_2)\, e_1 \ \triangleright\ (\underline{\lambda}\, i\, .\, e_2')\, \underline{@}\, e_1' : \tau_2}$$

$$\begin{array}{rcl}
\mathcal{S}_{c'}\, [\![(\underline{\lambda}\, i\, .\, e_2)\, \underline{@}\, e_1]\!]\, \rho\, \kappa & = & \text{let } m = \text{fresh}() \text{ in } \mathcal{S}_{c'}\, [\![e_1]\!]\, \rho\, \lambda\, v\, .\, (\underline{\lambda}\, m\, .\, \mathcal{S}_{c'}\, [\![e_2]\!]\, \rho[i \mapsto \underline{m}]\, \kappa)\, \underline{@}\, v \\
\mathcal{S}_{d'}\, [\![(\underline{\lambda}\, i\, .\, e_2)\, \underline{@}\, e_1]\!]\, \rho & = & \text{shift } \kappa \text{ in let } m = \text{fresh}() \text{ in } (\underline{\lambda}\, m\, .\, \text{reset}\, (\kappa\, (\mathcal{S}_{d'}\, [\![e_2]\!]\, \rho[i \mapsto \underline{m}])))\, \underline{@}\, \mathcal{S}_{d'}\, [\![e_1]\!]\, \rho
\end{array}$$

Figure 10: Improved binding-time analysis and specialization for beta-redexes

this strategy is likely to lead to non-termination. Various approaches have been devised to determine when it is safe to reduce the application of a top-level function during partial evaluation.

Here again, when the application of a top-level function is not reduced, there is an expression, the body of the top-level definition, whose value is the value of the entire expression. In this case the control-based binding-time improvement is in general problematical. To avoid non-termination, the specializer must create only a bounded number of instances of each top-level function definition. Each specialization of a dynamic application has a distinct continuation that is incomparable to any other continuation of the specializer. Thus it is unsafe to propagate the body of a top-level definition to the context of each corresponding dynamic application.

Here the technique of CPS-transforming the source program prior to partial evaluation can produce better results. The user of the partial evaluator can manually identify situations where the continuation can be safely propagated to the specialization of the body of the top-level definition. Schism's filters provide a convenient mechanism to express which arguments, including the continuation, which in CPS is an explicit argument, should be propagated to the specialization of the body of the top-level function definition [14].

6.2 Dynamic conditional expressions

The syntax, binding-time analysis, and specialization of conditional expressions is shown in Figure 11. A conditional expression is classified as dynamic when the value of the test expression is dynamic. The value of a conditional expression is either the value of the consequent expression or the value of the alternate expression. We achieve a control-based binding-time improvement by propagating the context of the conditional expression to both the consequent expression and the alternate expression.

If the propagated context is non-trivial, this binding-time improvement can lead to exponential code explosion. Furthermore, because conditional expressions typically affect termination, a standard strategy used by both Schism and Similix is to extract each conditional expression into a new top-level definition. The specialization of a conditional expression thus takes advantage of the strategy for preventing non-termination of top-level function calls. In a partial evaluator that performs this preprocessing step, the propagated context is always trivial, and thus the reorganization of the continuation achieves no improvement [45].

Again, because it allows propagation decisions to be made on an individual basis by the user, the technique of first CPS-transforming the source program can achieve better results here. Polyvariant binding-time analysis is particularly helpful here, since the continuation is explicitly applied more than once [13].

$$\frac{\Gamma \vdash e_1 \;\triangleright\; e_1' : \text{bool} \qquad \Gamma \vdash e_2 \;\triangleright\; e_2' : \tau \qquad \Gamma \vdash e_3 \;\triangleright\; e_3' : \tau}{\Gamma \vdash e_1 \;\rightarrow\; e_2,\, e_3 \;\triangleright\; e_1' \;\rightarrow\; e_2',\, e_3' : \tau}$$

$$\frac{\Gamma \vdash e_1 \;\triangleright\; e_1' : \mathsf{d} \qquad \Gamma \vdash e_2 \;\triangleright\; e_2' : \mathsf{d} \qquad \Gamma \vdash e_3 \;\triangleright\; e_3' : \mathsf{d}}{\Gamma \vdash e_1 \;\rightarrow\; e_2,\, e_3 \;\triangleright\; e_1' \underline{\rightarrow} e_2' \underline{,}\, e_3' : \mathsf{d}}$$

$$\begin{aligned}
\mathcal{S}_d\,[\![e_1 \;\rightarrow\; e_2,\, e_3]\!]\,\rho &= \begin{cases} \mathcal{S}_d\,[\![e_2]\!]\,\rho & \text{if } \mathcal{S}_d\,[\![e_1]\!]\,\rho = \text{true} \\ \mathcal{S}_d\,[\![e_3]\!]\,\rho & \text{otherwise} \end{cases} \\
\mathcal{S}_d\,[\![e_1 \underline{\rightarrow} e_2 \underline{,}\, e_3]\!]\,\rho &= \mathcal{S}_d\,[\![e_1]\!]\,\rho \underline{\rightarrow} \mathcal{S}_d\,[\![e_2]\!]\,\rho \underline{,}\, \mathcal{S}_d\,[\![e_3]\!]\,\rho
\end{aligned}$$

Figure 11: Binding-time analysis and specialization for conditional expressions

$$\frac{\Gamma \vdash e_1 \;\triangleright\; e_1' : \mathsf{d} \qquad \Gamma \vdash e_2 \;\triangleright\; e_2' : \tau \qquad \Gamma \vdash e_3 \;\triangleright\; e_3' : \tau}{\Gamma \vdash e_1 \;\rightarrow\; e_2,\, e_3 \;\triangleright\; e_1' \underline{\rightarrow} e_2' \underline{,}\, e_3' : \tau}$$

$$\begin{aligned}
\mathcal{S}_{c'}\,[\![e_1 \underline{\rightarrow} e_2 \underline{,}\, e_3]\!]\,\rho\,\kappa &= \mathcal{S}_{c'}\,[\![e_1]\!]\,\rho\,\lambda b\,.\,b \underline{\rightarrow} \mathcal{S}_{c'}\,[\![e_2]\!]\,\rho\,\kappa \underline{,}\, \mathcal{S}_{c'}\,[\![e_3]\!]\,\rho\,\kappa \\
\mathcal{S}_{d'}\,[\![e_1 \underline{\rightarrow} e_2 \underline{,}\, e_3]\!]\,\rho &= \text{shift } \kappa \text{ in } \mathcal{S}_{d'}\,[\![e_1]\!]\,\rho \underline{\rightarrow} \text{reset}\,(\kappa\,(\mathcal{S}_{d'}\,[\![e_2]\!]\,\rho)) \underline{,}\, \text{reset}\,(\kappa\,(\mathcal{S}_{d'}\,[\![e_3]\!]\,\rho))
\end{aligned}$$

Figure 12: Improved binding-time analysis and specialization for conditional expressions

7 Implementation issues

Thus far we have seen two implementations of control-based binding-time improvements. The first is expressed in a variant of continuation-passing style and the second is expressed in direct style. We have argued that the direct-style implementation is more convenient for the implementor of a partial evaluator because it requires fewer modifications to the specializer, and we have mentioned that the direct-style implementation gives better performance than the continuation-style implementation when shift and reset are implemented directly using call/cc and side effects. In this section we report some experiments to test these assertions.

7.1 Bootstrapping

Using control operators and the CPS transformation enables the following bootstrapping procedure: freely experiment with control in the direct-style specializer, and then 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 value as well. The first author has used this method to bootstrap the specializer in the new version of Schism [14]. 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).

This experiment illustrated another possible improvement. In Schism, binding-time information is compiled into specialization actions, to remove interpretive overhead in the specializer [15]. Completely static and completely dynamic expressions are factored out, and the specializer can concentrate on specialization proper, with a smaller continuation traffic. The shift/reset method applies there as well.

7.2 Measures

Analyzing the cost of our four partial evaluators PE_d, PE_c, $\mathsf{PE}_{d'}$, and $\mathsf{PE}_{c'}$ raises two issues. We need to compare the costs of the continuation-style and direct-style 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.

