Sound Specialization in the Presence of Computational Effects

Julia L. Lawall[1] and Peter Thiemann[2]

[1] IRISA/INRIA Rennes, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France, jll@irisa.fr
[2] Universität Tübingen, Sand 13, D-72076 Tübingen, Germany, thiemann@informatik.uni-tuebingen.de

Abstract Moggi's computational lambda calculus λ_c is a well-established model of computation. We define a two-level version $\underline{\lambda}_c$ of the computational lambda calculus and demonstrate that it is an inevitable description for sound specialization. We implement the calculus in terms of a standard two-level lambda calculus via a continuation-passing style transformation. This transformation is sound and complete with respect to λ_c; it forms a reflection in the two-level lambda calculus of $\underline{\lambda}_c$. As a practical ramification of this work we show that several published specialization algorithms are unsound and develop a sound specializer similar to continuation-based specializers.

Keywords: lambda calculus, partial evaluation, functional programming, continuations.

1 Introduction

What is a correct partial evaluator for the lambda calculus? Most of the work addressing this question has focused on the relationship between the binding-time analysis and the specializer [20, 21, 23, 30, 32, 38] using a variety of models (structured operational semantics, denotational semantics, reduction systems, logical frameworks). In essence, they prove that a specializer processing a well-annotated program cannot confuse code with other values.

Another aspect of the correctness of a partial evaluator is soundness. It is well known that traditional specialization is not sound for the call-by-value lambda calculus, since beta reduction can discard dynamic computation. Such a reduction is unsound in the presence of nontermination, and in extensions of such a specializer to languages with side effects. The well-known solution is to explicitly insert let expressions in a preprocessing pass. By itself, this solution adversely affects binding times, requiring users to perform so-called *control-based* binding-time improvements.

Hatcliff and Danvy [23] have shown that a specializer based on Moggi's computational meta-language is sound without necessarily deteriorating binding-times. They demonstrated that the monadic theory of the language provides a formal justification for let-insertion, and expresses the essence of control-based binding-time improvements. While their emphasis is on setting up an evaluation-order independent framework for specialization, our approach is more modest in that we consider the restriction of the computational meta-language to the image of the call-by-value translation: the computational lambda calculus λ_c. This possibility has also been noted but not pursued by Hatcliff and Danvy. Both approaches yield a sound specializer for all monadic models of the calculus. Such a specializer can be used to specialize programs written in languages with arbitrary monadic effects (nontermination, state, exceptions, control, nondeterminism).

Recently, traditional specialization based on the lambda calculus has been extended to manipulate continuations in order to improve binding times (*continuation-based specialization*) [3,5,9,25,37]. We show that several of these approaches do not extend straightforwardly to specializers that are sound in all monadic models. We then present a new continuation-passing style (CPS) transformation on the two-level computational lambda calculus that, when composed with a traditional specializer, produces a specializer that is sound and complete with respect to Moggi's computational lambda calculus, and has good binding-time properties. This implementation is strongly tied to $\underline{\lambda_c}$ since the two-level CPS transformation and its associated direct-style transformation form a reflection of the computational lambda calculus-specializer in the traditional specialization for the lambda calculus. A reflection is a strengthened version of a Galois connection (where we regard reduction as a preorder) [35].

One of the key points of this work is its immediate applicability. Whereas Hatcliff and Danvy's work requires the construction of a new specializer along their operational rules, we show that sound specialization can be achieved by first transforming the annotated term to CPS and then using a straightforward specializer. The sound specializer may also be constructed mechanically by symbolic composition of our CPS transformation with specialization. This results in a specializer similar to previously proposed continuation-based specializers, but which is sound in the presence of computational effects. $\underline{\lambda_c}$ gives a formal basis for proving the soundness of such a specializer.

The rest of this paper is organized as follows. After a brief introduction to partial evaluation in Section 2 we present in Section 3 the traditional two-level call-by-value lambda calculus [30, 32] and define specialization in terms of reduction rules. We demonstrate that this specializer is not sound and propose to achieve soundness by adding extra reduction rules that essentially perform let-insertion on the fly. Section 4 recasts the improved specializer in terms of the computational lambda calculus. This specializer is *well-behaved* in the sense that it has good binding-time properties like a continuation-based specializer. Next, in Section 5 we examine the previous approaches to describe and implement well-behaved specialization, assess their soundness, and define a novel sound approach. In Section 6, we present a new variant of the CPS transformation that results in a specializer that is both sound and well-behaved and produces residual programs that are in direct style. Extending Sabry and Wadler's result for the standard call-by-value lambda calculus [35], we show in Section 7 that our CPS transformation on two-level terms forms a reflection in the two-level computational lambda calculus of a restriction of the two-level lambda calculus, ensuring that reduction in a specializer built according to our construction corresponds exactly to reduction in the two-level computational lambda calculus. Putting our results into practice, Section 8 defines a stand-alone specializer based on $\underline{\lambda}_c$. Finally, Section 9 discusses related work and Section 10 concludes and considers directions for future work.

2 Partial Evaluation

Partial evaluation [24] is an automatic program transformation technique aimed at staging computation. In the typical case, a source program `p` has two inputs `ins`—which is known early (at binding time *static*)—and `ind`—which is known late (at binding time *dynamic*). Partial evaluation specializes `p` with respect to `ins`. The result is a *residual* program `p'`. When `p'` is applied to `ind`, the result is the same as applying `p` to the whole input `ins` and `ind`.

In *offline* partial evaluation, a binding-time analysis propagates throughout the whole program the information about which inputs are static. Binding-time analysis annotates each construct in the source program with a binding time, indicating whether or not it should be reduced during specialization. The specializer is then merely a reducer of statically annotated constructs.

3 Two-Level Lambda Calculus

The annotations of the binding-time analysis can be conveniently expressed using the notation of the Nielsons' two-level lambda calculus [20, 31]. Specialization is then simply reduction in the two-level lambda calculus. $\lambda\beta_V$, the standard call-by-value lambda calculus with β-conversion, is the traditional model of computation for strict functional languages [8, 26, 33]. Thus, to describe partial evaluation, we begin with $\underline{\lambda}\beta_V$, a two-level lambda calculus based on $\lambda\beta_V$.

To describe $\underline{\lambda}\beta_V$, and subsequent calculi T, throughout the paper, we use the following notation. Let $\Rightarrow$ denote the compatible closure of $\longrightarrow$, including reduction under lambda abstractions. Let $\Rrightarrow$ denote the reflexive and transitive closure of $\Rightarrow$. Equality, $=$, is the symmetric closure of $\Rrightarrow$, i.e., the equivalence relation generated by it. Reduction in the calculus T is indicated by $T \vdash \ldots$. We omit this qualification if T is clear from the context. Particular to two-level lambda calculi, we let $|E|$ denote the standard lambda term resulting from E by removing all annotations. Throughout, we assume the variable convention, i.e., the sets of bound and free variables of each term are disjoint.

3.1 The Calculus $\underline{\lambda}\beta_V$

Figure 1 defines the syntax and the single step reduction rule of the calculus $\underline{\lambda}\beta_V$. It is a traditional two-level call-by-value lambda calculus, where underlining corresponds to a dynamic binding time and the absence of underlining corresponds

$$
\begin{array}{lll}
\text{Terms} & E & ::= V \mid N \\
\text{Values} & V & ::= x \mid \lambda x.E \mid \underline{\lambda} x.E \mid E \underline{@} E \\
\text{Non-Values} & N & ::= E @ E
\end{array}
$$

$$(\lambda x.E)@V \longrightarrow E[x := V] \tag{1}$$

Figure1. The Calculus $\underline{\lambda}\beta_V$

to a static binding time. Intuitively, the underlined constructs denote primitive operations that construct code. whereas the other constructs have their usual meaning.

The standard notion of computation in $\lambda\beta_V$ tries to reduce a term to a value. Terms that cannot be reduced to a value are called *stuck terms*, e.g., $a@b$. In $\underline{\lambda}\beta_V$ there are additional possibilities for terms to get stuck. For example, the term $(\underline{\lambda}x.E)@V$ is not a value, but it cannot be reduced further. In contrast, a dynamic redex like $(\underline{\lambda}x.E)\underline{@}V$ is not a stuck term, because it is a value according to our definition. It appears unreduced in the residual code.

One facet of the correctness of a binding-time analysis is to ensure that no stuck terms arise during specialization [20, 21, 23, 30, 32, 38]. Our development is independent from the goal of preventing stuck terms. Therefore, we refrain from specifying such an analysis here.

3.2 Conservativeness

What do we expect from a sound partial evaluator? The minimal condition should be that $\underline{\lambda}\beta_V$ is *conservative* with respect to $\lambda\beta_V$. That is:

Definition 1. A two-level calculus T is *conservative* with respect to its one-level version $|T|$ (which is obtained by dropping all annotations from terms and rules) if for terms E_1, E_2 of T we have

$$T \vdash E_1 = E_2 \qquad \Rightarrow \qquad |T| \vdash |E_1| = |E_2|.$$

Observation 2. $\underline{\lambda}\beta_V$ is not conservative with respect to $\lambda\beta_V$.

