Offline Partial Evaluation for Shift and Reset

Kenichi Asai[*]
College of Computer and Information Science, Northeastern University
360 Huntington Avenue, Boston, MA 02115
asai@ccs.neu.edu

ABSTRACT

This paper presents an offline partial evaluator for the λ-calculus with the delimited continuation constructs *shift* and *reset*. Based on Danvy and Filinski's type system for shift and reset, we first present a type system that specifies well-annotated terms. We then show a specializer that receives an annotated term and produces the output in continuation-passing style (CPS). The correctness of our partial evaluator is established using the technique of logical relations. Thanks to the explicit reference to the type of continuations, we can establish the correctness using the standard proof technique of structural induction, despite the fact that the specializer itself is written in CPS. The paper also shows an efficient constraint-based binding-time analysis as well as how to extend the present work to richer language constructs, such as recursion and conditionals.

Categories and Subject Descriptors

D.1.1 [**Software**]: Programming Techniques—*applicative (functional) languages*; D.3.2 [**Programming Languages**]: Language Classifications—*applicative (functional) languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*control structures*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*partial evaluation*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*lambda calculus and related systems*

General Terms

Languages, Theory

Keywords

Delimited continuations, offline partial evaluation, binding-time analysis (BTA), continuation-passing style (CPS), CPS transformation, logical relations

1. INTRODUCTION

A delimited continuation, proposed by Danvy and Filinski [5], is an abstraction representing a *part* of program execution as an ordinary function. It differs from the standard continuation captured by, *e.g.*, call-with-current-continuation (or call/cc for short) found in Scheme [12] in that it captures only the continuation up to an enclosing control delimiter. Because we can manipulate (parts of) continuations without transforming the whole program into a continuation-passing style (CPS), we obtain both the expressive power of non-tail calls in CPS programs and the simplicity of direct style programs. Delimited continuations have been used in various places, including backtracking and non-deterministic programming [5], translations to A-normal forms [8], let-insertion in partial evaluation [14, 16], and an implementation of reflective languages in direct style [2].

As the use of delimited continuations becomes popular, it becomes more important for program transformations to be able to handle them properly, effectively, and efficiently. In this paper, we consider one program transformation, namely, partial evaluation [11]. In previous work [1], we presented an online partial evaluator for the λ-calculus extended with the delimited continuation constructs *shift* and *reset*. As an online partial evaluator, it checks at specialization time a binding time of each subexpression. If it is known (or *static*), it is reduced; if unknown (or *dynamic*), residualized. However, since it requires checking of binding times every time an expression is processed, the specialization is inefficient. Furthermore, its correctness was left open [1].

To overcome these problems, we investigate its *offline* variant. Offline partial evaluators perform a binding-time analysis (BTA) prior to specialization to annotate each subterm as either static or dynamic. At specialization time, only those subterms that are classified as static are reduced. Since the binding-time analysis is done once and for all before specialization, the overhead of checking binding times at specialization time is removed and specialization is expected to be more efficient than when using an online specializer. Offline partial evaluation has another benefit in that it has an established method for proving its correctness [17].

In this paper, we first review the type system for shift and reset presented by Danvy and Filinski [4]. The binding-time rules for well-annotatedness are formalized based on this type system. We then present a specializer that receives an annotated term and produces a specialized term in CPS. It is essentially the mixture of a CPS interpreter and a CPS transformer.

Since the result of partial evaluation is in CPS, we could

---

[*] On leave from Ochanomizu University. 2-1-1 Otsuka Bunkyo-ku Tokyo 112-8610 Japan. `asai@is.ocha.ac.jp`

PEPM'04, August 24–26, 2004, Verona, Italy.

obtain the same result by first transforming the input into CPS and then performing the standard partial evaluation that does not handle shift and reset. We did not take this approach, because obtaining the specialized result in CPS is not our ultimate goal. Rather, we want to establish a reasoning method for programs written in CPS, and even for programs written with shift and reset. Toward this goal, the present paper is a first step and shows a correctness proof for a specializer written in CPS.

The correctness of our partial evaluator relative to well-annotatedness is established using the technique of logical relations, following Wand [17]. The two main obstacles for the correctness proof are the presence of shift and reset in the source language and the fact that the specializer is written in CPS. It turns out that the former is not a serious problem, since the output of the specializer is in CPS and thus the standard logical relations for the λ-calculus can be used. We only need to establish the correspondence between the types in the source language and the types in the output language. The latter problem, *i.e.*, the fact that the specializer is itself written in CPS, complicates the proof of correctness. The standard structural induction does not seem to be applicable at first sight, because even if a term has some type τ, the final result of the computation may have a different type, so we cannot use the induction hypothesis. The key point to make structural induction work is to use a type system that takes continuations into account. With a proper assumption on the type of continuations, the proof of correctness goes through, naturally following the CPS structure of the specializer.

Although the type system for well-annotatedness specifies whether already annotated terms are well-formed or not, it does not specify how to obtain well-annotated terms from unannotated ones. As usual, we reformulate the type system as syntax-directed rules to obtain a constraint-based binding-time analysis. We also demonstrate how to extend our work to cope with richer language constructs, such as recursion and conditionals.

The paper is organized as follows. After briefly introducing shift and reset with examples in Section 2 and our language in Section 3, we show a CPS interpreter (Section 4) and a CPS transformer (Section 5). We then review the type system for shift and reset in Section 6. In Section 7, we extend the type system to the binding-time specification defining well-annotatedness. The specializer is shown in Section 8, and its correctness is proved in Section 9. After showing the binding-time analysis in Section 10, we extend our language to richer constructs in Section 11. Section 12 illustrates examples of specialization. Related work is in Section 13 and the paper concludes in Section 14. Appendix A contains the complete proof of correctness.

2. SHIFT AND RESET

To manipulate delimited continuations, Danvy and Filinski [5] introduced two constructs: *shift* and *reset*. Let us write shift as $\xi k.\, M$ and reset as $\langle M\rangle$ where M is an arbitrary term. Intuitively, shift takes its current continuation up to its enclosing reset, binds it to k, and discards the current continuation with which the whole shift expression was called. For example, in the expression $1+\langle 10+\xi k.\, k\,(k\,100)\rangle$, k is bound to a continuation $\lambda v.\, 10+v$. Applying it twice to 100 yields 120, the continuation $\langle 10+\cdot\rangle$ is discarded, the value of $\langle 10+\xi k.\, k\,(k\,100)\rangle$ is thus 120, and the final result is 121. The precise semantics of shift and reset is typically given by a CPS interpreter (see Section 4) or by CPS transformation (see Section 5).

Below, we show examples of the use of shift and reset. Scheme is used for the presentation of examples. In Scheme, an implementation of shift and reset using `call/cc` is available [13]. There is also a direct and more efficient implementation of shift and reset [9].