7.2.1 An experiment

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 increases the size of the resulting substitution and thus the size of the residual program.

We consider four versions of the source program. In each 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. `pm-static-cps` is the CPS counterpart of `pm-static`. `pm-dynamic-cps` is the CPS counterpart of `pm-dynamic`.

The tables in Figure 13 compare the time and space costs of specializing these programs using PE_d, $\mathsf{PE}_{d'}$, PE_c, and $\mathsf{PE}_{c'}$. We consider patterns of varying depth and length, and thus specializations where both the size of the continuation and the size of the residual program vary. The following sections analyze these results.

7.2.2 Comparing the direct-style and the continuation-style 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 direct-style source programs [41]. Scheme goes as far as to leave the sequencing order unspecified to allow the compiler writer further opportunities for optimization [11]. 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,[3] our system of reference [23, 33].

These observations are borne out by the comparison of PE_c and PE_d ($\mathsf{PE}_c/\mathsf{PE}_d$) and by the comparison of $\mathsf{PE}_{c'}$ and $\mathsf{PE}_{d'}$ ($\mathsf{PE}_{c'}/\mathsf{PE}_{d'}$). In each case the continuation-style specializer (PE_c and $\mathsf{PE}_{c'}$) uses more space than its direct-style counterpart (PE_d and $\mathsf{PE}_{d'}$, resp.), and in some cases almost twice as much. The time consumed by each continuation-style specializer is, in almost all cases, more than that of its direct-style counterpart, as well.

7.2.3 Assessing the cost of the improvement

Continuation-based specialization (PE_c and $\mathsf{PE}_{c'}$) 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. The naïve specializer, in contrast, residualizes these tests. Thus, it performs much less static computation and constructs many fewer data structures.

[3] The Chez Scheme compiler is a direct-style compiler with an efficient implementation of first-class continuations [23, 33].

pm-static									
		PE_c/PE_d		$PE_{c'}/PE_{d'}$		$PE_{d'}/PE_d$		$PE_{c'}/PE_c$	
continuation	residual program size	time	space	time	space	time	space	time	space
small	small	1.16	1.48	1.14	1.40	1.11	1.09	1.09	1.03
large	small	1.27	1.77	1.25	1.69	1.06	1.01	1.05	0.96
small	large	1.18	1.61	1.01	1.09	4.65	6.22	3.97	4.20
large	large	1.26	1.71	1.01	1.12	4.05	5.31	3.24	3.48

pm-dynamic									
		PE_c/PE_d		$PE_{c'}/PE_{d'}$		$PE_{d'}/PE_d$		$PE_{c'}/PE_c$	
continuation	residual program size	time	space	time	space	time	space	time	space
small	small	1.48	1.50	0.93	1.42	1.43	1.05	0.90	1.00
large	small	1.18	1.78	1.23	1.61	0.93	1.10	0.97	1.00
small	large	1.14	1.63	1.20	1.51	0.93	1.08	0.98	1.00
large	large	1.14	1.72	1.20	1.57	0.93	1.09	0.98	1.00

pm-static-cps									
		PE_c/PE_d		$PE_{c'}/PE_{d'}$		$PE_{d'}/PE_d$		$PE_{c'}/PE_c$	
continuation	residual program size	time	space	time	space	time	space	time	space
small	small	1.12	1.62	1.04	1.63	1.08	0.96	1.00	0.97
large	small	1.19	1.71	1.04	1.68	1.16	1.01	1.01	1.00
small	large	0.96	1.08	1.02	1.14	0.66	0.60	0.70	0.63
large	large	1.05	1.11	1.03	1.17	0.79	0.64	0.78	0.67

pm-dynamic-cps									
		PE_c/PE_d		$PE_{c'}/PE_{d'}$		$PE_{d'}/PE_d$		$PE_{c'}/PE_c$	
continuation	residual program size	time	space	time	space	time	space	time	space
small	small	1.14	1.64	1.14	1.58	1.00	1.03	1.00	1.00
large	small	1.35	1.89	1.37	1.87	0.99	1.01	1.00	1.00
small	large	1.00	1.08	1.01	1.08	0.99	1.00	1.00	1.00
large	large	1.23	1.51	1.15	1.51	1.07	1.00	1.00	1.00

Figure 13: Relative costs of specializing direct-style and CPS pattern-matching programs with respect to several patterns

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 general, when the body of a dynamic let expression returns a dynamic value, it is useless to reorganize the continuations. Similarly, dynamic conditional expressions with dynamic branches, and beta-redexes with a dynamic argument and where the body of the λ-abstraction is dynamic do not offer any opportunity for a control-based binding-time improvement (see Section 6). In our experiments with let expressions we observed that reorganizing the continuations does not affect performance.

7.2.4 Iterative programs

When the continuation accessed by shift is always the trivial well-behaved continuation, continuation-based specialization does not introduce more opportunities for binding-time improvement (*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 run-time and run-space of the continuation-based specializer should be identical to that of the naïve specializer when the source program is in continuation-style. As noted in Section 1.6, all four specializers should produce the maximally specialized residual program as well. Similarly, when an arbitrary tail-recursive program is specialized, the control stack of PE_d and the continuation of PE_c do not grow (see [16, Prop. 1]). Therefore their run-time and their run-space should not diverge. This analysis is confirmed by the time and space costs for `pm-static-cps` and `pm-dynamic-cps`.

7.2.5 An alternative implementation

In Figure 9, each continuation application is surrounded by a reset. Thus Felleisen's control (a.k.a. $\mathcal{F}$) and prompt operators [24] can be used here as well, even though they go beyond CPS.[4]

Using Sitaram and Felleisen's implementation [56], we have also measured the time and space costs of $\mathsf{PE}_{d'}$ when shift and reset are replaced by control and prompt. Both costs are virtually unchanged.

7.2.6 Side issues

Independently of partial evaluation, the measures above compare the running of direct-style code, possibly in the presence of first-class continuations, and the running of naïve CPS code — where by "naïve" we mean that no special provision is made for the continuation [27]. $\mathsf{PE}_{d'}$ yields a situation where large parts of the same continuation are repeatedly captured and restored.[5] The common parts are naturally shared in CPS, but in direct-style, 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 [2, 3, 33, 47].

7.3 Conclusion

Bondorf's motivation for $\mathsf{Spec}_{c'}$ is the cost of specializing CPS programs [6]. In addition to the cost of specializing a CPS source program, however, there is the cost of simply running the CPS program $\mathsf{Spec}_{c'}$. This problem does not occur in our direct-style, continuation-based specializer $\mathsf{Spec}_{d'}$. As reported in this

[4]Control and prompt, in general, do not have any continuation-style counterpart [19].

[5]At the 1988 Lisp conference, Clinger, Hartheimer, and Ost reported that typically 80% of captured continuations are shared in MacScheme [12].

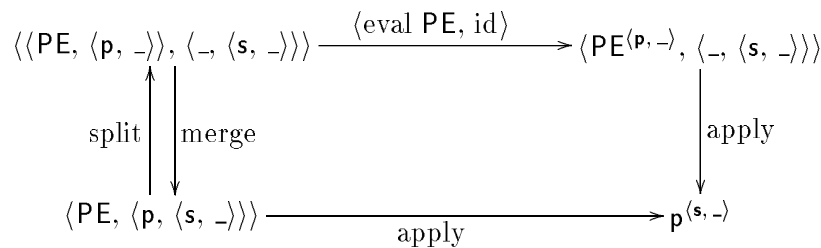
section, this specializer can run without incurring the overhead of CPS code in a language implementation that supports call/cc and side effects efficiently.

This cost is lifted to self-application, however, because the style of continuation-passing, continuation-based specializers is similar to CPS. The next section addresses this problem.