Proof. Consider

$$
\begin{array}{l}
E_1 = (\lambda x.\underline{\lambda} z.z)@((\underline{\lambda} x.x\underline{@}x)\underline{@}(\underline{\lambda} x.x\underline{@}x)) \\
E_2 = \underline{\lambda} z.z
\end{array}
$$

Clearly, $\underline{\lambda}\beta_V \vdash E_1 = E_2$ but in $\lambda\beta_V$ $|E_1|$ diverges whereas $|E_2|$ is in normal form.

3.3 The Calculus $\underline{\lambda}\beta'_V$

$\underline{\lambda}\beta_V$ is not conservative with respect to $\lambda\beta_V$ because a static beta reduction can discard or duplicate code in the argument representing an application. The unannotated counterpart of such code is not a value in $\lambda\beta_V$, which thus does not reduce the corresponding redex. Therefore we restrict the notion of value in $\underline{\lambda}\beta_V$ so that code representing an application is no longer included, producing the calculus $\underline{\lambda}\beta'_V$.

$$
\begin{array}{lrl}
\text{Terms} & E & ::= V \mid N \\
\text{Values} & V & ::= x \mid \lambda x.E \mid \underline{\lambda} x.E \\
\text{Non-Values} & N & ::= E@E \mid E\underline{@}E
\end{array}
$$

This change has a profound effect on the reduction properties of terms. Consider an application of a lambda abstraction to an argument that reduces to code representing an application. Since such code no longer reduces to a value, the whole term is stuck because the top-level application cannot be reduced. In particular, the body of the lambda abstraction is not available to the context for further reduction. For example, the following term is stuck in $\underline{\lambda}\beta'_V$ (but not in $\underline{\lambda}\beta_V$):

$$((\lambda x.\lambda z.z)@(a\underline{@}b))@\underline{\lambda} w.w \tag{2}$$

The abstraction $\lambda z.z$ is not available to its context, the application to $\underline{\lambda} w.w$. In order to avoid stuck terms $\underline{\lambda}\beta'_V$, forces us to annotate everything as code:

$$((\underline{\lambda} x.\underline{\lambda} z.z)\underline{@}(a\underline{@}b))\underline{@}\underline{\lambda} w.w$$

If we view

$$(\underline{\lambda} x.E)\underline{@}(a\underline{@}b)$$

as a let expression

$$\text{let } x = a\underline{@}b \text{ in } E$$

then this annotation is a consequence of the well-formedness criterion found in the textbook by Jones and others [24, sec. 10.1.4]: if the header term $a\underline{@}b$ of the let expression is dynamic then so is E and the entire expression.

As observed by the developers of continuation-based partial evaluation [3,25], fewer subterms can be annotated as static without getting stuck terms because $\underline{\lambda}\beta'_V$ has no rule to propagate the context of a dynamic let expression to its body, even though such a transformation is semantically correct. Therefore a calculus that is sound while keeping the number of stuck terms as low as in $\underline{\lambda}\beta_V$ is highly desirable.

3.4 The Calculus $\underline{\lambda}\beta_V^{\text{let}}$

To this end, we construct the calculus $\underline{\lambda}\beta_V^{\text{let}}$, which includes let expressions "let $x = E$ in E" and let reduction. $\underline{\lambda}\beta_V^{\text{let}}$ also performs certain context reorganizations involving let expressions.

$$
\begin{array}{lrl}
\text{Terms} & E & ::= V \mid N \\
\text{Values} & V & ::= x \mid \lambda x.E \mid \underline{\lambda} x.E \\
\text{Non-Values} & N & ::= E@E \mid E\underline{@}E \mid \text{let } x = E \text{ in } E
\end{array}
$$

$$(\lambda x.E)@V \longrightarrow E[x := V] \tag{3}$$
$$\text{let } x = V \text{ in } E \longrightarrow E[x := V] \tag{4}$$
$$\text{let } y = (\text{let } x = e_1 \text{ in } e_2) \text{ in } e_3 \longrightarrow \text{let } x = e_1 \text{ in } (\text{let } y = e_2 \text{ in } e_3) \tag{5}$$
$$N@E \longrightarrow \text{let } x = N \text{ in } x@E \tag{6}$$
$$V@N \longrightarrow \text{let } x = N \text{ in } V@x \tag{7}$$

The resulting calculus $\underline{\lambda}\beta_V^{\text{let}}$ does not have the problems of $\underline{\lambda}\beta_V'$ with stuck terms and allows further reduction of the term (2).

$$\begin{array}{l}
(\text{let } x = a\underline{@}b \text{ in } \lambda z.z)@\underline{\lambda}w.w \\
\Rightarrow \text{let } y = (\text{let } x = a\underline{@}b \text{ in } \lambda z.z) \text{ in } y@\underline{\lambda}w.w \\
\Rightarrow \text{let } x = a\underline{@}b \text{ in let } y = \lambda z.z \text{ in } y@\underline{\lambda}w.w \\
\Rightarrow \text{let } x = a\underline{@}b \text{ in } (\lambda z.z)@\underline{\lambda}w.w \\
\Rightarrow \text{let } x = a\underline{@}b \text{ in } \underline{\lambda}w.w
\end{array}$$

In this calculus $\underline{\lambda}\beta_V^{\text{let}}$ let-expressions can float from any place where they might impede a reduction. Hence the calculus allows for a certain amount of context reorganization.

Lemma 3. *1.* $\underline{\lambda}\beta_V^{\text{let}} \vdash (\text{let } x = E_1 \text{ in } E_2)@E_3 \;=\; \text{let } x = E_1 \text{ in } E_2@E_3$*;*
2. $\underline{\lambda}\beta_V^{\text{let}} \vdash V@(\text{let } x = E_1 \text{ in } E_2) \;=\; \text{let } x = E_1 \text{ in } V@E_2$*;*
3. $\underline{\lambda}\beta_V^{\text{let}} \vdash \text{let } y = (\text{let } x = E_1 \text{ in } E_2) \text{ in } E_3 \;=\; \text{let } x = E_1 \text{ in } (\text{let } y = E_2 \text{ in } E_3)$*.*

Proof.

1. $(\text{let } x = E_1 \text{ in } E_2)@E_3$
$$\begin{array}{l}
\Rightarrow_6 \text{let } y = (\text{let } x = E_1 \text{ in } E_2) \text{ in } y@E_3 \\
\Rightarrow_5 \text{let } x = E_1 \text{ in let } y = E_2 \text{ in } y@E_3 \\
\Leftarrow_6 \text{let } x = E_1 \text{ in } E_2@E_3
\end{array}$$

2. $V@(\text{let } x = E_1 \text{ in } E_2)$
$$\begin{array}{l}
\Rightarrow_7 \text{let } y = (\text{let } x = E_1 \text{ in } E_2) \text{ in } V@y \\
\Rightarrow_5 \text{let } x = E_1 \text{ in let } y = E_2 \text{ in } V@y \\
\Leftarrow_6 \text{let } x = E_1 \text{ in } V@E_2
\end{array}$$

3. $\text{let } y = (\text{let } x = E_1 \text{ in } E_2) \text{ in } E_3 \Rightarrow_5 \text{let } x = E_1 \text{ in } (\text{let } y = E_2 \text{ in } E_3)$

Unfortunately, $\underline{\lambda}\beta_V^{\text{let}}$ is again not conservative with respect to $\lambda\beta_V$. The equalities that we have just proved and also the rules 5, 6, and 7 are not provable in $\lambda\beta_V+(4)$ because $\lambda\beta_V$ lacks rules to reorganize contexts. For example, in $\underline{\lambda}\beta_V^{\text{let}}$

$$\text{let } y = (\text{let } x = z@z \text{ in } x) \text{ in } y$$

and

$$\text{let } x = z@z \text{ in } (\text{let } y = x \text{ in } y)$$

are equal. In $\lambda\beta_V$, the former is in normal form, whereas the latter reduces further to

$$\text{let } x = z@z \text{ in } x$$

Thus in $\lambda\beta_V$, the terms are not equal.

$$
\begin{array}{ll}
\text{Terms} & e ::= v \mid n \\
\text{Values} & v ::= x \mid \lambda x.e \\
\text{Non-Values} & n ::= e@e \mid \text{let } x = e \text{ in } e
\end{array}
$$

$$
\begin{aligned}
(\lambda x.e)@v &\longrightarrow e[x := v] && (8)\\
\text{let } x = v \text{ in } e &\longrightarrow e[x := v] && (9)\\
\text{let } y = (\text{let } x = e_1 \text{ in } e_2) \text{ in } e_3 &\longrightarrow \text{let } x = e_1 \text{ in let } y = e_2 \text{ in } e_3 && (10)\\
n@e &\longrightarrow \text{let } x = n \text{ in } x@e && (11)\\
v@n &\longrightarrow \text{let } x = n \text{ in } v@x && (12)
\end{aligned}
$$

Figure2. Computational Lambda Calculus λ_c

$$
\begin{array}{ll}
\text{Terms} & E ::= V \mid N \\
\text{Values} & V ::= x \mid \lambda x.E \mid \underline{\lambda} x.E \\
\text{Non-Values} & N ::= E@E \mid \text{let } x = E \text{ in } E \mid E\underline{@}E
\end{array}
$$

$$
\begin{aligned}
(\lambda x.E)@V &\longrightarrow E[x := V] && (13)\\
\text{let } x = V \text{ in } E &\longrightarrow E[x := V] && (14)\\
\text{let } y = (\text{let } x = E_1 \text{ in } E_2) \text{ in } E_3 &\longrightarrow \text{let } x = E_1 \text{ in let } y = E_2 \text{ in } E_3 && (15)\\
N@E &\longrightarrow \text{let } x = N \text{ in } x@E && (16)\\
V@N &\longrightarrow \text{let } x = N \text{ in } V@x && (17)\\
N\underline{@}E &\longrightarrow \text{let } x = N \text{ in } x\underline{@}E && (18)\\
V\underline{@}N &\longrightarrow \text{let } x = N \text{ in } V\underline{@}x && (19)
\end{aligned}
$$

Figure3. Two-Level Computational Lambda Calculus $\underline{\lambda}_c$

4 Two-Level Computational Lambda Calculus

A calculus describing context manipulations around let expressions is already known: Moggi's computational lambda calculus λ_c [28]. The calculus is defined in Figure 2. The reduction rules are taken from work of Sabry and Wadler [35].