2.1 Example: Append

We first show a simple program presented by Danvy and Filinski [4]. The following program is a curried version of append. It receives a list `x` and returns a function, which, when applied to a list `y`, appends a list `x` to `y`.

```
(define (append x)
  (letrec ((app (lambda (x)
                  (if (null? x)
                    (shift k k)
                    (cons (car x)
                          (app (cdr x)))))))
    (reset (app x))))
```

The behavior of this program is understood as follows. When called with an argument `x`, `append` calls an auxiliary function `app` in an empty context. The function `app` then traverses over the list `x`, leaving in the continuation each element of the list to be consed later. When `app` finally reaches the empty list, *it captures the current continuation and returns it directly.* The captured continuation is exactly the function we want: when applied, it conses all the elements of `x` one by one in the reverse order.

```
> (define f (append '(1 2 3)))
> (f '(10 11 12))
(1 2 3 10 11 12)
```

2.2 Example: Reverse

We next show a reverse program, also due to Danvy and Filinski [4]. Although simple, it demonstrates a typical use of shift and reset.

```
(define (reverse lst)
  (letrec ((rev (lambda (lst)
                  (if (null? lst)
                    '()
                    (shift k ; else
                      (cons (car lst)
                        (k (rev (cdr lst)))))))))
    (reset (rev lst))))

> (reverse '(1 2 3))
(3 2 1)
```

To understand the behavior of this program, we first notice that `(shift k (k M))` is equal to `M` if `k` does not occur free in `M`. Capturing the current continuation in `k` and simply passing it a result of `M` without making any other use of `k` is the same as simply evaluating `M`. Thus, if the else branch of the if expression in `rev` was

```
(shift k (k (cons (car lst) (rev (cdr lst)))))
```

which is equal to `(cons (car lst) (rev (cdr lst)))`, the reverse function would become an identity function. Having

```
(shift k (cons (car lst) (k (rev (cdr lst)))))
```

for the else branch means that we *reverse* the order of “the rest of the computation `k`” and consing `(car lst)`. Suppose that we are reversing a list `’(1 2 3)` and we have just reached to process the last element `3`. The recursive call `(rev (cdr lst))` would thus return an empty list. At this time, “the rest of the computation” is “to construct a reversed list for the elements seen so far (namely, `1` and `2`) starting from an empty list.” After we obtain the reversed list `(2 1)` as the result of the rest of the computation, we cons `3` on top of it to obtain `(3 2 1)`.

The following execution trace demonstrates how the reordering actually happens. In the trace, captured continuations are underlined (except for the first shift where the captured continuation is an identity function and hence not shown).

```
> (reverse ’(1 2 3))
=> (reset (rev ’(1 2 3)))
=> (reset (shift k (cons 1 (k (rev ’(2 3))))))
=> (reset (cons 1 (rev ’(2 3))))
=> (reset (cons 1
     (shift k (cons 2 (k (rev ’(3)))))))
=> (reset (cons 2 (cons 1 (rev ’(3)))))
=> (reset (cons 2 (cons 1
     (shift k (cons 3 (k (rev ’()))))))))
=> (reset (cons 3 (cons 2 (cons 1 (rev ’())))))
(3 2 1)
```

It is instructive to consider the CPS transformed version of the same reverse program. There is a one-to-one correspondence between the two programs. Readers are invited to write down the execution trace for this program.

```
(define (reverse-cps lst)
  (letrec ((rev-cps (lambda (lst cont)
             (if (null? lst)
               (cont ’())
               (rev-cps (cdr lst) ; else
                 (lambda (x)
                   (cons (car lst) (cont x))))))))
    (reset (rev-cps lst (lambda (x) x)))))

> (reverse-cps ’(1 2 3))
(3 2 1)
```

In the ordinary CPS programs, all the serious calls are tail calls. But here in the else branch, `cont` is used in a nonstandard way. By making use of non-tail calls in CPS programs, we obtain more expressive power over control of execution. With shift and reset, we obtain it without transforming the whole program into CPS.

2.3 Example: A-Normal Form Translation

In this section, we show how shift and reset can be used to implement a translation of programs into A-normal forms [8]. The A-normal form is an intermediate language used in a compiler that assigns a unique name to each subexpression. Given a possibly nested expression, the transformer produces a sequence of let bindings and a program where no nested function calls are used. For example, a nested expression `(h (g (f x)) (f y))` is transformed into the following A-normal form:

```
(let* ((p (f x)) (q (g p)) (r (f y)))
  (h q r))
```

An interesting observation on this translation is that the produced program does not follow the structure of the input program. The innermost expression `(f x)` in the input program appears in the outermost position in the transformed program. This makes it hard to implement the translation. One usually requires to maintain a list of let bindings as an additional parameter.

Notice that the situation here is close to the reverse program. In the reverse program, the last element required to appear at the beginning. Here, the subexpression at the leaf of the syntax tree needs to appear at the front of the let bindings. Thus, all we need to do is to *reorder* the translation process and the construction of let bindings. Using shift and reset, we can implement it naturally in direct style as follows:

```
(define (a exp)
  (cond ((symbol? exp) exp) ; variable
        ((eq? (car exp) ’lambda)
         ; (lambda (var) body)
         (let ((var  (car (car (cdr exp))))
               (body (car (cdr (cdr exp)))))
           ‘(lambda (,var) ,(reset (a body)))))
        (else ; application (m n)
          (shift k
            (let ((m (gensym ’m)) (n (gensym ’n)))
              ‘(let* ((,m ,(a (car exp)))
                      (,n ,(a (car (cdr exp)))))
                 ,(k ‘(,m ,n)))))))))
(define (a-normal exp) (reset (a exp)))
```

The interesting part of this program is when the input is an application (the else branch of the function `a`). It first grabs the current continuation and creates fresh names, say `p` and `q`, for the operator and operand of the application. It then continues to the rest of the computation `k` as though the translation result of the current application was `(p q)`. After whole the translation is finished and the result is obtained, it adds the necessary bindings on top of it. If other application expressions appear during the recursive calls `(a (car exp))` or `(a (car (cdr exp)))`, other bindings are created and are added further on top the current bindings.

When the input expression is a lambda expression, `reset` is used so that the let bindings produced during the translation of the body do not appear outside the lambda expression.