8 Self-applicable partial evaluation

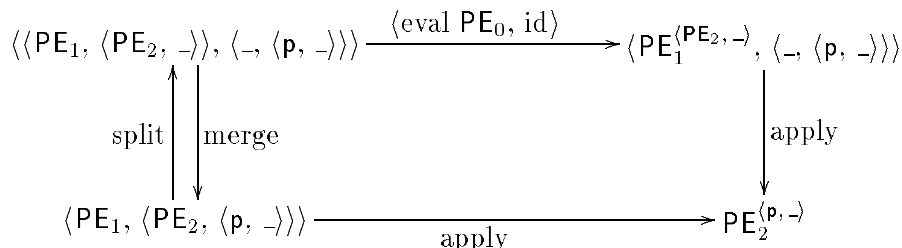
8.1 Background

Sometimes, one needs to specialize the same source program p several times, *e.g.*, with respect to different input values. In other words, one needs to apply PE several times to an input $\langle \mathsf{p}, _ \rangle$ where p is fixed. This repeated specialization can be made more efficient by first specializing PE with respect to p, as captured in the following diagram.



This diagram is a particular instance of the diagram in Section 1.1, where the source program is PE and the static input is $\langle \mathsf{p}, _ \rangle$.

Furthermore, one may need to specialize PE several times, *e.g.*, with respect to different source programs. In other words, one may need to apply PE several times to an input $\langle \mathsf{PE}, _ \rangle$. This repeated specialization can be made more efficient by first specializing PE with respect to PE, as captured in the following diagram. For clarity, we index the occurrences of PE.

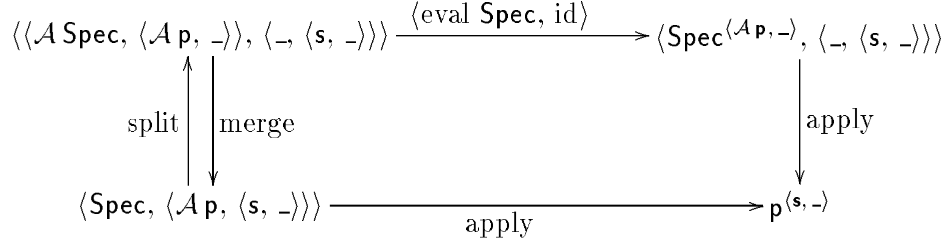


Again this diagram is a particular instance of the diagram in Section 1.1 Here the source program is PE_1 and the static input is $\langle \mathsf{PE}_2, _ \rangle$.

Since $\mathsf{PE}_1^{\langle \mathsf{PE}_2, _ \rangle}$ transforms a program into a partial evaluator dedicated to this program, it acts as a partial-evaluation compiler [17]. In the rest of this section, we refer to it as "pecom". In the particular case where the source program is an interpreter, the diagrams above specialize into the "Futamura projections" [30, 38]. There, pecom has the functionality of a compiler generator, and thus is often referred to as "cogen" in the literature [8, 35].

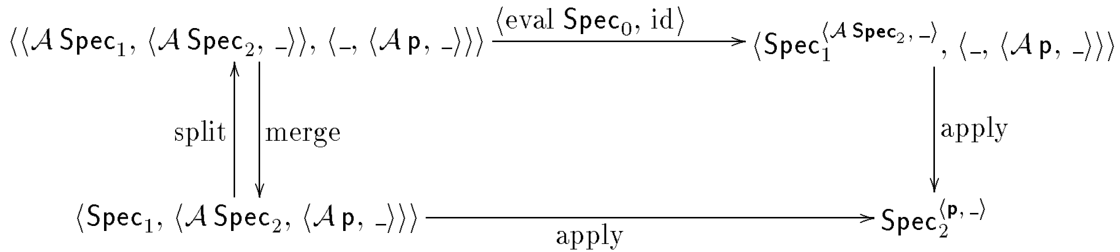
8.2 Offline partial evaluation

In the context of offline partial evaluation, it is the specializer, not the partial evaluator, that is self-applied. Thus the diagrams above need to be adjusted. The first diagram is rewritten as follows:



This diagram is a particular instance of the diagram in Section 1.7, where the source program is Spec and the static input is $\langle \mathcal{A}\,\mathsf{p},\ _\rangle$. In particular, the binding time of the input to p is fixed and thus the resulting dedicated specializer $\mathsf{Spec}^{\langle \mathcal{A}\,\mathsf{p},\ _\rangle}$ expects an input with a fixed binding-time format.

Similarly, the second diagram is rewritten as follows:

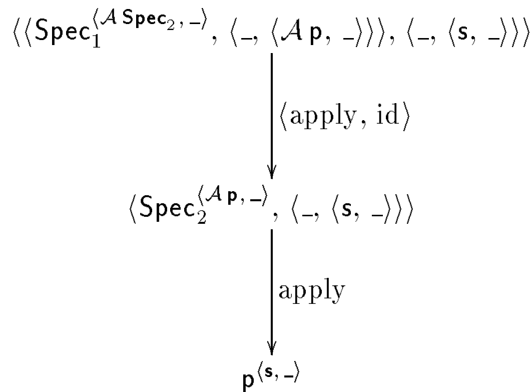


Again, this diagram is a specialized version of the previous diagram, where the static input is $\langle \mathcal{A}\,\mathsf{Spec}_2,\ _\rangle$. In particular, the binding-time of the input to Spec_2 is fixed.

8.3 Handwriting pecom

Over the last five years [34, 35] handwritten pecoms have been found to be more efficient than those generated by self-application. This approach has been taken for C [1], for ML [5], and for λ-Mix [8].

First pecom is used to generate a specializer tailored to the source program; then this dedicated specializer can be repeatedly applied to static inputs. As confirmed by the right-hand sides of the diagrams above, two "apply" steps are required when using pecom to obtain a specialized program.



$$
\begin{array}{rcl}
\mathcal{S}_{c'}^{\mathcal{S}_{c'}}\,[\![i]\!]\,\kappa & = & \kappa\,\underline{i} \\
\mathcal{S}_{c'}^{\mathcal{S}_{c'}}\,[\![\lambda\, i\,.\,e]\!]\,\kappa & = & \text{let } n = \text{fresh}()\text{ in }\kappa\,(\underline{\lambda}\, i\,.\,\underline{\lambda}\, n\,.\,\mathcal{S}_{c'}^{\mathcal{S}_{c'}}\,[\![e]\!]\,\lambda\, v\,.\,\underline{n}\,\underline{@}\,v) \\
\mathcal{S}_{c'}^{\mathcal{S}_{c'}}\,[\![e_1 \mathbin{@} e_2]\!]\,\kappa & = & \text{let } n = \text{fresh}()\text{ in }\mathcal{S}_{c'}^{\mathcal{S}_{c'}}\,[\![e_1]\!]\,\lambda\, v\,.\,\mathcal{S}_{c'}^{\mathcal{S}_{c'}}\,[\![e_2]\!]\,\lambda\, w\,.\,(v\,\underline{@}\,w)\,\underline{@}\,(\underline{\lambda}\, n\,.\,\kappa\,\underline{n}) \\
\mathcal{S}_{c'}^{\mathcal{S}_{c'}}\,[\![\text{let } i = e_1 \text{ in } e_2]\!]\,\kappa & = & \mathcal{S}_{c'}^{\mathcal{S}_{c'}}\,[\![e_1]\!]\,\lambda\, v\,.\,\underline{\text{let}}\; i = v\;\underline{\text{in}}\;\mathcal{S}_{c'}^{\mathcal{S}_{c'}}\,[\![e_2]\!]\,\kappa \\
\mathcal{S}_{c'}^{\mathcal{S}_{c'}}\,[\![\underline{\lambda}\, i\,.\,e]\!]\,\kappa & = & \text{let } m = \text{fresh}()\text{ in }\kappa\,(\underline{\text{let}}\; m = \underline{\text{fresh}}\,()\;\underline{\text{in}}\;\underline{\text{let}}\; i = \underline{\underline{\underline{m}}}\;\underline{\text{in}}\;\underline{\underline{\lambda}}\,\underline{m}\,.\,\mathcal{S}_{c'}^{\mathcal{S}_{c'}}\,[\![e]\!]\,\lambda\, x\,.\,x) \\
\mathcal{S}_{c'}^{\mathcal{S}_{c'}}\,[\![e_1\,\underline{@}\,e_2]\!]\,\kappa & = & \mathcal{S}_{c'}^{\mathcal{S}_{c'}}\,[\![e_1]\!]\,\lambda\, v\,.\,\mathcal{S}_{c'}^{\mathcal{S}_{c'}}\,[\![e_2]\!]\,\lambda\, w\,.\,\kappa\,(v\,\underline{\underline{@}}\,w) \\
\mathcal{S}_{c'}^{\mathcal{S}_{c'}}\,[\![\underline{\text{let}}\; i = e_1\;\underline{\text{in}}\; e_2]\!]\,\kappa & = & \text{let } m = \text{fresh}() \\
 & & \text{in }\mathcal{S}_{c'}^{\mathcal{S}_{c'}}\,[\![e_1]\!]\,\lambda\, v\,.\,\underline{\text{let}}\; m = \underline{\text{fresh}}\,()\;\underline{\text{in}}\;\underline{\text{let}}\; i = \underline{\underline{\underline{m}}}\;\underline{\text{in}}\;\underline{\underline{\text{let}}}\;\underline{m} = v\;\underline{\underline{\text{in}}}\;\mathcal{S}_{c'}^{\mathcal{S}_{c'}}\,[\![e_2]\!]\,\kappa
\end{array}
$$