Guided by λ_c, we complete the calculus $\underline{\lambda}\beta_V^{\text{let}}$ to $\underline{\lambda}_c$, the two-level computational lambda calculus. Figure 3 shows the reduction rules.

Let $\Rightarrow_0$ denote reduction by rules 18 and 19.

Proposition 4. *$\Rightarrow_0$ is strongly normalizing: every sequence of $\Rightarrow_0$ reductions terminates.*

Proposition 5. *$\Rightarrow_0$ is Church-Rosser: if $E \Rrightarrow_0 E_1$ and $E \Rrightarrow_0 E_2$ then there exists E_3 such that $E_1 \Rrightarrow_0 E_3$ and $E_2 \Rrightarrow_0 E_3$.*

Proof. A case analysis on the relative positions of redexes coupled with a case analysis on reduction rules shows that $\Rightarrow_0$ is weakly Church-Rosser. The proposition then follows by Newman's lemma [1, Proposition 3.1.25] and proposition 4.

Theorem 6. *$\underline{\lambda}_c$ is Church-Rosser.*

Proof. The reduction system for the static part has been shown to be Church-Rosser by Moggi [27]. By proposition 5, $\Rightarrow_0$ is Church-Rosser. A case analysis establishes that $\Rightarrow_0$ commutes with $\Rightarrow$ from λ_c. By the lemma of Hindley and Rosen [1, Proposition 3.3.5], the combined notion of reduction is Church-Rosser.

Lemma 7. *$\underline{\lambda}_c$ is conservative with respect to λ_c.*

Proof. **(13)** is conservative since $|V|$ is a value in λ_c.
(14) idem.
(15) rule is identical to (10).
(16) is conservative since $|N|$ is a non-value in λ_c.
(17) is conservative since $|V|$ is a value and $|N|$ is a non-value in λ_c.
(18) same as 16.
(19) same as 17.

The conservativity of $\underline{\lambda}_c$ implies that static reduction during specialization does not break any equalities provable in λ_c. Technically, λ_c also includes notions of eta-reduction. We do not have eta-reduction in $\underline{\lambda}_c$ because eta-reduction is not commonly used in partial evaluators. There is no problem to add it to the calculus but there is no gain in doing so.

5 Well-behaved Specialization Using CPS Transformation

The two-level computational lambda calculus describes sound yet well-behaved specialization because it is conservative with respect to its basis λ_c which is sound and complete for all monadic models. Most existing specializers are based, however, on $\underline{\lambda}\beta_V$, rather than the two-level computational lambda calculus. In this section we show how a continuation-passing style (CPS) transformation has been used to obtain well-behaved specialization based on $\underline{\lambda}\beta_V$.

5.1 The CPS Transformation

The *continuation* of an expression is a representation of the rest of the computation, which is awaiting the value of the expression. Continuation-passing style (CPS) is an explicit encoding of continuations into a lambda term. In CPS, each continuation is represented as a one-argument function that maps the value of a subexpression to the result of the program.

Transformations have been developed to convert ordinary (*direct style*, or DS) terms into CPS [33] and to convert CPS terms back to DS [10]. Figure 4 shows the CPS transformation for λ_c terms [35].

root expression

$$E^* = E : k$$

serious expressions

$$V : K = K @ V^\dagger$$
$$(\text{let } x = E_1 \text{ in } E_2) : K = E_1 : (\lambda x.E_2 : K)$$
$$(N @ E) : K = N : (\lambda x.(x @ E) : K)$$
$$(V @ N) : K = N : (\lambda x.(V @ x) : K)$$
$$(V_1 @ V_2) : K = V_1{}^\dagger @ V_2{}^\dagger @ K$$

trivial expressions

$$x^\dagger = x$$
$$(\lambda x.E)^\dagger = \lambda x.\lambda k.(E : k)$$

Figure4. Call-by-value CPS Transformation [35]

For partial evaluation, the most interesting aspect of the CPS transformation is that it propagates the context of a let expression into the body. This has led to the use of CPS to implement context reorganizations similar to those performed by λ_c.

5.2 History of Continuation-based Partial Evaluation

Consel and Danvy first observed that if the source program is converted to CPS, static data is never blocked by a dynamic let expression during subsequent specialization [9]. Since their CPS transformation is insensitive to binding times and converts both static and dynamic constructs to CPS, the residual program is in CPS as well.

Bondorf then observed that it is preferable to keep the source program in direct style during binding-time analysis, and to produce a specialized program in direct style [3]. This has several advantages: Binding-time analysis of a direct style source program is less expensive, because there are fewer higher-order functions. The result of binding-time analysis is also easier for the user to interpret. Finally, it is easier for the user to determine whether all the expected specialization has been achieved if the residual program is in the same style as the source program. Bondorf proposed that the effect of converting the source program to CPS could be achieved by manipulating continuations within the specializer [3]. Since the specializer has access to binding times, continuations are only used to propagate the context into static applications. Thus there are no continuations in the residual code.

Dussart and Danvy [11] realized that the continuation manipulation could be factored out of Bondorf's specializer into a CPS transformation of the source program that would follow binding-time analysis. Thus the binding-time analysis is not slowed down by continuations, and the CPS transformation can take advantage of binding times to avoid transforming residual code into CPS. Another innovation [15] was to only propagate the static part of the context into a dynamic let expression. A specializer that only propagates static contexts is also used in Hatcliff and Danvy's specification [23].

5.3 Extending the CPS Transformation to Two-Level Terms

As Dussart and Danvy observed, we can view all of the proposed continuation-based partial evaluators as the composition of a CPS transformation of two-level terms followed by specialization. In this section we describe the CPS transformation corresponding to each approach and assess the soundness of the generated specializer.

Full Context Propagation with Residual Programs in CPS The first approach is to perform the standard call-by-value CPS transformation on unannotated terms [9]. Thus, for comparison with the other systems, we duplicate the rules for static applications and lambda abstractions for the corresponding dynamic terms.

$$(E_1 \underline{@} E_2) : K = E_1 : \lambda a.(E_2 : \lambda b.a \underline{@} b \underline{@} \underline{\lambda} y.K @ y)$$
$$(\underline{\lambda} x.E)^{\dagger} = \underline{\lambda} x.\underline{\lambda} k.(E : k)$$

After the CPS transformation, the argument of every application is an identifier or a lambda abstraction. Therefore, no duplication of code or discarding of computation can happen, and this approach leads to sound specialization. The CPS transformation propagates all (static as well as dynamic) contexts to their uses. More formally, we can prove that specializing the CPS-transformed program implements exactly the reductions of $\underline{\lambda}_c$, if we consider let $x = E_1$ in E_2 as syntactic sugar for $(\lambda x.E_2) @ E_1$. The downside of this approach is that, as noted above, it yields residual programs in CPS. We call this transformation **FC** (**f**ull context propagation with residual programs in **C**PS).

Full Context Propagation with Residual Programs in Direct Style The integration of context propagation into the specializer [3] corresponds to extending the CPS transformation with the following rules:

$$(E_1 \underline{@} E_2) : K = E_1 : \lambda a.(E_2 : \lambda b.K @ (a \underline{@} b))$$
$$(\underline{\lambda} x.E)^{\dagger} = \underline{\lambda} x.(E : \lambda x.x)$$

Like in the **FC** CPS transformation, this transformation propagates both static and dynamic contexts. However, unlike the **FC** transformation, the body of a lambda abstraction is transformed with the identity continuation and no continuation is passed to a residual application. Thus residual terms are not in CPS. We call this transformation **FD0** (**f**ull context propagation with residual programs in **d**irect style).

The **FD0** CPS transformation treats code representing an application as a value, i.e., such code is sent to the continuation. Consequently, the specializer can be shown not to be conservative with respect to $\underline{\lambda}\beta_V$ using the same example as in the proof of Observation 1. The solution to this problem is *let insertion* [4]: before specialization, insert a let expression around the body of every lambda abstraction with a dynamic parameter. These dynamic let expressions introduce names for dynamic arguments and guarantee that the code for the arguments appears in the residual program.