If we CPS transform the above program, we obtain the familiar implementation of A-normal form translation [8]:

```
(define (a-cps exp cont)
  (cond ((symbol? exp) (cont exp)) ; variable
        ((eq? (car exp) ’lambda)
         ; (lambda (var) body)
         (let ((var  (car (car (cdr exp))))
               (body (car (cdr (cdr exp)))))
           (cont ‘(lambda (,var)
                    ,(a-cps body (lambda (x) x))))))
        (else ; application (m n)
          (let ((m (gensym ’m)) (n (gensym ’n)))
            (a-cps (car exp) (lambda (u)
              (a-cps (car (cdr exp)) (lambda (v)
                ‘(let* ((,m ,u) (,n ,v))
                   ,(cont ‘(,m ,n)))))))))))
(define (a-normal-cps exp)
  (a-cps exp (lambda (x) x)))
```

3. THE LANGUAGE

The input language we consider is a call-by-value left-to-right λ-calculus extended with shift and reset. Its syntax is defined as follows:

$$\boldsymbol{M} = \boldsymbol{x} \mid \boldsymbol{\lambda x.\, M} \mid \boldsymbol{M@M} \mid \boldsymbol{\xi k.\, M} \mid \langle \boldsymbol{M} \rangle$$

We use a bold font to emphasize that the input terms described by this syntax are data manipulated by programs, such as an interpreter, a CPS transformer, and a partial evaluator. We assume that the input language is typed according to the type system shown later in Section 6. The partial evaluator itself appears to work for untyped terms, but the correctness theorem critically depends on types.

When the input language is processed through a binding-time analysis, each term is annotated with either static or dynamic. The syntax for the annotated term is defined as follows:

$$\begin{array}{rcl} \boldsymbol{M} & = & \boldsymbol{x} \mid \overline{\boldsymbol{\lambda}}\boldsymbol{x.\, M} \mid \underline{\boldsymbol{\lambda}}\boldsymbol{x.\, M} \mid \boldsymbol{M}\overline{\boldsymbol{@}}\boldsymbol{M} \mid \boldsymbol{M}\underline{\boldsymbol{@}}\boldsymbol{M} \mid \\ & & \overline{\boldsymbol{\xi}}\boldsymbol{k.\, M} \mid \underline{\boldsymbol{\xi}}\boldsymbol{k.\, M} \mid \langle \boldsymbol{M} \rangle \end{array}$$

As usual, we use overlines and underlines to indicate static and dynamic expressions, respectively. There is no annotation for variables. The binding time of a variable is determined by the value of the variable. There is no annotation for reset, either. We will see that it can be always treated as static.

The metalanguage that is used to define various transformers is a standard functional language whose core syntax is defined using a standard font as follows:

$$M = x \mid \lambda x.\, M \mid M\, M$$

We use juxtaposition without @ for application. When necessary, we use richer constructs such as case expressions and if expressions. We assume that the metalanguage has a way to represent input terms and annotated terms as normal forms, such as datatypes found in a standard functional language. Wand [17] uses an explicit encoding through the Church representation for this purpose.

Throughout this paper, we use $=$ for definition or α-equality and $=_\beta$ for β-equality.

4. CPS INTERPRETER

A CPS interpreter defining the semantics of the input language is given as follows:

$$\begin{array}{rcl} \mathcal{I}\,\llbracket \boldsymbol{x} \rrbracket\,\rho\,\kappa & = & \kappa\,(\rho\,(\boldsymbol{x})) \\ \mathcal{I}\,\llbracket \boldsymbol{\lambda x.\, M} \rrbracket\,\rho\,\kappa & = & \kappa\,(\lambda x.\, \lambda k.\, \mathcal{I}\,\llbracket \boldsymbol{M} \rrbracket\,\rho[x/\boldsymbol{x}]\,k) \\ \mathcal{I}\,\llbracket \boldsymbol{M@N} \rrbracket\,\rho\,\kappa & = & \mathcal{I}\,\llbracket \boldsymbol{M} \rrbracket\,\rho\,\lambda m.\, \mathcal{I}\,\llbracket \boldsymbol{N} \rrbracket\,\rho\,\lambda n.\, m\,n\,\kappa \\ \mathcal{I}\,\llbracket \boldsymbol{\xi k.\, M} \rrbracket\,\rho\,\kappa & = & \mathcal{I}\,\llbracket \boldsymbol{M} \rrbracket\,\rho[\lambda v.\, \lambda k.\, k\,(\kappa\,(v))/\boldsymbol{k}]\,\lambda x.\, x \\ \mathcal{I}\,\llbracket \langle \boldsymbol{M} \rangle \rrbracket\,\rho\,\kappa & = & \kappa\,(\mathcal{I}\,\llbracket \boldsymbol{M} \rrbracket\,\rho\,\lambda x.\, x) \end{array}$$

It accepts a datum representing a term, an environment ρ, and a continuation κ, and returns a value represented in CPS. Given an environment ρ and a value V, $\rho[V/\boldsymbol{x}]$ is the same environment as ρ except that $\rho(\boldsymbol{x}) = V$.

The first three rules constitute a standard interpreter for the λ-calculus written in CPS. In the rule for abstractions, x and $\boldsymbol{x}$ are two different entities. The former is a variable in the metalanguage whereas the latter is a datum representing a variable named $\boldsymbol{x}$. The choice of the particular variable x for $\boldsymbol{x}$ does not have any effect on $\mathcal{I}$, but leads to a slightly simpler correctness proof when a partial evaluator is considered. Since the input language is left-to-right, $\boldsymbol{M}$ is evaluated before $\boldsymbol{N}$ in the rule for $\boldsymbol{M@N}$.

The rule for $\boldsymbol{\xi k.\, M}$ states that the continuation κ is captured as a function $\lambda v.\, \lambda k.\, k\,(\kappa\,(v))$, and bound to $\boldsymbol{k}$ for the later use. When applied, the function invokes the captured continuation κ. If the execution of κ finishes, the result is passed to the original continuation k. Namely, the captured continuation is *composable.* This is in contrast to call/cc in Scheme where the execution never comes back to the original continuation once the captured continuation is invoked. The identity continuation $\lambda x.\, x$ is used for the execution of the body $\boldsymbol{M}$ of $\boldsymbol{\xi k.\, M}$. It enables us to discard the captured continuation by not invoking it in $\boldsymbol{M}$. The use of $\lambda x.\, x$ is in contrast to call/cc where the value of $\boldsymbol{M}$ is passed to κ if the captured continuation is not applied in $\boldsymbol{M}$.

The rule for $\langle \boldsymbol{M} \rangle$ installs an identity continuation $\lambda x.\, x$ for the evaluation of $\boldsymbol{M}$. When it finishes, the result is applied to κ *in direct style.* Since κ is not passed to $\mathcal{I}$, the continuation captured by some $\boldsymbol{\xi k.\, N}$ in $\boldsymbol{M}$ is up to this enclosing $\langle \boldsymbol{M} \rangle$, and does not include the continuation κ. Namely, the continuation is *delimited* by $\langle \boldsymbol{M} \rangle$.

5. CPS TRANSFORMER

The semantics of the input language can be given not only by a CPS interpreter but also CPS transformation. Providing both an interpreter and a transformer would facilitate the construction of a specializer, as we will see in Section 8.