Figure 14: Continuation-style, continuation-based pecom

Continuation-based partial evaluation can be beneficial at both stages. Thus, 16 possible pecoms exist, corresponding to the possible instantiations of Spec_1 and Spec_2:

Spec_2 / Spec_1	direct style	cont. style	cont. style cont. based	direct style cont. based
direct style	$\mathsf{Spec}_d^{\mathsf{Spec}_d}$	$\mathsf{Spec}_d^{\mathsf{Spec}_c}$	$\mathsf{Spec}_d^{\mathsf{Spec}_{c'}}$	$\mathsf{Spec}_d^{\mathsf{Spec}_{d'}}$
cont. style	$\mathsf{Spec}_c^{\mathsf{Spec}_d}$	$\mathsf{Spec}_c^{\mathsf{Spec}_c}$	$\mathsf{Spec}_c^{\mathsf{Spec}_{c'}}$	$\mathsf{Spec}_c^{\mathsf{Spec}_{d'}}$
cont. style cont. based	$\mathsf{Spec}_{c'}^{\mathsf{Spec}_d}$	$\mathsf{Spec}_{c'}^{\mathsf{Spec}_c}$	$\mathsf{Spec}_{c'}^{\mathsf{Spec}_{c'}}$	$\mathsf{Spec}_{c'}^{\mathsf{Spec}_{d'}}$
direct style cont. based	$\mathsf{Spec}_{d'}^{\mathsf{Spec}_d}$	$\mathsf{Spec}_{d'}^{\mathsf{Spec}_c}$	$\mathsf{Spec}_{d'}^{\mathsf{Spec}_{c'}}$	$\mathsf{Spec}_{d'}^{\mathsf{Spec}_{d'}}$

For example, $\mathsf{Spec}_c^{\mathsf{Spec}_{d'}}$ is a continuation-style program generating direct-style, continuation-based dedicated specializers, and $\mathsf{Spec}_{d'}^{\mathsf{Spec}_d}$ is a direct-style, continuation-based program generating direct-style dedicated specializers.

$\mathsf{Spec}_d^{\mathsf{Spec}_d}$ has been handwritten for C, for ML, and for λ-Mix [1, 5, 8]. $\mathsf{Spec}_{c'}^{\mathsf{Spec}_{c'}}$ has been handwritten for λ-Mix [8].

8.4 Continuation-style, continuation-based partial-evaluation compilers

Bondorf and Dussart have integrated the control-based binding-time improvement for let expressions into a handwritten partial-evaluation compiler $\mathsf{Spec}_{c'}^{\mathsf{Spec}_{c'}}$, shown in Figure 14 [8]. Since both pecom and the code it produces are in a style similar to CPS, a double overhead for running CPS code is incurred. We propose to construct $\mathsf{Spec}_{d'}^{\mathsf{Spec}_{d'}}$, thus avoiding this overhead at both stages.

For conciseness, in the following section, we abbreviate Spec in the name of a handwritten pecom by $\mathcal{S}$ — *e.g.*, $\mathcal{S}_{d'}^{\mathcal{S}_{d'}}$.

8.5 Direct-style, continuation-based partial-evaluation compilers

Given a binding-time analyzed source program, a specialized specializer constructs the code for the actions that a specializer would perform. Thus, following the definition of $\mathcal{S}_{d'}$, the transformation of a dynamic λ-abstraction should construct a $\underline{\text{reset}}$ expression, and the transformation of a dynamic let expression should

$$
\begin{array}{rcl}
\mathcal{S}_{d'}^{\mathcal{S}_{d'}}[\![i]\!] & = & \underline{i} \\
\mathcal{S}_{d'}^{\mathcal{S}_{d'}}[\![\lambda\, i\,.\,e]\!] & = & \underline{\lambda}\, i\,.\,\mathrm{reset}\,(\mathcal{S}_{d'}^{\mathcal{S}_{d'}}[\![e]\!]) \\
\mathcal{S}_{d'}^{\mathcal{S}_{d'}}[\![e_1 \mathbin{@} e_2]\!] & = & (\mathit{complete}\,(e_1))\,\underline{@}\,(\mathit{complete}\,(e_2)) \\
\mathcal{S}_{d'}^{\mathcal{S}_{d'}}[\![\mathrm{let}\ i = e_1\ \mathrm{in}\ e_2]\!] & = & \mathrm{shift}\ k\ \mathrm{in}\ \underline{\mathrm{let}}\ i = \mathit{complete}\,(e_1)\,\underline{\mathrm{in}}\ \mathrm{reset}\,(k\,(\mathcal{S}_{d'}^{\mathcal{S}_{d'}}[\![e_2]\!])) \\
\mathcal{S}_{d'}^{\mathcal{S}_{d'}}[\![\underline{\lambda}\, i\,.\,e]\!] & = & \mathrm{let}\ m = \mathrm{fresh}() \\
 & & \mathrm{in}\ \underline{\mathrm{reset}}\,(\underline{\mathrm{let}}\ m = \underline{\mathrm{fresh}}\,()\ \underline{\mathrm{in}}\ \underline{\mathrm{let}}\ i = \underline{\underline{\underline{m}}}\ \underline{\mathrm{in}}\ \underline{\underline{\lambda}}\,\underline{m}\,.\,\underline{\mathrm{reset}}\,(\mathrm{reset}\,(\mathcal{S}_{d'}^{\mathcal{S}_{d'}}[\![e]\!]))) \\
\mathcal{S}_{d'}^{\mathcal{S}_{d'}}[\![e_1\,\underline{@}\,e_2]\!] & = & (\mathit{complete}\,(e_1))\,\underline{\underline{@}}\,(\mathit{complete}\,(e_2)) \\
\mathcal{S}_{d'}^{\mathcal{S}_{d'}}[\![\underline{\mathrm{let}}\ i = e_1\ \underline{\mathrm{in}}\ e_2]\!] & = & \mathrm{let}\ k_1 = \mathrm{fresh}() \\
 & & \mathrm{in\ shift}\ k \\
 & & \quad \mathrm{in}\ \underline{\mathrm{shift}}\ k_1 \\
 & & \qquad \underline{\mathrm{in}}\ \mathrm{let}\ m = \mathrm{fresh}() \\
 & & \qquad\quad \mathrm{in}\ \underline{\mathrm{let}}\ m = \underline{\mathrm{fresh}}\,() \\
 & & \qquad\qquad \underline{\mathrm{in}}\ \underline{\mathrm{let}}\ i = \underline{\underline{\underline{m}}} \\
 & & \qquad\qquad\quad \underline{\mathrm{in}}\ \underline{\underline{\mathrm{let}}}\ \underline{m} = \mathit{complete}\,(e_1) \\
 & & \qquad\qquad\qquad \underline{\underline{\mathrm{in}}}\ \underline{\mathrm{reset}}\,(\underline{k_1}\ \underline{@}\,(\mathrm{reset}\,(k\,(\mathcal{S}_{d'}^{\mathcal{S}_{d'}}[\![e_2]\!]))))
\end{array}
$$

where *complete* is defined as follows:

$$
\mathit{complete}\,(e) \quad = \quad \mathrm{let}\ a = \mathrm{fresh}()\ \mathrm{in\ shift}\ k\ \mathrm{in}\ \underline{\mathrm{let}}\ a = \mathcal{S}_{d'}^{\mathcal{S}_{d'}}[\![e]\!]\ \underline{\mathrm{in}}\ k\ \underline{a}
$$

Figure 15: Direct-style, continuation-based pecom

construct a shift expression, a reset expression, and a continuation application. Where $\mathcal{S}_{d'}$ constructs residual code, $\mathcal{S}_{d'}^{\mathcal{S}_{d'}}$ emits primitive operators that construct residual code. These are indicated by a double underline.