Although let insertion resolves the problem of discarding code representing a non-terminating computation, it does not ensure soundness. The CPS transformation considered in this section can reorder residual computations by reorganizing code. This code reorganization becomes a problem if we extend the specializer to a language with side effects, even if all side effects are to appear in the residual program.

Consider the following expression, in which print is a side-effecting operator that prints its argument:

$$((\underline{\lambda} f.\underline{\text{let}}\ x = \underline{\text{print}@}a\ \underline{\text{in}}\ \underline{\lambda} y.y)\underline{@}h)\underline{@}(\underline{\text{let}}\ y = \underline{\text{print}@}b\ \underline{\text{in}}\ k) \tag{20}$$

This completely dynamic term outputs first a and then b when evaluated (assuming a left-to-right evaluation order). The CPS transformation propagates the residual let expression binding y to the top of the residual expression. Because every expression is annotated as residual, specialization performs no other significant simplifications. Thus the residual code is:

$$\text{let}\ y = \text{print}@b\ \text{in}\ ((\lambda f.\text{let}\ x = \text{print}@a\ \text{in}\ \lambda y.y)@h)@k$$

This residual term incorrectly prints b before a. Here the problem cannot be resolved by let insertion, since no lambda parameter is bound during specialization.

The actual implementation of Similix resolves the problem using an evaluation-order dependency analysis [6]. This solution is, however, ad hoc, and is not documented in the literature.

The same problems arise with all other specializers based on the **FD0** CPS transformation [5, 25, 37].

Static Context Propagation with Residual Programs in Direct Style The **FD0** CPS transformation can be modified in order to avoid propagating dynamic contexts [15]. The corresponding **SD** CPS transformation translates the dynamic constructs as follows:

$$\begin{aligned} (E_1\underline{@}E_2) : K &= K@((E_1 : \lambda x.x)\underline{@}(E_2 : \lambda x.x)) \\ (\underline{\lambda} x.E)^\dagger &= \underline{\lambda} x.(E : \lambda x.x) \end{aligned}$$

Because code representing an application is still treated as a value, specialization based on this CPS transformation is not sound. Again the counterexample is the term used in the proof of Observation 1. As for the **FD0** CPS transformation, let insertion during preprocessing prevents a problem in this case.

The **SD** CPS transformation also reorders computations, although in fewer cases. In particular, the corresponding specializer treats the previous example expression (20) correctly:

$$((\underline{\lambda} f.\underline{\text{let}}\ x = \underline{\text{print}}\,\underline{@}a\ \underline{\text{in}}\ \underline{\lambda} y.y)\underline{@}h)\underline{@}(\underline{\text{let}}\ y = \underline{\text{print}}\,\underline{@}b\ \underline{\text{in}}\ k)$$

Because the outermost application is residual, no context is propagated into the body of the residual let expression binding y. The residual code is thus identical to the source code, and the side-effects are performed in the correct order.

There are, however, expressions in which the **SD** specializer still reorders computation. Consider the following expression (again assuming a left-to-right evaluation order):

$$(\lambda x.\lambda y.f\underline{@}x\underline{@}y)@(\underline{\text{print}}\,\underline{@}a)@(\underline{\text{let}}\ z = \underline{\text{print}}\,\underline{@}b\ \underline{\text{in}}\ k)$$

Here there is no dynamic context to block the propagation of the entire context into the body of the residual let expression. The result of specialization is thus:

$$\text{let } z = \text{print}@b \text{ in } f@(\text{print}@a)@k$$

The side effects are performed out of order. Again, let insertion during preprocessing eliminates the problem. We have proved that this approach is sound in the presence of let insertion. However, we have chosen to concentrate on the approach discussed in the next subsection, which is sound without relying on let insertion.

Let Insertion Considered Unnecessary As a final step, we can avoid the preprocessing step that inserts let expressions. Instead we construct the transformation so that the specializer produces residual programs in A-normal form [18]. A-normal form results from exhaustive application of rules (15), (16), and (17) from Fig. 3. The results is a specializer that is sound without any preprocessing and whose residual programs are well suited to compilation despite being in direct style.

The extension of the CPS transformation to dynamic terms now includes:

$$\begin{aligned}(V_1\underline{@}V_2) : K &= \text{let } y = {V_1}^\dagger\underline{@}{V_2}^\dagger \text{ in } K@y \\ (\underline{\lambda} x.E)^\dagger &= \underline{\lambda} x.(E : (\lambda y.y))\end{aligned}$$

Figure 5 shows the complete transformation. This transformation enables full context propagation and produces residual programs in DS, hence it yields an **FD** specializer. We show the soundness and the completeness of this transformation in Sec. 6.

The residual programs generated with this approach are directly related to **FC** style residual programs. Since the programs are in A-normal form it is only necessary to perform continuation introduction [22]. Therefore, it is straightforward to transfer our results to the **FC** approach.

$$
\begin{array}{ll}
\text{root expression} & \\
E^* & = E : k \\
 & \\
\text{serious expression with continuation} & \\
V : K & = K@V^\dagger \\
(N@E) : K & = N : (\lambda x.(x@E) : K) \\
(V@N) : K & = N : (\lambda x.(V@x) : K) \\
(V_1@V_2) : K & = {V_1}^\dagger@{V_2}^\dagger@K \\
(\text{let } x = E_1 \text{ in } E_2) : K & = E_1 : (\lambda x.E_2 : K) \\
(N\underline{@}E) : K & = N : (\lambda x.(x\underline{@}E) : K) \\
(V\underline{@}N) : K & = N : (\lambda x.(V\underline{@}x) : K) \\
(V_1\underline{@}V_2) : K & = \text{let } y = {V_1}^\dagger\underline{@}{V_2}^\dagger \text{ in } K \downarrow y \\
 & \\
\text{trivial expression, e.g., value} & \\
x^\dagger & = x \\
(\lambda x.E)^\dagger & = \lambda x.\lambda k.(E : k) \\
(\underline{\lambda} x.E)^\dagger & = \underline{\lambda} x.(E : (\lambda y.y)) \\
 & \\
k \downarrow y & = k@y \\
(\lambda x.x) \downarrow y & = (\lambda x.x)@y \\
(\lambda x.M) \downarrow y & = M[x := y]
\end{array}
$$

Figure5. Two-Level CPS Transformation

6 The Two-Level CPS Transformation

The first goal is a simulation result of $\underline{\lambda}_c$ by $\underline{\lambda}\beta_V$ using a two-level (call-by-value) CPS transformation. It is defined in Fig. 5.

Definition 8. (Characterization of 2CPS terms)

continuations
$K ::= k \mid \lambda x.M \mid \lambda x.x$

values
$W ::= x \mid \lambda x.\lambda k.M \mid \underline{\lambda} x.M$

serious expressions
$M ::= K@W \mid W@W@K \mid \text{let } y = W\underline{@}W \text{ in } M$

Observation 9. $\forall E.\exists M.E^* = M$.

The 2CPS reductions restrict the reductions of $\underline{\lambda}\beta_V$ in the following way.

Definition 10. The terms of the calculus $\underline{\lambda}_{cps}$ are the 2CPS terms of Definition 8. The reduction rules of $\underline{\lambda}_{cps}$ are defined by

$$(\lambda x.\lambda k.M)@W@K \longrightarrow M[x := W, k := K] \tag{21}$$

$$(\lambda x.M)@W \longrightarrow M[x := W] \tag{22}$$

This calculus does not reduce $(\lambda x.x)@W$ to W because this reduction would complicate the direct style transformation, which in turn introduces technical complications in the subsequent proofs. For the same reason we disallow the reduction of $(\lambda x.\lambda k.M)@W$ to $\lambda k.M[x := W]$ (cf. [35]).

Reduction in $\underline{\lambda}_{cps}$ of CPS-transformed terms simulates reduction in $\underline{\lambda}_c$.

Lemma 11 (Simulation). *If* $\underline{\lambda}_c \vdash E_1 \Rightarrow E_2$ *then* $\forall K.\underline{\lambda}_{cps} \vdash E_1 : K \Rrightarrow E_2 : K$.

Proof. For the cases not involving dynamic constructs, this has been proved by Sabry and Wadler [35]. It remains to consider the dynamic constructs.

- $\underline{\lambda}_c \vdash N\underline{@}E \Rightarrow \text{let } x = N \text{ in } x\underline{@}E$. Now $(N\underline{@}E) : K = N : (\lambda x.(x\underline{@}E) : K)$ and $(\text{let } x = N \text{ in } x\underline{@}E) : K = N : (\lambda x.(x\underline{@}E) : K)$ which proves the claim.
- $\underline{\lambda}_c \vdash V\underline{@}N \Rightarrow \text{let } x = N \text{ in } V\underline{@}x$. Here $(V\underline{@}N) : K = N : (\lambda x.(V\underline{@}x) : K)$ and $(\text{let } x = N \text{ in } V\underline{@}x) : K = N : (\lambda x.(V\underline{@}x) : K)$ which proves the claim.
- $\underline{\lambda}_c \vdash \underline{\lambda}x.E_1 \Rightarrow \underline{\lambda}x.E_2$. Hence $\underline{\lambda}_c \vdash E_1 \Rightarrow E_2$ and by induction $\underline{\lambda}_{cps} \vdash E_1 : \lambda y.y \Rrightarrow E_2 : \lambda y.y$. Now $(\underline{\lambda}x.E_1) : K = K@\underline{\lambda}x.(E_1 : \lambda y.y) \Rrightarrow K@\underline{\lambda}x.(E_2 : \lambda y.y) = (\underline{\lambda}x.E_2) : K$.