A CPS transformer for the input language is defined as follows:

$$\begin{array}{rcl} \mathcal{C}\,\llbracket \boldsymbol{x} \rrbracket\,\rho\,\kappa & = & \kappa\,(\rho\,(\boldsymbol{x})) \\ \mathcal{C}\,\llbracket \boldsymbol{\lambda x.\, M} \rrbracket\,\rho\,\kappa & = & \kappa\,(\boldsymbol{\lambda x^\diamond.\, \lambda k^\diamond.}\, \mathcal{C}\,\llbracket \boldsymbol{M} \rrbracket\,\rho[\boldsymbol{x}^\diamond/\boldsymbol{x}]\,\lambda v.\, \boldsymbol{k}^\diamond\boldsymbol{@}v) \\ \mathcal{C}\,\llbracket \boldsymbol{M@N} \rrbracket\,\rho\,\kappa & = & \mathcal{C}\,\llbracket \boldsymbol{M} \rrbracket\,\rho\,\lambda m.\, \mathcal{C}\,\llbracket \boldsymbol{N} \rrbracket\,\rho\,\lambda n.\, m\boldsymbol{@}n\boldsymbol{@\lambda t^\diamond.}\, \kappa\,(\boldsymbol{t}^\diamond) \\ \mathcal{C}\,\llbracket \boldsymbol{\xi k.\, M} \rrbracket\,\rho\,\kappa & = & \mathcal{C}\,\llbracket \boldsymbol{M} \rrbracket\,\rho[\boldsymbol{\lambda v^\diamond.\, \lambda k^\diamond.\, k^\diamond @}(\kappa\,(\boldsymbol{v}^\diamond))/\boldsymbol{k}]\,\lambda x.\, x \\ \mathcal{C}\,\llbracket \langle \boldsymbol{M} \rangle \rrbracket\,\rho\,\kappa & = & \kappa\,(\mathcal{C}\,\llbracket \boldsymbol{M} \rrbracket\,\rho\,\lambda x.\, x) \end{array}$$

A variable with the superscript $^\diamond$ indicates that it is a fresh variable (produced by gensym). The CPS transformer $\mathcal{C}$ accepts a datum representing a term, an environment ρ, and a continuation κ, and returns the CPS counterpart of the input term. Since the output is a datum, it is in bold font. The conventional CPS transformation does not use an environment but uses the original variable itself as the CPS transformation of a variable. Here, we allow renaming of variables using an environment to make better correspondence to the partial evaluator.

The first three rules constitute a CPS transformer for the λ-calculus. A variable is translated to what is stored in the environment ρ. A lambda abstraction is transformed to a CPS term where the bound variable $\boldsymbol{x}$ is renamed to a variable $\boldsymbol{x}^\diamond$. η-expansion is used to turn a dynamic continuation $\boldsymbol{k}^\diamond$ into a static continuation $\lambda v.\, \boldsymbol{k}^\diamond\boldsymbol{@}v$ to make all the continuations passed to $\mathcal{C}$ be static. It enables one-pass CPS transformation [6] without producing the so-called *administrative* β-redexes. The transformation of an application proceeds by first transforming its function and argument. Then, their results are collected to produce an application expression. The result of the application is given a fresh name $\boldsymbol{t}^\diamond$, which is passed to the rest of the transformation. Here, η-expansion is used to turn a static continuation κ into a representation $\boldsymbol{\lambda t^\diamond.}\, \kappa\,(\boldsymbol{t}^\diamond)$, which needs to appear in the result of CPS transformation.

The rule for $\boldsymbol{\xi k.\, M}$ is similar to the one in the interpreter. It captures the transformation-time continuation κ

and binds it to $\boldsymbol{k}$. The only difference is that the bound value is a datum representing a CPS term rather than a CPS value. The rule for $\langle \boldsymbol{M} \rangle$ is exactly the same as the one in the interpreter.

6. TYPE SYSTEM

We now show a type system for our input language. This type system is due to Danvy and Filinski [4].

Conventionally, each term is associated with one type, *e.g.*, 3 has a type integer. In the presence of shift and reset, however, one type is not sufficient. Consider the term $\boldsymbol{\xi k.\, k}$. When executed in a context, it grabs the current continuation (up to the enclosing reset) and immediately returns it. Thus, the type of $\boldsymbol{\xi k.\, k}$ is the same as the type of its continuation. So when we talk about types of terms, it is natural to think about the types of their continuations, too. Following Danvy and Filinski's notation [4], we use a judgment of the following form:

$$A, \alpha \vdash \boldsymbol{M} : \tau, \beta$$

Here, A is a type environment mapping free variables in $\boldsymbol{M}$ to their types. It says that a term $\boldsymbol{M}$ has a type τ in a continuation of type $\tau \to \alpha$ and returns a value of type β. The intention here is that if $\boldsymbol{M}$ captures its continuation, it has a type $\tau \to \alpha$; using the captured continuation, $\boldsymbol{M}$ continues its computation and finally results in a value of type β. For example, in the expression $\langle 1 + \xi k.\, (k\,(k\,0) = 3)\rangle$, the term $\xi k.\, (k\,(k\,0) = 3)$ has a type int and its continuation has a type $\mathsf{int} \to \mathsf{int}$. Using the continuation of this type, it returns a boolean value false as a result of the whole expression. If $\boldsymbol{M}$ does not grab its continuation and use it in a non-standard way, the types α and β are always the same.

Taking into account the types of continuations affects the definition of a function type. The traditional function type $\sigma \to \tau$ consists of two types, the types of the argument and the result. The function type we use, on the other hand, consists of four types: $\sigma/\alpha \to \tau/\beta$. It represents a function that receives a value of type σ, passes a value of type τ to a continuation of type $\tau \to \alpha$, and gives the final result of type β. Thus, the syntax of the type is defined as:

$$\tau \quad = \quad o \mid \tau/\tau \to \tau/\tau$$

for some o representing a base type (*e.g.*, integer). In the following, the traditional function type $\sigma \to \tau$ appears only for the terms after CPS transformation.

Since the judgment of the form $A, \alpha \vdash \boldsymbol{M} : \tau, \beta$ and the function type of the form $\sigma/\alpha \to \tau/\beta$ are not familiar to most readers, before presenting the type system itself, we first state its soundness theorem relative to the CPS interpreter. It helps us understand the type system. Moreover, it establishes and makes clear the relationship between the type system and the CPS interpreter, and also between the function type and the traditional one.

For each type τ, define the domain D_τ as follows:

$$\begin{array}{rcl} D_o & = & \mathsf{N}_\perp \\ D_{\sigma/\alpha \to \tau/\beta} & = & [D_\sigma \to [D_\tau \to D_\alpha] \to D_\beta] \end{array}$$

where $\mathsf{N}_\perp$ is a domain for the base type o with a bottom element $\perp$ and $[D_\tau \to D_\alpha]$ is a standard domain of continuous functions from D_τ to D_α. We assume that $\to$ associates to the right and write $[D_\alpha \to D_\sigma \to D_\tau]$ to mean $[D_\alpha \to [D_\sigma \to D_\tau]]$. For an environment ρ, we say $\rho \models A$ iff for all $\boldsymbol{x} \in \mathit{dom}(A)$, $\rho\,(\boldsymbol{x}) \in D_{A(\boldsymbol{x})}$.