In $\mathcal{S}_{d'}^{\mathcal{S}_{d'}}$, as opposed to *e.g.*, $\mathcal{S}_{d}^{\mathcal{S}_{d'}}$, as much of the reorganization of the continuation should happen at pecom time as possible. The approach is identical to the construction of $\mathcal{S}_{d'}$. To carry out the control-based binding-time improvement at pecom time, we introduce a shift expression around the transformation of a dynamic let expression. The named continuation is applied to the transformation of the body of the let expression, thus achieving the improvement. The residual shift expression then further accesses the portion of the continuation that is not available at pecom time, namely the continuation of the application of a static λ-abstraction.

As in the construction of $\mathcal{S}_{d'}$, we must ensure that this shift expression accesses a well-behaved continuation. $\mathcal{S}_{d'}$ contains a reset expression around the transformation of the body of a dynamic λ-abstraction because a term, rather than a function value, is constructed. In $\mathcal{S}_{d'}^{\mathcal{S}_{d'}}$ a term is constructed both in the transformation of a static λ-abstraction and in the transformation of a dynamic λ-abstraction. Thus, in both cases a reset expression is required. Similarly a shift expression is required both in the transformation of a static let expression and in the transformation of a dynamic let expression to propagate the continuation to the body. The reset expression around the application of the named continuation in both cases again cuts off the transformation of the body from the construction of the residual term.

$\mathcal{S}_{d'}^{\mathcal{S}_{d'}}$ manipulates continuations both at pecom time and at evaluation time. The procedure *complete* prevents the processing of subexpressions from being interleaved. *Complete* inserts a residual let expression naming the result of calling $\mathcal{S}_{d'}^{\mathcal{S}_{d'}}$. This let expression ensures that the code produced by the call to $\mathcal{S}_{d'}^{\mathcal{S}_{d'}}$ and the code produced by evaluating the result are evaluated before the code produced by any call to $\mathcal{S}_{d'}^{\mathcal{S}_{d'}}$ in the continuation of the current expression. In the transformation of a static application, for example, either

the expression in function position is completely processed first or the expression in argument position is completely processed first, depending on the sequencing order of the meta-language of the evaluator [52]. The processing of the two is not, however, interleaved. A shift expression is used here, as for the other residual let expressions constructed by $\mathcal{S}_{d'}^{\mathcal{S}_{d'}}$, to propagate the continuation to the body of the let expression.

$\mathcal{S}_{d'}^{\mathcal{S}_{d'}}$ is shown in Figure 15.

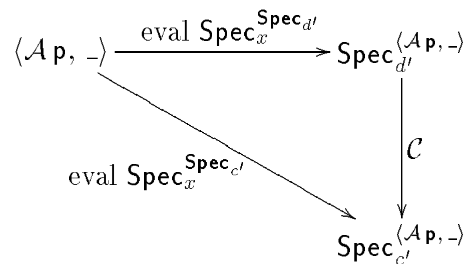
Proposition 2 *Automatically transforming $\mathcal{S}_{d'}^{\mathcal{S}_{d'}}$ (Figure 15) into CPS gives $\mathcal{S}_{c'}^{\mathcal{S}_{d'}}$. Composing the result with the CPS transformation (Figure 8) gives $\mathcal{S}_{c'}^{\mathcal{S}_{c'}}$ (Figure 14).*

Along the same lines as in Section 5.3, the correctness of $\mathcal{S}_{d'}^{\mathcal{S}_{d'}}$ follows directly from the correctness of the CPS transformation and from Bondorf and Dussart's correctness proof for $\mathcal{S}_{c'}^{\mathcal{S}_{c'}}$ [8].

8.6 Summary

The overall situation is summarized in the following commuting diagram.

$\forall x \in \{d', c'\}$,



$\mathsf{Spec}_{c'}^{\mathsf{Spec}_{c'}}$ is obtained by CPS-transforming $\mathsf{Spec}_{d'}^{\mathsf{Spec}_{c'}}$, and $\mathsf{Spec}_{c'}^{\mathsf{Spec}_{d'}}$, by CPS-transforming $\mathsf{Spec}_{d'}^{\mathsf{Spec}_{d'}}$.

9 Related work

The present work is a part of our investigation of the direct-style transformation in the presence of first-class continuations [20]. The technique for composing the CPS transformation with $\mathcal{S}_{c'}^{\mathcal{S}_{d'}}$ in Section 8.5 was developed in the first author's PhD thesis [42].

Like Similix [6] and Schism [14], Pell-Mell, the ML partial evaluator of Malmkjær, Heintze, and Danvy also uses continuation-based specialization [46].

Independently of our work, Felleisen observed that Bondorf's key step corresponds to delimiting and abstracting control [25]. Friedman and Ashley also observed that shift and reset could be used to develop a self-applicable partial evaluator [28]. Finally, Dussart pursued $\mathsf{Spec}_{d'}^{\mathsf{Spec}_{d'}}$ along the lines of Section 8.5, as outlined in Section 3.2 of the conference version of this paper [22].

Other continuation-based program transformations can be expressed in direct style with shift and reset, the CPS transformation itself, for example [18, 19].

10 Conclusion

Continuation-based partial evaluation achieves a control-based binding-time improvement: relocating the contexts of dynamic expressions for a better support of static data flow. This relocation can be achieved by pre-transforming the source program into CPS. The effect of this pre-transformation can also be integrated into a partial evaluator.

In an offline partial evaluator, the binding-time improvement requires modifying both the binding-time analysis (Section 3) and the specializer (Sections 4 and 5). The binding-time analysis classifies more values as static. The specializer can process the extra static information by representing control either explicitly or implicitly. One specializer is the CPS counterpart of the other.

The same argument holds for partial-evaluation compilers at two stages: at (partial-evaluation) compile time, and at (partial-evaluation) run time. Both a partial-evaluation compilers and the compiled partial evaluators are instances of a specializer. Each can represent control either explicitly or implicitly. These two representations are related through the CPS transformation (Section 8).

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. They also enable a more efficient implementation of continuation-based partial evaluation (Section 7).

Acknowledgements

Andrzej Filinski and Karoline Malmkjær provided valuable criticism and encouraging comments. Thanks are also due to the LFP94 referees for perceptive comments, and to Charles Consel for support. The diagrams were drawn with Kristoffer Rose's XY-pic package.

A Continuation-based partial evaluation in Standard ML

This appendix contains the complete SML code implementing the four specializers (PE_d, PE_c, $\mathsf{PE}_{d'}$, and $\mathsf{PE}_{c'}$), the two pecoms ($\mathsf{Spec}_{d'}^{\mathsf{Spec}_{d'}}$ and $\mathsf{Spec}_{c'}^{\mathsf{Spec}_{c'}}$), and the CPS transformation.

The source and target language is defined by the type `residual`, shown at the top of Figure 16. Figure 17 uses these constructors to encode the following lambda term:

$$\lambda f\,.\,\lambda x\,.\,(\lambda g\,.\,(\lambda h\,.\,h\,(g\,(f\,x)))\,(\lambda j\,.\,j\,(\lambda a\,.\,a)))$$
$$(\lambda y\,.\,\mathrm{let}\;y_1 = y\;\mathrm{in}\;\lambda z\,.\,z)$$

The language of annotated terms is defined by the type `annotated`, shown at the bottom of Figure 16. In our example, the let expression binding y_1 is dynamic, but its body has a static value. Thus $\mathcal{A}$ and $\mathcal{A}'$ yield different versions of this term. Figure 18 shows the annotated program (`exp`) produced by $\mathcal{A}$. This program is suitable for specialization by Spec_d or Spec_c. Figure 20 shows the annotated program (`exp'`) produced by $\mathcal{A}'$. This program is suitable for specialization by $\mathsf{Spec}_{d'}$ or $\mathsf{Spec}_{c'}$ and for partial-evaluation compilation by $\mathsf{Spec}_{d'}^{\mathsf{Spec}_{d'}}$ or $\mathsf{Spec}_{c'}^{\mathsf{Spec}_{c'}}$.