Corollary 12. *The 2CPS transformation maps well-annotated terms to well-annotated terms, i.e., no confused redexes arise during reduction of the image of a well-annotated term.*

Proof. Immediate, by the simulation lemma 11.

Figure 6 show the direct style transformation on two-level terms corresponding to our CPS transformation. In this transformation the third case for transforming let expressions is only taken if the first two cases do not apply.

Now we can prove the completeness of $\underline{\lambda}_{cps}$ reductions on CPS-transformed $\underline{\lambda}_c$ terms.

Lemma 13. *1. If* $\underline{\lambda}_{cps} \vdash W_1 \Rightarrow W_2$ *then* $\underline{\lambda}_c \vdash {W_1}^\natural \Rrightarrow {W_2}^\natural$.
2. If $\underline{\lambda}_{cps} \vdash M_1 \Rightarrow M_2$ *then* $\underline{\lambda}_c \vdash {M_1}^\sharp \Rrightarrow {M_2}^\sharp$.

Proof. Simultaneous induction.

1. case distinction over W_1
   (a) $W_1 \equiv x$: no reduction possible.
   (b) $W_1 \equiv \lambda x.\lambda k.M_1$ hence $W_2 \equiv \lambda x.\lambda k.M_2$ for some M_2 with $\underline{\lambda}_{cps} \vdash M_1 \Rightarrow M_2$. By induction, $\underline{\lambda}_c \vdash {M_1}^\sharp \Rrightarrow {M_2}^\sharp$.
   Now ${W_1}^\natural = (\lambda x.\lambda k.M_1)^\natural = \lambda x.{M_1}^\sharp \Rrightarrow \lambda x.{M_2}^\sharp = (\lambda x.\lambda k.M_2)^\natural = {W_2}^\natural$.
   (c) $W_1 \equiv \underline{\lambda}x.M_1$ and $\underline{\lambda}_{cps} \vdash M_1 \Rightarrow M_2$. By induction $\underline{\lambda}_c \vdash {M_1}^\sharp \Rrightarrow {M_2}^\sharp$.
   Therefore ${W_1}^\natural = \underline{\lambda}x.{M_1}^\sharp \Rrightarrow \underline{\lambda}x.{M_2}^\sharp = \underline{\lambda}x.{M_2}^\natural$.
2. case distinction over M_1
   (a) $M_1 \equiv K_1@W_1$ now ${M_1}^\sharp = {K_1}^\flat[{W_1}^\natural]$.

transforming M

$$
\begin{array}{ll}
(K@W)^\sharp & = K^\flat[W^\natural] \\
(W_1@W_2@K)^\sharp & = K^\flat[{W_1}^\natural@{W_2}^\natural] \\
(\text{let } y = W_1\underline{@}W_2 \text{ in } k@y)^\sharp & = {W_1}^\natural\underline{@}{W_2}^\natural \\
(\text{let } y = W_1\underline{@}W_2 \text{ in } (\lambda x.x)@y)^\sharp & = {W_1}^\natural\underline{@}{W_2}^\natural \\
(\text{let } y = W_1\underline{@}W_2 \text{ in } M)^\sharp & = \text{let } y = {W_1}^\natural\underline{@}{W_2}^\natural \text{ in } M^\sharp
\end{array}
$$

transforming values W

$$
\begin{array}{ll}
x^\natural & = x \\
(\lambda x.\lambda k.M)^\natural & = \lambda x.M^\sharp \\
(\underline{\lambda} x.M)^\natural & = \underline{\lambda} x.M^\sharp
\end{array}
$$

transforming contexts K

$$
\begin{array}{ll}
k^\flat[\,] & = [\,] \\
(\lambda x.x)^\flat[\,] & = [\,] \\
(\lambda x.M)^\flat[\,] & = \text{let } x = [\,] \text{ in } M^\sharp
\end{array}
$$

Figure6. Two-Level Direct Style Transformation

i. Suppose $W_1 \Rightarrow W_2$. By induction ${W_1}^\natural \Rrightarrow {W_2}^\natural$ therefore ${K_1}^\flat[{W_1}^\natural] \Rightarrow$ $\Rightarrow {K_1}^\flat[{W_2}^\natural] = (K_1@W_2)^\sharp$.

ii. Suppose $K_1 \equiv \lambda x.M$. Now $(\lambda x.M)@W_1 \Rightarrow M[x := W_1]$. Further ${M_1}^\sharp = (\lambda x.M)^\flat[{W_1}^\natural] = \text{let } x = {W_1}^\natural \text{ in } M^\sharp \Rightarrow M^\sharp[x := W^\natural] = (M[x := W])^\sharp$. [the last step needs a separate proof, which is straightforward, cf. [35, Lemma 7.4]]

iii. Suppose $K_1 \equiv \lambda x.x$. This is not a 2CPS redex.

iv. Suppose $K_1 \equiv \lambda x.M$ and $M \Rightarrow M'$. By induction.

(b) $M_1 \equiv W_1@W_2@K$

i. reductions inside W_1, W_2, or K by induction.

ii. Suppose $W_1 \equiv \lambda x.\lambda k.M$ then $\underline{\lambda}_{cps} \vdash M_1 \Rightarrow M[x := W_2][k := K]$. Further ${M_1}^\sharp = K^\flat[(\lambda x.\lambda k.M)^\natural@{W_2}^\natural] = K^\flat[(\lambda x.M^\sharp)@{W_2}^\natural] \Rightarrow K^\flat[(M[x := W_2])^\sharp]$.

A. $K \equiv k$: $k^\flat[(M[x := W_2])^\sharp] = (M[x := W_2])^\sharp = (M[x := W_2][k := k])^\sharp$.

B. $K \equiv \lambda y.M'$: now we need to prove by induction over M that $\text{let } y = (M[x := W_2])^\sharp \text{ in } M'^\sharp = (M[x := W_2][k := \lambda y.M'])^\sharp$. [left out]

C. $K \equiv \lambda y.y$: on the one side we have $(\lambda y.y)^\flat[(M[x := W_2])^\sharp] = M[x := W_2]^\sharp$; on the other side we obtain $(M[x := W_2][k := \lambda y.y])^\sharp = (M[x := W_2][k := k])^\sharp = (M[x := W_2])^\sharp$.

(c) $M_1 \equiv \text{let } y = W_1\underline{@}W_2 \text{ in } M$. Reductions inside of W_1 and W_2 are handled by induction. If $M \Rightarrow M'$ then neither $M \equiv k@y$ nor $M \equiv (\lambda x.x)@y$. The remaining case works by induction.

Corollary 14. $\underline{\lambda}_c \vdash E_1 = E_2$ *iff* $\underline{\lambda}_{cps} \vdash {E_1}^* = {E_2}^*$.

Proof. By lemmas 11 and 13.

7 A Reflection on Partial Evaluation

In this section we establish the result that our 2CPS transformation and the associated 2DS transformation form a reflection in $\underline{\lambda}_{cps}$ of $\underline{\lambda}_c$. Therefore, an implementation of a specializer using the 2CPS transformation yields exactly the same results as a $\underline{\lambda}_c$ reducer.

First we review some definitions about Galois connections and reflections [12] adapted to the current setting of reductions [35].

Definition 15. Two maps $^* : S \to T$ and $^\sharp : T \to S$ form a *Galois connection* from S to T whenever for all $E \in E$ and all $M \in T$

$$E \twoheadrightarrow_S M^\sharp \text{ if and only if } E^* \twoheadrightarrow_T M$$

There is a standard alternative characterization, which is more suited for our purposes.

Proposition 16. *Two maps* * *and* $^\sharp$ *form a* Galois connection *from* S *to* T *if and only if all of the following conditions hold.*

1. $E \twoheadrightarrow_S E^{*\sharp}$;
2. $M^{\sharp^*} \twoheadrightarrow_T M$;
3. $E \twoheadrightarrow_S E'$ *implies* $E^* \twoheadrightarrow_T E'^*$;
4. $M \twoheadrightarrow_T M'$ *implies* $M^\sharp \twoheadrightarrow_S M'^\sharp$.

Definition 17. A Galois connection of maps * and $^\sharp$ forms a *reflection in* S *of* T if $M^{\sharp^*} \equiv M$.

Lemma 18. $\underline{\lambda}_c \vdash E \twoheadrightarrow E^{*\sharp}$

Proof. As in [35], we need to prove the stronger result

$$K^\flat[E] \twoheadrightarrow (E : K)^\sharp$$

by induction on E. We also need to prove

$$V \twoheadrightarrow V^{\dagger\natural}$$

simultaneously. Further details of the proof are deferred to Appendix A.

To show that our two-level transformation forms a reflection, we first must refine the characterization of the image of the CPS transformation.