Theorem 1. *If* $A, \alpha \vdash \boldsymbol{M} : \tau, \beta$ *and* $\rho \models A$*, then* $\mathcal{I}\,\llbracket \boldsymbol{M} \rrbracket\, \rho \in [[D_\tau \to D_\alpha] \to D_\beta]$.

In the following paragraphs, we show the typing rules one by one. There are five cases, one for each syntactic form. In each case, we state the relationship to the CPS interpreter, which serves as a subproof of the above theorem. As a whole, they constitute the proof of the theorem done by induction on the construction of $A, \alpha \vdash \boldsymbol{M} : \tau, \beta$.

Variable. The typing rule for variables goes as follows:

$$A[\boldsymbol{x} : \tau], \alpha \vdash \boldsymbol{x} : \tau, \alpha$$

The type of a variable is stored in the type environment. Here, $A[\boldsymbol{x} : \tau]$ is the same type environment as A except that $\boldsymbol{x}$ is mapped to τ. Since a variable cannot capture its continuation, the type of the final result α becomes the same as the return type of its continuation.

The relationship to the CPS interpreter can be illustrated by decorating it with type information as superscripts:[1]

$$\mathcal{I}\,\llbracket \boldsymbol{x} \rrbracket\, \rho\, \kappa^{\tau \to \alpha} \quad = \quad (\kappa^{\tau \to \alpha}\, (\rho\,(\boldsymbol{x}))^{\tau})^{\alpha}$$

Assuming that $\rho\,(\boldsymbol{x}) \in D_\tau$ and $\kappa \in [D_\tau \to D_\alpha]$, we have $\mathcal{I}\,\llbracket \boldsymbol{x} \rrbracket\, \rho\, \kappa \in D_\alpha$, which proves Theorem 1 for this case.

Abstraction. The typing rule for abstractions is:

$$\frac{A[\boldsymbol{x} : \sigma], \alpha \vdash \boldsymbol{M} : \tau, \beta}{A, \delta \vdash \boldsymbol{\lambda x.\, M} : \sigma/\alpha \to \tau/\beta, \delta}$$

Since constructing a function itself does not alter the context, the same δ is used for the result of continuation and the final result. The body $\boldsymbol{M}$ of $\boldsymbol{\lambda x.\, M}$ is typed according to the type of the function: assuming that $\boldsymbol{x}$ is of type σ, $\boldsymbol{M}$ has a type τ in the continuation of type $\tau \to \alpha$ and the result type should be β.

The corresponding CPS interpreter with type information is:

$$\begin{array}{rl} & \mathcal{I}\,\llbracket \boldsymbol{\lambda x.\, M} \rrbracket\, \rho\, \kappa^{(\sigma \to (\tau \to \alpha) \to \beta) \to \delta} \\ = & (\kappa\, (\lambda x^{\sigma}.\, \lambda k^{\tau \to \alpha}.\, (\mathcal{I}\,\llbracket \boldsymbol{M} \rrbracket\, \rho[x^{\sigma}/\boldsymbol{x}]\, k^{\tau \to \alpha})^{\beta}))^{\delta} \end{array}$$

To prove Theorem 1 for this case, we observe the premise $A[\boldsymbol{x} : \sigma], \alpha \vdash \boldsymbol{M} : \tau, \beta$ of the typing rule corresponds to $(\mathcal{I}\,\llbracket \boldsymbol{M} \rrbracket\, \rho[x^{\sigma}/\boldsymbol{x}]\, k^{\tau \to \alpha})^{\beta}$. Assuming that $k \in [D_\tau \to D_\alpha]$, the induction hypothesis gives us $\mathcal{I}\,\llbracket \boldsymbol{M} \rrbracket\, \rho[x/\boldsymbol{x}]\, k \in D_\beta$ since $\rho[x/\boldsymbol{x}] \models A[\boldsymbol{x} : \sigma]$. Using this induction hypothesis, we obtain $\lambda x.\, \lambda k.\, (\mathcal{I}\,\llbracket \boldsymbol{M} \rrbracket\, \rho[x/\boldsymbol{x}]\, k) \in [D_\sigma \to [D_\tau \to D_\alpha] \to D_\beta] = D_{\sigma/\alpha \to \tau/\beta}$. Assuming that $\kappa \in [D_{\sigma/\alpha \to \tau/\beta} \to D_\delta]$, we have $\kappa\, (\lambda x.\, \lambda k.\, \mathcal{I}\,\llbracket \boldsymbol{M} \rrbracket\, \rho[x/\boldsymbol{x}]\, k) \in D_\delta$ as expected.

Application. The typing rule for applications goes as follows:

$$\frac{A, \delta \vdash \boldsymbol{M} : \sigma/\alpha \to \tau/\epsilon, \beta \quad A, \epsilon \vdash \boldsymbol{N} : \sigma, \delta}{A, \alpha \vdash \boldsymbol{M@N} : \tau, \beta}$$

We want to say that $\boldsymbol{M@N}$ has a type τ in a continuation of type $\tau \to \alpha$ and returns a value of type β. Since the input language is left-to-right, we first need to check that $\boldsymbol{M}$ has a function type. At this time, the result type β is inherited from the result type of $\boldsymbol{M@N}$, because $\boldsymbol{M}$ might capture its continuation and terminate the evaluation of the whole

---

[1] We write $\kappa^{\tau \to \sigma}$ to mean that κ is in the domain $[D_\tau \to D_\sigma]$, and so on.

$\boldsymbol{M}\,\boldsymbol{@}\boldsymbol{N}$. Now, suppose that $\boldsymbol{M}$ returns a function (or it captures its continuation and passes a function to it). Since the continuation of the evaluation of $\boldsymbol{M}$ is that of $\boldsymbol{N}$, the result type of $\boldsymbol{N}$ must be the return type of the continuation of $\boldsymbol{M}$, namely δ. Notice that $\boldsymbol{N}$ does not have control over the result type β of the whole application $\boldsymbol{M}\,\boldsymbol{@}\boldsymbol{N}$. Even though $\boldsymbol{N}$ captures its continuation and discards it, it is only effective if $\boldsymbol{M}$ allows it, namely, if $\boldsymbol{M}$ passes a value to its continuation.