Figure 25 shows the unimproved direct-style specializer, Spec_d. Figure 26 shows the unimproved continuation-passing-style specializer Spec_c. The result of applying either Spec_d or Spec_c to `exp` is shown in Figure 19. Because `Lam ("z",Var "z")` is hidden by the dynamic let expression naming y_1, the application of `Lam ("z",Var "z")` to `Lam ("a",Var "a")` is not reduced during specialization.

Figure 27 shows the direct-style continuation-based specializer, $\mathsf{Spec}_{d'}$. Figure 28 shows the continuation-style, continuation-based specializer $\mathsf{Spec}_{c'}$. The result of applying either $\mathsf{Spec}_{d'}$ or $\mathsf{Spec}_{c'}$ to `exp'` is shown in Figure 21. Here the application to `Lam ("a",Var "a")` is propagated to the body of the dynamic let expression. Thus the application of `Lam ("z",Var "z")` to `Lam ("a",Var "a")` is reduced during specialization, producing the residual code `Lam ("g_15",Var "g_15")`.

```
structure PE_Exp =
  struct

    datatype residual = Var of string
                      | Lam of string * residual
                      | App of residual * residual
                      | Let of string * residual * residual
                      | Op0 of string
                      | Op1 of string * residual
                      | Op2 of string * residual * residual
                      | Op3 of string * residual * residual * residual
                      | Shift of string * residual
                      | Reset of residual

    datatype annotated = over_Var of string
                       | over_Lam of string * annotated
                       | over_App of annotated * annotated
                       | over_Let of string * annotated * annotated
                       | under_Lam of string * annotated
                       | under_App of annotated * annotated
                       | under_Let of string * annotated * annotated

  end
```

Figure 16: The source and target language of λ-terms and the language of annotated λ-terms in Standard ML (Figures 1 and 2)

The direct-style, continuation-based pecom $\mathsf{Spec}_{d'}^{\mathsf{Spec}_{d'}}$ is shown in Figure 29. The continuation-style, continuation-based pecom $\mathsf{Spec}_{c'}^{\mathsf{Spec}_{c'}}$ is shown in Figure 30. The result of applying these partial-evaluation compilers to the annotated source program `exp'` is not shown, for space reasons. In each case, transliterating the result into SML and evaluating it yields the same residual code as in Figure 21.

Some auxiliary definitions are shown in Figures 22, 23, and 24. `is_text_d` and `is_value_d` remove type tags from the results of the direct-style specializers. As noted in Section 2, the correctness of the binding-time analysis ensures that `is_text_d` is only applied to results labeled `Text_d` and `is_value_d` is only applied to results labeled `Value_d`. `is_text_c` and `is_value_c` are the corresponding operations for the continuation-passing style specializers. `init_env` is the initial environment. `extend_env` extends the environment. `fresh` creates fresh strings, which represent fresh identifiers.

The CPS transformation on source and target (unannotated) terms is shown in Figure 31. Filinski's implementation of shift and reset using call/cc is shown in Figure 32 [26]. The Scheme counterpart of this definition was used in obtaining the measurements of Section 7.2.

References

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, May 1994. DIKU TechReport 94/19.

[2] Andrew W. Appel and Zhong Shao. An empirical and analytic study of stack vs. heap cost for language with closures. Technical report CS-TR-450-94, Department of Computer Science, Princeton University,

```
Lam ("f",
     Lam ("x",
          App (Lam ("g",
                    App (Lam ("h",
                              App (Var "h",
                                   App (Var "g",
                                        App (Var "f",
                                             Var "x")))),
                         Lam ("j",
                              App (Var "j",
                                   Lam ("a", Var "a"))))),
               Lam ("y", Let ("y1",
                              Var "y",
                              Lam ("z", Var "z"))))))
```

Figure 17: The example, in the source language

```
val exp = under_Lam ("f", under_Lam ("x",
    over_App (over_Lam ("g",
                        over_App (over_Lam ("h",
                                            over_App (over_Var "h",
                                                      over_App (over_Var "g",
                                                                under_App (over_Var "f",
                                                                           over_Var "x")))),
                                  over_Lam ("j",
                                            under_App (over_Var "j",
                                                       under_Lam ("a", over_Var "a"))))),
              over_Lam ("y", under_Let ("y1",
                                        over_Var "y",
                                        under_Lam ("z", over_Var "z"))))))
```

Figure 18: The example, annotated by $\mathcal{A}$ — `exp`

```
Lam ("g_2",
     Lam ("g_3",
          App (Let ("g_4", App (Var "g_2", Var "g_3"), Lam ("g_5", Var "g_5")),
               Lam ("g_6", Var "g_6"))))
```

Figure 19: The result of specializing `exp`

```
val exp' = under_Lam ("f", under_Lam ("x",
    over_App (over_Lam ("g",
                    over_App (over_Lam ("h",
                                    over_App (over_Var "h",
                                          over_App (over_Var "g",
                                                under_App (over_Var "f",
                                                       over_Var "x")))),
                          over_Lam ("j",
                                    over_App (over_Var "j",
                                          under_Lam ("a", over_Var "a"))))),
              over_Lam ("y", under_Let ("y1",
                                 over_Var "y",
                                 over_Lam ("z", over_Var "z"))))))
```

Figure 20: The example, annotated by $\mathcal{A}'$ — `exp'`

```
Lam ("g_12",
     Lam ("g_13",
          Let ("g_14", App (Var "g_12", Var "g_13"), Lam ("g_15", Var "g_15"))))
```

Figure 21: The result of specializing `exp'`

Princeton, New Jersey, March 1994.

[3] Henry G. Baker. CONS should not CONS its arguments, or, a lazy alloc is a smart alloc. *ACM SIGPLAN Notices*, 27(3):24–34, March 1992.

[4] 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.

[5] Lars Birkedal and Morten Welinder. Handwriting program generator generators. In Manuel Hermenegildo and Jaan Penjam, editors, *Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 198–214, Madrid, Spain, September 1994.

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

[7] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[8] Anders Bondorf and Dirk Dussart. Improving CPS-based partial evaluation: Writing cogen by hand. In Peter Sestoft and Harald Søndergaard, editors, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical Report 94/9, University of Melbourne, Australia, pages 1–10, Orlando, Florida, June 1994.

[9] Robert (Corky) Cartwright, editor. *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, Snowbird, Utah, July 1988.

[10] 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.

```
structure PE_Val =
  struct
    open PE_Exp

    exception AnnotError of string

    datatype int_d = Text_d of residual
                   | Value_d of int_d -> int_d

    fun is_text_d (Text_d t) = t
      | is_text_d _ = raise AnnotError "Text_d expected"

    fun is_value_d (Value_d v) = v
      | is_value_d _ = raise AnnotError "Value_d expected"


    datatype int_c = Text_c of residual
                   | Value_c of int_c -> (int_c -> int_c) -> int_c

    fun is_text_c (Text_c t) = t
      | is_text_c _ = raise AnnotError "Text_c expected"

    fun is_value_c (Value_c v) = v
      | is_value_c _ = raise AnnotError "Value_c expected"
  end
```

Figure 22: Intermediate values during specialization

```
structure PE_Env =
  struct
    exception Undeclared_Variable of string

    fun init_env x = raise Undeclared_Variable x

    fun extend_env (r, i, v) x = if x = i then v else r x
  end
```

Figure 23: Environment module

```
structure PE_Aux =
  struct
    val fresh = let val x = ref 1
                in fn () => (x := !x + 1; "g_" ^ (makestring (!x)))
                end
  end
```