Lemma 19. *In Definition 8 we can replace* $\underline{\lambda} x.M$ *by*

$$\begin{array}{ll} W & ::= \ldots \mid \underline{\lambda} x.P \\ P & ::= (\lambda x.x)@W \mid W@W@\lambda x.x \mid \\ & \quad \text{let } y = W\underline{@}W \text{ in } P \mid (\lambda x.P)@W \mid W@W@\lambda x.P \end{array}$$

and restrict K *to*

$$K ::= k \mid \lambda x.M$$

Observation 9 still holds for this refined definition.

Terms in the syntactic category P are the image of CPS transformation with respect to the identity continuation. This fact is intuitively obvious since P only arises as body of a dynamic lambda abstraction $\underline{\lambda} x.P$.

Lemma 20. *Let* M *be a CPS term with subterm* P. *Let* $E_M = M^\sharp$ *under the 2DS transformation and* E_P *the subterm corresponding to* P. *Then: In the course of* ${E_M}^*$ *the subterm* E_P *is transformed using* $E_P : (\lambda x.x)$.

Proof. By context induction on the possible contexts of P.

Lemma 21. $M \equiv {M^\sharp}^*$.

Proof. Here we need the refined characterization of CPS terms of Lemma 19 and apply Lemma 20.

Hence, we simultaneously need to prove

1. $M \equiv {M^\sharp}^*$;
2. $W \equiv {W^\natural}^\dagger$;
3. $P \equiv P^\sharp : (\lambda x.x)$.

Further details of the proof are deferred to Appendix A.

Corollary 22. *The 2CPS transformation* $E \mapsto E^*$ *and the 2DS transformation* $M \mapsto M^\sharp$ *form a reflection in* $\underline{\lambda}_{cps}$ *of* $\underline{\lambda}_c$.

Proof. Immediate by definition of a reflection and lemmas 18 and 21.

Corollary 23. *The 2CPS transformation* $E \mapsto E^*$ *and the 2DS transformation* $M \mapsto M^\sharp$ *form a Galois connection from* $\underline{\lambda}_c$ *to* $\underline{\lambda}_{cps}$.

In earlier work, Sabry and Felleisen have established the notion of an equational correspondence as a means to express properties of a translation between equational theories.

Definition 24 ([34]). Two maps * and $^\sharp$ form an *equational correspondence* from S to T if and only if all of the following conditions hold.

1. $S \vdash E = E^{*\sharp}$;
2. $T \vdash {M^\sharp}^* = M$;
3. $S \vdash E = E'$ implies $T \vdash E^* = {E'}^*$;
4. $T \vdash M = M'$ implies $S \vdash M^\sharp = {M'}^\sharp$.

Corollary 25. *The 2CPS transformation* $E \mapsto E^*$ *and the 2DS transformation* $M \mapsto M^\sharp$ *form an equational correspondence from* $\underline{\lambda}_c$ *to* $\underline{\lambda}_{cps}$.

$$
\begin{aligned}
\mathcal{S}[\![x]\!]\rho &= \lambda k.k(\rho x)\\
\mathcal{S}[\![\lambda x.E]\!]\rho &= \lambda k.k(\lambda y.\mathcal{S}[\![E]\!]\rho[x := y])\\
\mathcal{S}[\![E_1 @ E_2]\!]\rho &= \lambda k.\mathcal{S}[\![E_1]\!]\rho\lambda y_1.\mathcal{S}[\![E_2]\!]\rho\lambda y_2.y_1 y_2 k\\
\mathcal{S}[\![\underline{\lambda} x.E]\!]\rho &= \lambda k.k(\underline{\lambda} y.\mathcal{S}[\![E]\!]\rho[x := y]\lambda z.z)\\
\mathcal{S}[\![E_1 \underline{@} E_2]\!]\rho &= \lambda k.\mathcal{S}[\![E_1]\!]\rho\lambda y_1.\mathcal{S}[\![E_2]\!]\rho\lambda y_2.\underline{\mathrm{let}}\ z = y_1 \underline{@} y_2\ \underline{\mathrm{in}}\ kz
\end{aligned}
$$

Figure7. Specializer Based on $\underline{\lambda}_c$

8 A Practical Sound Specializer

To obtain a specializer that properly implements $\underline{\lambda}_c$ we compose a vanilla specializer for the lambda calculus (e.g., [20]) with a two-level CPS transformation in the spirit of Fig. 5. The resulting specializer is shown in Fig. 7. This specializer constructs residual programs in A-normal form, which is especially suited for compilation. By construction, the specializer is correct for arbitrary monadic effects, e.g., state, exceptions, control, nondeterminism.

9 Related Work

λ_c is one of the contributions of Moggi's seminal work on monads [27, 28]. We have extended that calculus to a two-level calculus $\underline{\lambda}_c$ to obtain a calculus for sound call-by-value specialization that applies to all monadic models.

Sabry and Wadler [35] introduced the sharpened notions of Galois connection and reflection that we also use in this paper. We have extended their results for the computational lambda calculus to the two-level computational lambda calculus. In addition, they investigated a factorization of the transformation through Moggi's computational meta-language.

Barthe, Hatcliff, and Sørensen have generalized Galois connections and equational correspondences to a notion called $(\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4)$-*correspondence* [2]. Briefly, this notion abstracts from the four relations in Proposition 16 and Definition 24 so that Galois connections, reflections, and equational correspondences are all particular instances of $(\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4)$-correspondences. They have shown that there is no reflection in $\lambda\beta_V$ of λ_c, but that there is a reduction correspondence, where $\mathcal{R}_i$ is $\twoheadrightarrow\!\!\!\!\rightarrow$. Therefore we cannot expect that there is a reflection in $\underline{\lambda}\beta_V$ of $\underline{\lambda}_c$. However, it might be possible to generalize their result to our case, but this has not yet been investigated.

Hatcliff and Danvy [22] have previously investigated the connection between the computational meta-language and the CPS transformation. This resulted in a generic framework for CPS transformations. In a recent paper [23] the same authors specify and prove correct partial evaluation using the computational meta-language. To this end they also set up a two-level version of the meta-language. Their specification corresponds roughly to the **SD** CPS transformation described in Section 5.3. Therefore, our calculus $\underline{\lambda}_c$ is *not* isomorphic to a sublanguage of their two-level meta-language. Our results also support their thesis that Moggi's theory forms a sound basis for specialization. Their results have been achieved earlier but independently from our results. Another major difference is their reliance on a typed language whereas our framework is untyped. Based on this type information they define a well-formedness criterion that can be used as a specification for a binding-time analysis and prove the correctness of the binding-time analysis.

The calculus presented in this paper has been developed in the course of work on offline partial evaluation in the presence of state [16]. It forms the basis for a practical implementation of a partial evaluator for Scheme [8].

Continuation-based specialization can be an implementation for $\underline{\lambda}_c$. However, many specifications allow for additional reductions that are unsound in general. This topic is extensively discussed in Section 5.2.

Davies and Pfenning [13, 14] have shown that certain modal and temporal logics can serve as specifications for binding-time properties. These logical specifications give rise to binding-time analyses that are correct by definition for the typed lambda calculi derived from the above logics. The main contribution of their work is to show that these calculi describe staged computation in a natural way that is motivated by logics. They also identify a difference in the treatment of free variables in the Nielsons' two-level lambda calculus [31] and the variant generally used to express the annotations of partial evaluation [20]. In contrast, the focus of the present work is the sound specialization of programs with computational effects and not their proper staging. In fact, our technical results also apply to terms which are not properly staged.

Moggi [29] describes a categorical semantics of two-level languages based on Davies's framework. Again, the main emphasis of this work is the proper description of staging, which Moggi relates to the phase distinction problem in module languages. Although his semantics is geared towards pure two-level languages, it should be possible to extend it to encompass computational effects.

10 Conclusions

The calculus $\underline{\lambda}_c$ presented in this paper provides a sound basis for specialization of languages with computational effects. We have shown that it can be implemented using a two-level CPS transformation. The quality of this implementation derives from the fact that our transformation establishes a reflection in $\underline{\lambda}_{cps}$ of $\underline{\lambda}_c$.

Our calculus does not address the issue of staging programs with effects, or how to perform effects at specialization time. Instead it is targeted at the approach taken by Bondorf and Danvy, in which all operations with I/O effects or side effects are residualized [4]. Our approach provides sound specialization in this context.

It is possible to integrate the CPS transformation into the specializer, by symbolic composition, and thereby to avoid an additional transformation step (analogously to Bondorf's work [3]). The resulting specializer is written in non-standard CPS, which can be expressed in direct style with control operators, as was shown for the **FD0** specializer [25].

Likewise, it is straightforward to extend the calculus to more than two stages of computation and derive sound and powerful specializers for these applications [19,37].

Acknowledgement

Thanks to Dirk Dussart for stimulating discussions. Special thanks to John Hatcliff and Olivier Danvy for discussions, careful reading, and detailed comments on an earlier version of this paper.