This typing rule naturally reflects the CPS interpreter:

$$\begin{array}{ll} & \mathcal{I}\llbracket \boldsymbol{M}\,\boldsymbol{@}\boldsymbol{N} \rrbracket\, \rho\, \kappa^{\tau\to\alpha} \\ = & (\mathcal{I}\llbracket \boldsymbol{M} \rrbracket\, \rho\, \lambda m^{\sigma\to(\tau\to\alpha)\to\epsilon}.\, (\mathcal{I}\llbracket \boldsymbol{N} \rrbracket\, \rho\, \lambda n^{\sigma}.\, m\, n\, \kappa)^{\delta})^{\beta} \end{array}$$

To prove Theorem 1 for this case, we first find that the first premise of the typing rule corresponds to the whole expression $(\mathcal{I}\llbracket \boldsymbol{M} \rrbracket\, \rho\, \lambda m.\ \cdots)^{\beta}$, which gives the result type β. The induction hypothesis then directly proves $\mathcal{I}\llbracket \boldsymbol{M}\,\boldsymbol{@}\boldsymbol{N} \rrbracket\, \rho\, \kappa \in D_{\beta}$ provided that we can prove

$$\begin{array}{rl} \lambda m.\, (\mathcal{I}\llbracket \boldsymbol{N} \rrbracket\, \rho\, \lambda n.\, m\, n\, \kappa) \in & [D_{\sigma/\alpha\to\tau/\epsilon} \to D_{\delta}] \\ = & [[D_{\sigma} \to [D_{\tau} \to D_{\alpha}] \to D_{\epsilon}] \to D_{\delta}] \end{array}$$

So suppose that $m \in [D_{\sigma} \to [D_{\tau} \to D_{\alpha}] \to D_{\epsilon}]$. We need to show $\mathcal{I}\llbracket \boldsymbol{N} \rrbracket\, \rho\, \lambda n.\, m\, n\, \kappa \in D_{\delta}$, but it comes from the induction hypothesis for the second premise $(\mathcal{I}\llbracket \boldsymbol{N} \rrbracket\, \rho\, \lambda n^{\sigma}.\, m\, n\, \kappa)^{\delta}$, provided that we can prove $\lambda n.\, m\, n\, \kappa \in [D_{\sigma} \to D_{\epsilon}]$. Assuming that $n \in D_{\sigma}$, the last statement holds as expected since $m\, n\, \kappa \in D_{\epsilon}$.

Shift. The typing rule for shift is:

$$\frac{A[\boldsymbol{k} : \tau/\delta \to \alpha/\delta], \sigma \vdash \boldsymbol{M} : \sigma, \beta}{A, \alpha \vdash \boldsymbol{\xi k.\, M} : \tau, \beta}$$

$\boldsymbol{\xi k.\, M}$ captures its continuation of type $\tau \to \alpha$ and binds it to $\boldsymbol{k}$. Since the captured continuation itself does not alter its context, the type of $\boldsymbol{k}$ contains δ twice. The continuation of the body $\boldsymbol{M}$ has a type $\sigma \to \sigma$ since it is supplied with an identity continuation.

The corresponding CPS interpreter is:

$$\begin{array}{ll} & \mathcal{I}\llbracket \boldsymbol{\xi k.\, M} \rrbracket\, \rho\, \kappa^{\tau\to\alpha} \\ = & (\mathcal{I}\llbracket \boldsymbol{M} \rrbracket\, \rho[\lambda v^{\tau}.\, \lambda k^{\alpha\to\delta}.\, k\,(\kappa\,(v))/\boldsymbol{k}]\, \lambda x^{\sigma}.\, x^{\sigma})^{\beta} \end{array}$$

To prove Theorem 1 for this case, assume that $v \in D_{\tau}$, $k \in [D_{\alpha} \to D_{\delta}]$, and $\kappa \in [D_{\tau} \to D_{\alpha}]$. Since we then have $k\,(\kappa\,(v)) \in D_{\delta}$, it follows that $\lambda v^{\tau}.\, \lambda k^{\alpha\to\delta}.\, k\,(\kappa\,(v)) \in [D_{\tau} \to [D_{\alpha} \to D_{\delta}] \to D_{\delta}] = D_{\tau/\delta\to\alpha/\delta}$. Thus, we have $\rho[\lambda v^{\tau}.\, \lambda k.\, k\,(\kappa\,(v))/\boldsymbol{k}] \models A[\boldsymbol{k} : \tau/\delta \to \alpha/\delta]$. Now, we can use the induction hypothesis to show $\mathcal{I}\llbracket \boldsymbol{\xi k.\, M} \rrbracket\, \rho\, \kappa \in D_{\beta}$.

Reset. Finally, the typing rule for reset is:

$$\frac{A, \sigma \vdash \boldsymbol{M} : \sigma, \tau}{A, \alpha \vdash \langle \boldsymbol{M} \rangle : \tau, \alpha}$$

α occurs twice since $\langle \boldsymbol{M} \rangle$ does not alter its context, and σ occurs twice since $\boldsymbol{M}$ is supplied with an identity continuation.

The CPS interpreter is:

$$\mathcal{I}\llbracket \langle \boldsymbol{M} \rangle \rrbracket\, \rho\, \kappa^{\tau\to\alpha} \quad = \quad (\kappa\,(\mathcal{I}\llbracket \boldsymbol{M} \rrbracket\, \rho\, \lambda x^{\sigma}.\, x^{\sigma})^{\tau})^{\alpha}$$

From the induction hypothesis, we have $\mathcal{I}\llbracket \boldsymbol{M} \rrbracket\, \rho\, \lambda x.\, x \in D_{\tau}$. From $\kappa \in [D_{\tau} \to D_{\alpha}]$, we have $\mathcal{I}\llbracket \langle \boldsymbol{M} \rangle \rrbracket\, \rho\, \kappa \in D_{\alpha}$, which proves Theorem 1 for this case.

7. BINDING-TIME ANNOTATION

The offline partial evaluator is divided into two phases: a binding-time analysis and specialization. The binding-time analysis produces an annotated version of the input term by assigning each subterm a binding-time type. The binding-time type we use is defined as follows:

$$\tau \quad = \quad s \mid d \mid \tau/\tau \to \tau/\tau$$

The intention is that the type s denotes the type of "static" constants, the type d denotes the type of "dynamic" quantities (either constants or functions), and the type $\tau/\tau \to \tau/\tau$ denotes the type of static functions.

The annotated terms are produced by decorating a type-inference tree. Thus, instead of considering typing judgments of the form $A, \alpha \vdash \boldsymbol{M} : \tau, \beta$ as we did in the previous section, we now consider judgments of the form $A, \alpha \vdash \boldsymbol{M} : \tau, \beta\ [\boldsymbol{W}]$, meaning that a term $\boldsymbol{M}$ can be annotated as $\boldsymbol{W}$. The type system can be written as follows:

$$A[\boldsymbol{x} : \tau], \alpha \vdash \boldsymbol{x} : \tau, \alpha\ [\boldsymbol{x}] \quad (\mathit{var})$$

$$\frac{A[\boldsymbol{x} : \sigma], \alpha \vdash \boldsymbol{M} : \tau, \beta\ [\boldsymbol{W}]}{A, \delta \vdash \boldsymbol{\lambda x.\, M} : \sigma/\alpha \to \tau/\beta, \delta\ [\overline{\boldsymbol{\lambda}}\boldsymbol{x.\, W}]}\ (\mathit{abs\text{-}s})$$