Figure 24: Generator of fresh variables

```
structure PEd =
  struct

    open PE_Exp PE_Val PE_Env PE_Aux

    fun Sd (over_Var i) rho =
          rho i
      | Sd (over_Lam (i, e)) rho =
          Value_d (fn w => Sd e (extend_env (rho, i, w)))
      | Sd (over_App (e_0, e_1)) rho =
          (is_value_d (Sd e_0 rho)) (Sd e_1 rho)
      | Sd (over_Let (i, e_1, e_2)) rho =
          Sd e_2 (extend_env (rho, i, (Sd e_1 rho)))
      | Sd (under_Lam (i, e)) rho =
          let val m = fresh ()
          in Text_d (Lam (m,
                          is_text_d (Sd e (extend_env (rho, i, Text_d (Var m))))))
          end
      | Sd (under_App (e_0, e_1)) rho =
          Text_d (App (is_text_d (Sd e_0 rho),
                       is_text_d (Sd e_1 rho)))
      | Sd (under_Let (i, e_1, e_2)) rho =
          let val m = fresh ()
          in Text_d (Let (m,
                          is_text_d (Sd e_1 rho),
                          is_text_d (Sd e_2 (extend_env (rho, i, Text_d (Var m))))))
          end

      fun specialize e = is_text_d (Sd e init_env)

  end
```

Figure 25: Direct-style specializer in Standard ML (Figure 4)

```
structure PEc =
  struct

    open PE_Exp PE_Val PE_Env PE_Aux

    fun Sc (over_Var i) rho kappa =
          kappa (rho i)
      | Sc (over_Lam (i, e)) rho kappa =
          kappa (Value_c (fn w => fn k => Sc e (extend_env (rho, i, w)) k))
      | Sc (over_App (e_0, e_1)) rho kappa =
          Sc e_0
             rho
             (fn v_0 => Sc e_1
                           rho
                           (fn v_1 => is_value_c v_0 v_1 kappa))
      | Sc (over_Let (i, e_1, e_2)) rho kappa =
          Sc e_1 rho (fn v => Sc e_2 (extend_env (rho, i, v)) kappa)
      | Sc (under_Lam (i, e)) rho kappa =
          let val m = fresh ()
          in Sc e
                (extend_env (rho, i, Text_c (Var m)))
                (fn v => kappa (Text_c (Lam (m, is_text_c v))))
          end
      | Sc (under_App (e_0, e_1)) rho kappa =
          Sc e_0
             rho
             (fn v_0 => Sc e_1
                           rho
                           (fn v_1 => kappa (Text_c (App (is_text_c v_0, is_text_c v_1)))))
      | Sc (under_Let (i, e_1, e_2)) rho kappa =
          let val m = fresh ()
          in Sc e_1
                rho
                (fn v => Sc e_2
                            (extend_env (rho, i, (Text_c (Var m))))
                            (fn w => kappa (Text_c (Let (m, is_text_c v, is_text_c w)))))
          end

    fun specialize e = is_text_c (Sc e init_env (fn a => a))

  end
```

Figure 26: Continuation-style specializer in Standard ML (Figure 6)

```
structure PEd' =
  struct

    open PE_Exp PE_Val PE_Env PE_Aux intermediate_ctrl

    fun Sd' (over_Var i) rho =
          rho i
      | Sd' (over_Lam (i, e)) rho =
          Value_d (fn w => Sd' e (extend_env (rho, i, w)))
      | Sd' (over_App (e_0, e_1)) rho =
          is_value_d (Sd' e_0 rho) (Sd' e_1 rho)
      | Sd' (over_Let (i, e_1, e_2)) rho =
          Sd' e_2 (extend_env (rho, i, (Sd' e_1 rho)))
      | Sd' (under_Lam (i, e)) rho =
          let val m = fresh ()
          in Text_d (Lam (m,
                          is_text_d (reset (fn () =>
                                        Sd' e
                                          (extend_env (rho,
                                                       i,
                                                       (Text_d (Var m))))))))
          end
      | Sd' (under_App (e_0, e_1)) rho =
          Text_d (App (is_text_d (Sd' e_0 rho),
                       is_text_d (Sd' e_1 rho)))
      | Sd' (under_Let (i, e_1, e_2)) rho =
          shift (fn kappa =>
                   let val m = fresh ()
                   in Text_d (Let (m,
                                   is_text_d (Sd' e_1 rho),
                                   is_text_d (reset (fn () =>
                                                kappa (Sd' e_2
                                                         (extend_env (rho,
                                                                      i,
                                                                      Text_d (Var m))))))))
                end)

    fun specialize e = is_text_d (Sd' e init_env)

  end
```

Figure 27: Direct-style, continuation-based specializer in Standard ML (Figure 9)

```
structure PEc' =
  struct

    open PE_Exp PE_Val PE_Env PE_Aux

    fun Sc' (over_Var i) rho kappa =
          kappa (rho i)
      | Sc' (over_Lam (i, e)) rho kappa =
          kappa (Value_c (fn w => fn k => Sc' e (extend_env (rho, i, w)) k))
      | Sc' (over_App (e_0, e_1)) rho kappa =
          Sc' e_0
              rho
              (fn v_0 => Sc' e_1
                             rho
                             (fn v_1 => is_value_c v_0 v_1 kappa))
      | Sc' (over_Let (i, e_1, e_2)) rho kappa =
          Sc' e_1 rho (fn v => Sc' e_2 (extend_env (rho, i, v)) kappa)
      | Sc' (under_Lam (i, e)) rho kappa =
          let val m = fresh ()
          in kappa (Text_c (Lam (m,
                                 is_text_c (Sc' e
                                                (extend_env (rho,
                                                             i,
                                                             Text_c (Var m)))
                                                (fn x => x)))))
          end
      | Sc' (under_App (e_0, e_1)) rho kappa =
          Sc' e_0
              rho
              (fn v_0 => Sc' e_1
                             rho
                             (fn v_1 => kappa (Text_c (App (is_text_c v_0,
                                                            is_text_c v_1)))))
      | Sc' (under_Let (i, e_1, e_2)) rho kappa =
          let val m = fresh ()
          in Sc' e_1
                 rho
                 (fn v => Text_c (Let (m,
                                       is_text_c v,
                                       is_text_c (Sc' e_2
                                                      (extend_env (rho,
                                                                   i,
                                                                   Text_c (Var m)))
                                                      kappa))))
          end

    fun specialize e = is_text_c (Sc' e init_env (fn a => a))

  end
```

Figure 28: Continuation-passing, continuation-based specializer in Standard ML (Figure 7)

```
structure PEd'_PEd' =
  struct

    open PE_Exp PE_Val PE_Aux residual_ctrl

    fun Sd'_Sd' (over_Var i) =
          Var i
      | Sd'_Sd' (over_Lam (i, e)) =
          Lam (i, reset (fn () => Sd'_Sd' e))
      | Sd'_Sd' (over_App (e_1, e_2)) =
          App (complete e_1, complete e_2)
      | Sd'_Sd' (over_Let (i, e_1, e_2)) =
          shift (fn k => Let (i,
                              complete e_1,
                              reset (fn () => k (Sd'_Sd' e_2))))
      | Sd'_Sd' (under_Lam (i, e)) =
          let val m = fresh ()
          in Reset (Let (m,
                         Op0 "Fresh",
                         Let (i,
                              Op1 ("Var*", Var m),
                              Op2 ("Lam*",
                                   Var m,
                                   Reset (reset (fn () => Sd'_Sd' e))))))
          end
      | Sd'_Sd' (under_App (e_1, e_2)) =
          Op2 ("App*", complete e_1, complete e_2)
      | Sd'_Sd' (under_Let (i, e_1, e_2)) =
          let val k_1 = fresh ()
          in shift (fn k =>
                    Shift (k_1,
                           let val m = fresh ()
                           in Let (m,
                                   Op0 "Fresh",
                                   Let (i,
                                        Op1 ("Var*", Var m),
                                        Op3 ("Let*",
                                             Var m,
                                             complete e_1,
                                             Reset (App (Var k_1,
                                                         reset (fn () =>
                                                                k (Sd'_Sd' e_2)))))))
                           end))
          end
    and complete e =
          let val a = fresh ()
          in shift (fn k => Let (a, Sd'_Sd' e, k (Var a)))
          end

    fun specialize e = Sd'_Sd' e

  end
```