References

1. H. P. Barendregt. *The Lambda Calculus — Its Syntax and Semantics.* North-Holland, 1984.
2. Gilles Barthe, John Hatcliff, and Morten Heine Sørensen. Reflections on reflections. In Hugh Glaser and Herbert Kuchen, editors, *Programming Languages, Implementations, Logics, and Programs (PLILP '97)*, volume ? of *Lecture Notes in Computer Science*, page To appear, Southampton, England, September 1996. Springer-Verlag.
3. Anders Bondorf. Improving binding-times without explicit CPS conversion. In *Proc. 1992 ACM Conference on Lisp and Functional Programming*, pages 1–10, San Francisco, California, USA, June 1992.
4. Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Programming*, 16(2):151–195, 1991.
5. Anders Bondorf and Dirk Dussart. Improving CPS-based partial evaluation: Writing cogen by hand. In Peter Sestoft and Harald Søndergaard, editors, *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '94*, pages 1–10, Orlando, Fla., June 1994. ACM.
6. Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation: Extended version. Technical Report 93/4, DIKU, University of Copenhagen, Denmark, 1993. extended version of [7].
7. Anders Bondorf and Jesper Jørgensen. Efficient analysis for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.
8. William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language scheme. AI Memo 848b, MIT, 1991. ftp://swiss-ftp.ai.mit.edu/pub/scm/r4rs.texi.
9. Charles Consel and Olivier Danvy. For a better support of static data flow. In John Hughes, editor, *Proc. Functional Programming Languages and Computer Architecture 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 496–519, Cambridge, MA, 1991. Springer-Verlag.
10. Olivier Danvy. Back to direct style. *Science of Programming*, 22:183–195, 1994.
11. Olivier Danvy and Dirk Dussart. CPS transformation after binding-time analysis. Unpublished note, Computer Science Department, Aarhus University, April 1995.

12. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.
13. Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.
14. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 258–270, St. Petersburg, Fla., January 1996. ACM Press.
15. Dirk Dussart and Eddy Bevers. CPS transformation after binding-time analysis (extended abstract). In *Proceedings of the 7th Nordic Workshop on Programming Theory*, pages 112–126, Göteborg, Sweden, November 1995.
16. Dirk Dussart and Peter Thiemann. Partial evaluation for higher-order languages with state. Berichte des Wilhelm-Schickard-Instituts WSI-97-XX, Universität Tübingen, April 1997.
17. R. Kent Dybvig, editor. *International Conference on Functional Programming*, Philadelphia, PA, May 1996. ACM Press, New York.
18. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proc. of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, Albuquerque, New Mexico, June 1993.
19. Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In Swierstra and Hermenegildo [36], pages 259–278.
20. Carsten K. Gomard. A self-applicable partial evaluator for the lambda-calculus. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, 1992.
21. John Hatcliff. Mechanically verifying the correctness of an offline partial evaluator. In Swierstra and Hermenegildo [36], pages 279–298.
22. John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, pages 458–471, Portland, OG, January 1994. ACM Press.
23. John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 1997. To appear.
24. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
25. Julia Lawall and Olivier Danvy. Continuation-based partial evaluation. In *Proc. 1994 ACM Conference on Lisp and Functional Programming*, pages 227–238, Orlando, Florida, USA, June 1994. ACM Press.
26. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.
27. Eugenio Moggi. Computational lambda-calculus and monads. Technical Report ECS-LFCS-88-86, University of Edinburgh, 1988.
28. Eugenio Moggi. Computational lambda-calculus and monads. In *Proc. of the 4rd Annual Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, CA, June 1989. IEEE Computer Society Press.
29. Eugenio Moggi. A categorical account of two-level languages. In *Proc. Mathematical Foundations of Programming Semantics, Thirteenth Annual Conference*, volume 5 of *Electronic Notes in Theoretical Computer Science*, Pittsburgh, PA, March 1997. Carnegie Mellon University, Elsevier Science BV.
30. Flemming Nielson. A formal type system for comparing partial evaluators. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 349–384, Amsterdam, 1988. North-Holland.

31. Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages.* Cambridge University Press, 1992.
32. Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):347–364, July 1993.
33. Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
34. Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.
35. Amr Sabry and Philip Wadler. A reflection on call-by-value. In Dybvig [17], pages 13–24.
36. Doaitse Swierstra and Manuel Hermenegildo, editors. *Programming Languages, Implementations, Logics, and Programs (PLILP '95)*, volume 982 of *Lecture Notes in Computer Science*, Utrecht, The Netherlands, September 1995. Springer-Verlag.
37. Peter Thiemann. Cogen in six lines. In Dybvig [17], pages 180–189.
38. Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):365–387, July 1993.

A Proofs of Lemmas 18 and 21

Lemma 18 $\underline{\lambda}_c \vdash E \twoheadrightarrow E^{*\sharp}$.

Proof. As in [35], we need to prove the stronger result

$$K^\flat[E] \twoheadrightarrow (E : K)^\sharp$$

by induction on E. We also need to prove

$$V \twoheadrightarrow V^{\dagger\natural}$$

simultaneously.

- case $E \equiv x$: $(x : K)^\sharp = K@x^\sharp = K^\flat[x]$.
- see [35] for cases $\lambda x.E$, $E@E$, and let $x = E$ in E.
- case $\underline{\lambda} x.E$:

$$\begin{array}{rl} & ((\underline{\lambda} x.E) : K)^\sharp \\ = & K@\underline{\lambda} x.(E : (\lambda y.y))^\sharp \\ = & K^\flat[\underline{\lambda} x.(E : (\lambda y.y))^\sharp] \\ \overset{ind}{\twoheadleftarrow} & K^\flat[\underline{\lambda} x.(\lambda y.y)^\flat[E]] \\ = & K^\flat[\underline{\lambda} x.E] \end{array}$$

- case $E_1\underline{@}E_2$: we need three subcases:
  - case $N\underline{@}E$:

$$\begin{array}{rl} & ((N\underline{@}E) : K)^\sharp \\ = & (N : (\lambda x.(x\underline{@}E) : K))^\sharp \\ \overset{ind}{\twoheadleftarrow} & (\lambda x.(x\underline{@}E) : K)^\flat[N] \\ = & \text{let } x = N \text{ in } ((x\underline{@}E) : K)^\sharp \\ \overset{ind}{\twoheadleftarrow} & \text{let } x = N \text{ in } K^\flat[x\underline{@}E] \end{array}$$

    We proceed by case analysis on K:

1. case $K \equiv k$:

$$\begin{array}{cl} = & \text{let } x = N \text{ in } k^\flat[x\underline{@}E] \\ = & \text{let } x = N \text{ in } x\underline{@}E \\ \Lleftarrow & N\underline{@}E \\ = & k^\flat[N\underline{@}E] \end{array}$$

2. case $K \equiv \lambda x.x$: similar
3. case $K \equiv \lambda y.M$:

$$\begin{array}{cl} & \text{let } x = N \text{ in } (\lambda y.M)^\flat[x\underline{@}E] \\ = & \text{let } x = N \text{ in let } y = x\underline{@}E \text{ in } M^\sharp \\ \Lleftarrow & \text{let } y = \text{let } x = N \text{ in } x\underline{@}E \text{ in } M^\sharp \\ \Lleftarrow & \text{let } y = N\underline{@}E \text{ in } M^\sharp \\ = & (\lambda y.M)^\flat[N\underline{@}E] \end{array}$$

- case $V\underline{@}N$: similar
- case $V_1\underline{@}V_2$:

$$\begin{array}{cl} & ((V_1\underline{@}V_2) : K)^\sharp \\ = & (\text{let } y = {V_1}^\dagger\underline{@}{V_2}^\dagger \text{ in } K \downarrow y)^\sharp \end{array}$$

Now we need cases on K:

∗ case $K \equiv k$:

$$\begin{array}{cl} = & (\text{let } y = {V_1}^\dagger\underline{@}{V_2}^\dagger \text{ in } k \downarrow y)^\sharp \\ = & (\text{let } y = {V_1}^\dagger\underline{@}{V_2}^\dagger \text{ in } k@y)^\sharp \\ = & {V_1}^{\dagger\natural}\underline{@}{V_2}^{\dagger\natural} \\ \overset{ind}{\Lleftarrow} & V_1\underline{@}V_2 \\ = & k^\flat[V_1\underline{@}V_2] \end{array}$$

∗ case $K \equiv \lambda x.x$: Similar

∗ case $K \equiv \lambda x.M$:

$$\begin{array}{cl} = & (\text{let } y = {V_1}^\dagger\underline{@}{V_2}^\dagger \text{ in } (\lambda x.M) \downarrow y)^\sharp \\ = & (\text{let } y = {V_1}^\dagger\underline{@}{V_2}^\dagger \text{ in } M[x := y])^\sharp \\ = & \text{let } y = {V_1}^{\dagger\natural}\underline{@}{V_2}^{\dagger\natural} \text{ in } (M[x := y])^\sharp \\ \overset{ind}{\Lleftarrow} & \text{let } y = V_1\underline{@}V_2 \text{ in } (M[x := y])^\sharp \\ \equiv & \text{let } x = V_1\underline{@}V_2 \text{ in } M^\sharp \\ = & (\lambda x.M)^\flat[V_1\underline{@}V_2] \end{array}$$

The proof that $V \Rrightarrow V^{\dagger\natural}$ is as follows:

– $V \equiv x$: $x^{\dagger\natural} = x$.
– $V \equiv \lambda x.E$:

$$\begin{array}{cl} & (\lambda x.E)^{\dagger\natural} \\ = & (\lambda x.\lambda k.E : k)^\natural \\ = & \lambda x.(E : k)^\sharp \\ \overset{ind}{\Lleftarrow} & \lambda x.k^\flat[E] \\ & \lambda x.E \end{array}$$

– $V \equiv \underline{\lambda} x.E$:

$$\begin{array}{rl} & (\underline{\lambda} x.E)^{\dagger\natural} \\ = & (\underline{\lambda} x.(E : \lambda y.y))^{\natural} \\ = & \underline{\lambda} x.(E : \lambda y.y)^{\sharp} \\ \stackrel{ind}{\Leftarrow} & \underline{\lambda} x.(\lambda y.y)^{\flat}[E] \\ = & \underline{\lambda} x.E \end{array}$$

Lemma 21 $M \equiv M^{\sharp *}$.