$$\frac{A[\boldsymbol{x} : d], d \vdash \boldsymbol{M} : d, d\ [\boldsymbol{W}]}{A, \delta \vdash \boldsymbol{\lambda x.\, M} : d, \delta\ [\underline{\boldsymbol{\lambda}}\boldsymbol{x.\, W}]}\ (\mathit{abs\text{-}d})$$

$$\frac{\begin{array}{l} A, \delta \vdash \boldsymbol{M_1} : \sigma/\alpha \to \tau/\epsilon, \beta\ [\boldsymbol{W_1}] \\ A, \epsilon \vdash \boldsymbol{M_2} : \sigma, \delta\ [\boldsymbol{W_2}] \end{array}}{A, \alpha \vdash \boldsymbol{M_1 @ M_2} : \tau, \beta\ [\boldsymbol{W_1}\overline{\boldsymbol{@}}\boldsymbol{W_2}]}\ (\mathit{app\text{-}s})$$

$$\frac{A, \delta \vdash \boldsymbol{M_1} : d, \beta\ [\boldsymbol{W_1}] \quad A, d \vdash \boldsymbol{M_2} : d, \delta\ [\boldsymbol{W_2}]}{A, d \vdash \boldsymbol{M_1 @ M_2} : d, \beta\ [\boldsymbol{W_1}\underline{\boldsymbol{@}}\boldsymbol{W_2}]}\ (\mathit{app\text{-}d})$$

$$\frac{A[\boldsymbol{k} : \tau/\delta \to \alpha/\delta], \sigma \vdash \boldsymbol{M} : \sigma, \beta\ [\boldsymbol{W}]}{A, \alpha \vdash \boldsymbol{\xi k.\, M} : \tau, \beta\ [\overline{\boldsymbol{\xi}}\boldsymbol{k.\, W}]}\ (\mathit{shift\text{-}s})$$

$$\frac{A[\boldsymbol{k} : d], \sigma \vdash \boldsymbol{M} : \sigma, \beta\ [\boldsymbol{W}]}{A, d \vdash \boldsymbol{\xi k.\, M} : d, \beta\ [\underline{\boldsymbol{\xi}}\boldsymbol{k.\, W}]}\ (\mathit{shift\text{-}d})$$

$$\frac{A, \sigma \vdash \boldsymbol{M} : \sigma, \tau\ [\boldsymbol{W}]}{A, \alpha \vdash \langle \boldsymbol{M} \rangle : \tau, \alpha\ [\langle \boldsymbol{W} \rangle]}\ (\mathit{reset})$$

Among the eight rules, (*var*), (*abs-s*), (*app-s*), (*shift-s*), and (*reset*) are exactly the same as the five rules shown in the previous section except that annotated terms are attached. We add three rules for dynamic abstractions (*abs-d*), dynamic applications (*app-d*), and dynamic shift (*shift-d*). They are obtained by replacing the function type appearing in the static rule with d. For example, (*abs-d*) is obtained by replacing $\sigma/\alpha \to \tau/\beta$ in (*abs-s*) as well as all of its components (*i.e.*, σ, α, τ, and β) with d. The resulting rule (*abs-d*) reads: if $\boldsymbol{M}$ is dynamic assuming $\boldsymbol{x}$ is dynamic and in the dynamic continuation (of type $d \to d$), then $\boldsymbol{\lambda x.\, M}$ is also dynamic. Even though $\boldsymbol{\lambda x.\, M}$ is dynamic, the final result need not to be dynamic. Its continuation could discard the dynamic abstraction and return a static result. This is why we have still a type variable δ in (*abs-d*). (*app-d*) and (*shift-d*) are obtained similarly.

Note that there is no rule for dynamic reset. This is because it is always static: reset simply delimits the continuation, which is statically determinable at specialization time.

8. SPECIALIZER

Once the input term is annotated according to the typing rules shown in the previous section, it is passed to a specializer. The specializer described in this paper produces a specialized result in CPS. It is a mixture of the CPS interpreter and the CPS transformer: when the input term is static, it is statically reduced as in the CPS interpreter, whereas when the input term is dynamic, it produces the output in CPS as in the CPS transformer. The specializer is given as follows:

$$\begin{aligned}
\mathcal{P}\llbracket \boldsymbol{x} \rrbracket\, \rho\, \kappa &= \kappa\, (\rho\, (\boldsymbol{x})) \\
\mathcal{P}\llbracket \overline{\boldsymbol{\lambda}}\boldsymbol{x}\textbf{.}\, \boldsymbol{M} \rrbracket\, \rho\, \kappa &= \kappa\, (\lambda x.\, \lambda k.\, \mathcal{P}\llbracket \boldsymbol{M} \rrbracket\, \rho[x/\boldsymbol{x}]\, k) \\
\mathcal{P}\llbracket \underline{\boldsymbol{\lambda}}\boldsymbol{x}\textbf{.}\, \boldsymbol{M} \rrbracket\, \rho\, \kappa &= \kappa\, (\boldsymbol{\lambda x^{\diamond}}\textbf{.}\, \boldsymbol{\lambda k^{\diamond}}\textbf{.}\, \mathcal{P}\llbracket M \rrbracket\, \rho[\boldsymbol{x}^{\diamond}/\boldsymbol{x}]\, \lambda v.\, \boldsymbol{k}^{\diamond}\boldsymbol{@}v) \\
\mathcal{P}\llbracket \boldsymbol{M}\overline{\boldsymbol{@}}\boldsymbol{N} \rrbracket\, \rho\, \kappa &= \mathcal{P}\llbracket \boldsymbol{M} \rrbracket\, \rho\, \lambda m.\, \mathcal{P}\llbracket \boldsymbol{N} \rrbracket\, \rho\, \lambda n.\, m\, n\, \kappa \\
\mathcal{P}\llbracket \boldsymbol{M}\underline{\boldsymbol{@}}\boldsymbol{N} \rrbracket\, \rho\, \kappa &= \mathcal{P}\llbracket \boldsymbol{M} \rrbracket\, \rho\, \lambda m.\, \mathcal{P}\llbracket \boldsymbol{N} \rrbracket\, \rho\, \lambda n.\, m\boldsymbol{@}n\boldsymbol{@}\boldsymbol{\lambda t^{\diamond}}\textbf{.}\, \kappa\, (\boldsymbol{t}^{\diamond}) \\
\mathcal{P}\llbracket \overline{\boldsymbol{\xi}}\boldsymbol{k}\textbf{.}\, \boldsymbol{M} \rrbracket\, \rho\, \kappa &= \mathcal{P}\llbracket \boldsymbol{M} \rrbracket\, \rho[\lambda v.\, \lambda k.\, k\, (\kappa\, (v))/\boldsymbol{k}]\, \lambda x.\, x \\
\mathcal{P}\llbracket \underline{\boldsymbol{\xi}}\boldsymbol{k}\textbf{.}\, \boldsymbol{M} \rrbracket\, \rho\, \kappa &= \mathcal{P}\llbracket \boldsymbol{M} \rrbracket\, \rho[\boldsymbol{\lambda v^{\diamond}}\textbf{.}\, \boldsymbol{\lambda k^{\diamond}}\textbf{.}\, \boldsymbol{k}^{\diamond}\boldsymbol{@}(\kappa\, (\boldsymbol{v}^{\diamond}))/\boldsymbol{k}]\, \lambda x.\, x \\
\mathcal{P}\llbracket \langle \boldsymbol{M} \rangle \rrbracket\, \rho\, \kappa &= \kappa\, (\mathcal{P}\llbracket \boldsymbol{M} \rrbracket\, \rho\, \lambda x.\, x)
\end{aligned}$$