Figure 29: Direct-style, continuation-based pecom in Standard ML (Figure 15)

```
structure PEc'_PEc' =
  struct

    open PE_Exp PE_Val PE_Aux

    fun Sc'_Sc' (over_Var i) kappa =
          kappa (Var i)
      | Sc'_Sc' (over_Lam (i, e)) kappa =
          let val n = fresh ()
          in kappa (Lam (i, Lam (n, Sc'_Sc' e (fn v => App (Var n, v)))))
          end
      | Sc'_Sc' (over_App (e_1, e_2)) kappa =
          let val n = fresh ()
          in Sc'_Sc' e_1
                     (fn v => Sc'_Sc' e_2
                                       (fn w => App (App (v, w),
                                                     Lam (n, kappa (Var n)))))
          end
      | Sc'_Sc' (over_Let (i, e_1, e_2)) kappa =
          Sc'_Sc' e_1 (fn v => Let (i, v, Sc'_Sc' e_2 kappa))
      | Sc'_Sc' (under_Lam (i, e)) kappa =
          let val m = fresh ()
          in kappa (Let (m,
                         Op0 "Fresh",
                         Let (i,
                              Op1 ("Var*", (Var m)),
                              Op2 ("Lam*",
                                   Var m,
                                   Sc'_Sc' e (fn x => x)))))
          end
      | Sc'_Sc' (under_App (e_1, e_2)) kappa =
          Sc'_Sc' e_1
                  (fn v => Sc'_Sc' e_2
                                  (fn w => kappa (Op2 ("App*", v, w))))
      | Sc'_Sc' (under_Let (i, e_1, e_2)) kappa =
          let val m = fresh ()
          in Sc'_Sc' e_1
                     (fn v => Let (m,
                                   Op0 "Fresh",
                                   Let (i,
                                        Op1 ("Var*", Var m),
                                        Op3 ("Let*",
                                             Var m,
                                             v,
                                             Sc'_Sc' e_2 kappa))))
          end

    fun specialize e = Sc'_Sc' e (fn a => a)

  end
```

Figure 30: Continuation-style, continuation-based pecom in Standard ML (Figure 14)

```
structure CPS =
  struct

    open PE_Exp PE_Aux

    fun C (Var i) kappa =
          kappa (Var i)
      | C (Lam (i, e)) kappa =
          let val k = fresh ()
          in kappa (Lam (i, (Lam (k, C e (fn v => App (Var k, v))))))
          end
      | C (App (e_0, e_1)) kappa =
          let val v = fresh ()
          in C e_0 (fn v_0 =>
                    C e_1 (fn v_1 =>
                           App (App (v_0, v_1), Lam (v, kappa (Var v)))))
          end
      | C (Op0 p_0) kappa =
          kappa (Op0 p_0)
      | C (Op1 (p_1, e_1)) kappa =
          C e_1 (fn v_1 => kappa (Op1 (p_1, v_1)))
      | C (Op2 (p_2, e_1, e_2)) kappa =
          C e_1 (fn v_1 =>
                 C e_2 (fn v_2 =>
                        kappa (Op2 (p_2, v_1, v_2))))
      | C (Op3 (p_3, e_1, e_2, e_3)) kappa =
          C e_1 (fn v_1 =>
                 C e_2 (fn v_2 =>
                        C e_3 (fn v_3 =>
                               kappa (Op3 (p_3, v_1, v_2, v_3)))))
      | C (Let (i, e_1, e_2)) kappa =
          C e_1 (fn v_1 => Let (i, v_1, C e_2 kappa))
      | C (Shift (c, e)) kappa =
          let val v = fresh ()
              val k = fresh ()
          in App (Lam (c, C e (fn a => a)),
                  Lam (v, Lam (k, App (Var k, kappa (Var v)))))
          end
      | C (Reset e) kappa =
          kappa (C e (fn a => a))

    fun transform e = C e (fn a => a)

  end
```

Figure 31: The CPS transformation in Standard ML (Figure 8)

```
signature ESCAPE =
  sig
    type void
    val coerce : void -> 'a
    val escape : (('1a -> void) -> '1a) -> '1a
  end

structure Escape : ESCAPE =
  struct
    datatype void = VOID of void
    fun coerce (VOID v) = coerce v
    fun escape f = callcc (fn k => f (fn x => throw k x))
  end

signature CONTROL =
  sig
    type ans
    val reset : (unit -> ans) -> ans
    val shift : (('1a -> ans) -> ans) -> '1a
  end

functor Control (type ans) : CONTROL =
  struct
    open Escape
    exception MissingReset
    val mk : (ans -> void) ref = ref (fn _ => raise MissingReset)
    fun abort x = coerce (!mk x)
    type ans = ans
    fun reset t = escape (fn k => let val m = !mk
                                  in mk := (fn r => (mk := m; k r));
                                     abort (t ())
                                  end)
    fun shift h = escape (fn k =>
                          abort (h (fn v =>
                                    reset (fn () =>
                                           coerce (k v)))))
  end


structure residual_ctrl = Control (type ans = PE_Exp.residual)

structure intermediate_ctrl = Control (type ans = PE_Val.int_d)
```

Figure 32: Shift and reset in Standard ML of New Jersey

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

[12] William Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for continuations. In Cartwright [9], pages 124–131.

[13] Charles Consel. Polyvariant binding-time analysis for applicative languages. In Schmidt [55], pages 66–77.

[14] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In Schmidt [55], pages 145–154.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In Clinger [10], pages 299–310.

[21] Olivier Danvy and Karoline Malmkjær. Intensions and extensions in a reflective tower. In Cartwright [9], pages 327–341.

[22] Dirk Dussart. Personal communication, Pittsburgh, Pennsylvania, July 1993.

[23] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, 1987.

[24] 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.

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

[26] 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.

[27] 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.

[28] Daniel P. Friedman. Personal e-mail communication, September 1993.

[29] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* The MIT Press and McGraw-Hill, 1991.

[30] Yoshihito Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls 2, 5*, pages 45–50, 1971.

[31] Carsten K. Gomard. A self-applicable partial evaluator for the lambda-calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, April 1992.

[32] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

[33] 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.

[34] Carsten K. Holst. Syntactic currying: yet another approach to partial evaluation. Student Report 89-7-6, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, July 1989.

[35] 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.

[36] Neil D. Jones. Automatic program specialization: A re-examination from basic principles. In *Partial Evaluation and Mixed Computation*, pages 225–282. North-Holland, 1988.

[37] Neil D. Jones, Carsten K. Gomard, Anders Bondorf, Olivier Danvy, and Torben Æ. Mogensen. A self-applicable partial evaluator for the lambda calculus. In K. C. Tai and Alexander L. Wolf, editors, *IEEE International Conference on Computer Languages*, pages 49–58, New Orleans, Louisiana, 1990.

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

[39] 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.

[40] Stephen C. Kleene. *Introduction to Metamathematics.* D. van Nostrand, Princeton, New Jersey, 1952.

[41] David Kranz, Richard Kesley, Jonathan Rees, Paul Hudak, Jonathan Philbin, and Norman Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN'86 Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986.

[42] Julia L. Lawall. *Continuation Introduction and Elimination in Higher-Order Programming Languages.* PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, USA, July 1994.

[43] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In Carolyn L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 1994. ACM Press.

[44] 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.

[45] Karoline Malmkjær. Towards efficient partial evaluation. In Schmidt [55], pages 33–43.

[46] Karoline Malmkjær, Nevin Heintze, and Olivier Danvy. ML partial evaluation using set-based analysis. In John Reppy, editor, *Proceedings of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, Orlando, Florida, June 1994. Also appears as Technical report CMU-CS-94-129.

[47] Drew McDermott. An efficient environment allocation scheme in an interpreter for a lexically-scoped Lisp. In Ruth E. Davis and John R. Allen, editors, *Conference Record of the 1980 LISP Conference*, pages 154–162, Stanford, California, August 1980.

[48] 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.

[49] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[50] Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):347–363, 1993.

[51] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[52] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, 1972.

[53] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3/4):289–360, December 1993.

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

[55] 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.

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

[57] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

[58] 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.