Proof. Here we need the refined characterization of CPS terms of Lemma 19 and apply Lemma 20.

Hence, we simultaneously need to prove

1. $M \equiv M^{\sharp *}$;
2. $W \equiv W^{\natural \dagger}$;
3. $P \equiv P^{\sharp} : (\lambda x.x)$.

1. Case distinction on M.
   (a) case $M \equiv K@W$:

$$\begin{array}{rl} & (K@W)^{\sharp *} \\ = & K^{\flat}[W^{\natural}]^{*} \end{array}$$

   cases over K:
   i. $K \equiv k$:

$$\begin{array}{rl} = & k^{\flat}[W^{\natural}]^{*} \\ = & W^{\natural *} \\ = & k@W^{\natural\dagger} \\ \stackrel{ind}{\equiv} & k@W \end{array}$$

   ii. $K \equiv \lambda x.M$:

$$\begin{array}{rl} = & (\lambda x.M)^{\flat}[W^{\natural}]^{*} \\ = & \text{let } x = W^{\natural} \text{ in } M^{\sharp *} \\ = & W^{\natural} : (\lambda x.M^{\sharp *}) \\ \stackrel{ind}{\equiv} & W^{\natural} : (\lambda x.M) \\ = & (\lambda x.M)@W^{\natural\dagger} \\ \equiv & (\lambda x.M)@W \end{array}$$

   (b) case $M \equiv W_1@W_2@K$:

$$\begin{array}{rl} & (W_1@W_2@K)^{\sharp *} \\ = & K^{\flat}[W_1{}^{\natural}@W_2{}^{\natural}]^{*} \end{array}$$

   cases over K

i. case $K \equiv k$:

$$\begin{aligned} &= k^\flat[{W_1}^\natural @ {W_2}^\natural]^* \\ &= ({W_1}^\natural @ {W_2}^\natural)^* \\ &= {{W_1}^\natural}^\dagger @ {{W_2}^\natural}^\dagger @ k \\ &\overset{ind}{\equiv} W_1 @ W_2 @ k \end{aligned}$$

ii. case $K \equiv \lambda x.M$:

$$\begin{aligned} &= (\lambda x.M)^\flat[{W_1}^\natural @ {W_2}^\natural]^* \\ &= \text{let } x = {W_1}^\natural @ {W_2}^\natural \text{ in } {M^\sharp}^* \\ &= ({W_1}^\natural @ {W_2}^\natural) : (\lambda x.{M^\sharp}^*) \\ &= {{W_1}^\natural}^\dagger @ {{W_2}^\natural}^\dagger @ (\lambda x.{M^\sharp}^*) \\ &\overset{ind}{\equiv} W_1 @ W_2 @ (\lambda x.{M^\sharp}^*) \\ &\overset{ind}{\equiv} W_1 @ W_2 @ (\lambda x.M) \end{aligned}$$

(c) case $M \equiv \text{let } y = W_1 \underline{@} W_2 \text{ in } M_1$: case analysis on M_1:

i. case $M_1 \equiv k@y$:

$$\begin{aligned} & (\text{let } y = W_1 \underline{@} W_2 \text{ in } k@y)^\sharp : k \\ = \; & ({W_1}^\natural \underline{@} {W_2}^\natural) : k \\ = \; & \text{let } y = {{W_1}^\natural}^\dagger \underline{@} {{W_2}^\natural}^\dagger \text{ in } k \downarrow y \\ \overset{ind}{\equiv} \; & \text{let } y = W_1 \underline{@} W_2 \text{ in } k@y \end{aligned}$$

ii. case $M_1 \not\equiv k@y$:

$$\begin{aligned} & (\text{let } y = W_1 \underline{@} W_2 \text{ in } M_1)^\sharp : k \\ = \; & ({W_1}^\natural \underline{@} {W_2}^\natural) : (\lambda y.({M_1}^\sharp : k)) \\ = \; & \text{let } y = {{W_1}^\natural}^\dagger \underline{@} {{W_2}^\natural}^\dagger \text{ in } ({M_1}^\sharp : k) \downarrow y \\ \overset{ind}{\equiv} \; & \text{let } y = W_1 \underline{@} W_2 \text{ in } M^\sharp[y := y] : k \\ \overset{ind}{\equiv} \; & \text{let } y = W_1 \underline{@} W_2 \text{ in } M \end{aligned}$$

2. The next statement to prove is $W \equiv {W^\natural}^\dagger$ by case analysis on W.
   (a) case $W \equiv x$: obvious.
   (b) case $W \equiv \lambda x.\lambda k.M$: by induction, since $(\lambda x.\lambda k.M)^{\natural\dagger} = (\lambda x.M^\sharp)^\dagger = \lambda x.\lambda k.(M^\sharp : k)$.
   (c) case $W \equiv \underline{\lambda} x.P$: by induction, since $(\underline{\lambda} x.P)^{\natural\dagger} = (\underline{\lambda} x.P^\sharp)^\dagger = \underline{\lambda} x.(P^\sharp : (\lambda y.y))$.
3. Finally, we prove $P \equiv P^\sharp : (\lambda x.x)$ by case analysis on P.
   (a) case $P \equiv (\lambda x.x)@W$:

$$\begin{aligned} & ((\lambda x.x)@W)^\sharp : (\lambda x.x) \\ = \; & (\lambda x.x)^\flat[W^\natural] : (\lambda x.x) \\ = \; & W^\natural : (\lambda x.x) \\ = \; & (\lambda x.x)@{W^\natural}^\dagger \\ \overset{ind}{\equiv} \; & (\lambda x.x)@W \end{aligned}$$

(b) case $P \equiv W_1@W_2@\lambda x.x$:

$$\begin{array}{ll} & (W_1@W_2@(\lambda x.x))^\sharp : (\lambda x.x) \\ = & (\lambda x.x)^\flat[W_1{}^\natural @W_2{}^\natural] : (\lambda x.x) \\ = & W_1{}^\natural @W_2{}^\natural : (\lambda x.x) \\ = & W_1{}^{\natural\dagger}@W_2{}^{\natural\dagger}@(\lambda x.x) \\ \stackrel{ind}{\equiv} & W_1@W_2@(\lambda x.x) \end{array}$$

(c) case $P \equiv \text{let } y = W_1\underline{@}W_2 \text{ in } (\lambda x.x)@y$:

$$\begin{array}{ll} & (\text{let } y = W_1\underline{@}W_2 \text{ in } (\lambda x.x)@y)^\sharp : (\lambda x.x) \\ = & (W_1{}^\natural \underline{@}W_2{}^\natural) : (\lambda x.x) \\ = & \text{let } y = W_1{}^{\natural\dagger}\underline{@}W_2{}^{\natural\dagger} \text{ in } (\lambda x.x) \downarrow y \\ \stackrel{ind}{\equiv} & \text{let } y = W_1\underline{@}W_2 \text{ in } (\lambda x.x)@y \end{array}$$

(d) case $P \equiv \text{let } y = W_1\underline{@}W_2 \text{ in } P$ and $P \not\equiv (\lambda x.x)@y$:

$$\begin{array}{ll} & (\text{let } y = W_1\underline{@}W_2 \text{ in } P)^\sharp : (\lambda x.x) \\ = & (\text{let } y = W_1{}^\natural\underline{@}W_2{}^\natural \text{ in } P^\sharp) : (\lambda x.x) \\ = & (W_1{}^\natural\underline{@}W_2{}^\natural) : (\lambda y.P^\sharp : (\lambda x.x)) \\ = & \text{let } y = W_1{}^{\natural\dagger}\underline{@}W_2{}^{\natural\dagger} \text{ in } (\lambda y.P^\sharp : (\lambda x.x)) \downarrow y \\ \stackrel{ind}{\equiv} & \text{let } y = W_1\underline{@}W_2 \text{ in } P^\sharp : (\lambda x.x) \\ \stackrel{ind}{\equiv} & \text{let } y = W_1\underline{@}W_2 \text{ in } P \end{array}$$

(e) case $P \equiv (\lambda x.P)@W$: analogous to case $M \equiv (\lambda x.M)@W$.
(f) case $P \equiv W@W@\lambda x.P$: analogous to case $M \equiv W@W@\lambda x.M$.