Whenever an abstraction is residualized during partial evaluation, new names need to be used for its argument and continuation. Since our CPS transformer is already designed to introduce fresh names for them, the partial evaluator is literally the mixture of the CPS interpreter and the CPS transformer, except for the use of overlines and underlines in the input language. In particular, if the input term is annotated as completely dynamic, the specializer behaves exactly the same as the CPS transformer.

9. CORRECTNESS

We show the correctness of $\mathcal{P}$ by using the technique of logical relations following Wand [17]. Because the output of the specializer is in CPS and does not contain shift and reset, we do not have to consider the logical relation of terms with shift and reset, but rather we can use the standard logical relation on λ-terms.

We first define an evaluator for terms without shift and reset. It is used to interpret the output of our specializer.

$$\begin{aligned}
\mathcal{E}\llbracket \boldsymbol{x} \rrbracket\, \mu &= \mu\, (\boldsymbol{x}) \\
\mathcal{E}\llbracket \boldsymbol{\lambda x}\textbf{.}\, \boldsymbol{M} \rrbracket\, \mu &= \lambda x.\, \mathcal{E}\llbracket \boldsymbol{M} \rrbracket\, \mu[x/\boldsymbol{x}] \\
\mathcal{E}\llbracket \boldsymbol{M@N} \rrbracket\, \mu &= (\mathcal{E}\llbracket \boldsymbol{M} \rrbracket\, \mu)\, (\mathcal{E}\llbracket \boldsymbol{N} \rrbracket\, \mu)
\end{aligned}$$

It evaluates the input term as in the CPS interpreter $\mathcal{I}$. It differs from $\mathcal{I}$ in that it is written in direct style and does not include the cases for shift and reset.

Then, we define a binary relation R_τ on λ-terms for every type τ as follows:

$$\begin{aligned}
(M, M') \in R_s &\iff M = M' \\
(M, M') \in R_d &\iff \mathcal{E}\llbracket M \rrbracket\, \mu_{id} =_\beta M' \\
(M, M') \in R_{\sigma\to\tau} &\iff \text{for all } (N, N') \text{ such that } (N, N') \in R_\sigma, \\
&\qquad\ (M\, N, M'\, N') \in R_\tau
\end{aligned}$$

where $\mu_{id}(\boldsymbol{x}) = x$ for all $\boldsymbol{x}$. The use of μ_{id} in the definition of $(M, M') \in R_d$ shows that the relation is defined for open terms. We relate a datum M representing an open term containing a free variable $\boldsymbol{x}$ to an open λ-term that uses the free variable named x for $\boldsymbol{x}$ in the metalanguage.

Let the CPS transform τ^* of a type τ be defined as follows:

$$\begin{aligned}
s^* &= s \\
d^* &= d \\
(\sigma/\alpha \to \tau/\beta)^* &= \sigma^* \to (\tau^* \to \alpha^*) \to \beta^*
\end{aligned}$$

For environments ρ and ρ', we say $(\rho, \rho') \models A$ iff for all $\boldsymbol{x} \in \mathit{dom}(A)$, $(\rho\, (\boldsymbol{x}), \rho'\, (\boldsymbol{x})) \in R_{A(\boldsymbol{x})^*}$. Then, we can state the main theorem:

Theorem 2. *If* $A, \alpha \vdash \boldsymbol{M} : \tau, \beta\ [\boldsymbol{W}]$, $(\rho, \rho') \models A$, *and* $(\kappa, \kappa') \in R_{\tau^* \to \alpha^*}$, *then* $(\mathcal{P}\llbracket \boldsymbol{W} \rrbracket\, \rho\, \kappa, \mathcal{I}\llbracket \boldsymbol{M} \rrbracket\, \rho'\, \kappa') \in R_{\beta^*}$.

The theorem says roughly that the result of partially evaluating the annotated term $\boldsymbol{W}$ is related by R_{β^*} to the interpretation of the unannotated term $\boldsymbol{M}$ via the CPS interpreter. There are two crucial points to notice here. First, because our type system specifies the result type, we can talk about the relation between two CPS programs. If we used a traditional type system that specifies only a type of expressions, we could not make a statement about the entire execution, since the type system only told us about the type of $\boldsymbol{M}$ (and $\boldsymbol{W}$) and not the type of the final result. Second, because our type system specifies the type of continuations, we can now place a proper assumption about continuations: $(\kappa, \kappa') \in R_{\tau^* \to \alpha^*}$. Without this assumption, the proof does not go through.

Once the theorem statement is identified, it is easy to write down the proof. It is by induction on the construction of the proof of $A, \alpha \vdash \boldsymbol{M} : \tau, \beta\ [\boldsymbol{W}]$. There are eight cases, one for each proof rule. All the cases follow the CPS structure of the specializer. In particular, the cases for application (both static and dynamic) proceed left-to-right. The proof is shown in Appendix A.

A special case of the theorem establishes the correctness of partial evaluation. Let A_ϕ be an empty type environment and ρ_ϕ an empty environment.

Corollary 1. *If* $A_\phi, d \vdash \boldsymbol{M} : d, d\ [\boldsymbol{W}]$, *then* $\mathcal{E}\llbracket \mathcal{P}\llbracket \boldsymbol{W} \rrbracket\, \rho_\phi\, \lambda x.\, x \rrbracket\, \mu_{id} =_\beta \mathcal{I}\llbracket \boldsymbol{M} \rrbracket\, \rho_\phi\, \lambda x.\, x$.

Proof. Since $(\rho_\phi, \rho'_\phi) \models A_\phi$ and $(\lambda x.\, x, \lambda x.\, x) \in R_{d\to d}$, we have $(\mathcal{P}\llbracket \boldsymbol{W} \rrbracket\, \rho_\phi\, \lambda x.\, x, \mathcal{I}\llbracket \boldsymbol{M} \rrbracket\, \rho_\phi\, \lambda x.\, x) \in R_d$ from the theorem. The statement directly follows from the definition of R_d. ☐

Since the partial evaluator $\mathcal{P}$ behaves exactly the same as the CPS transformer $\mathcal{C}$ if the input term is annotated as completely dynamic, we obtain the following corollary, which establishes the correctness of CPS transformation.

Corollary 2. $\mathcal{E}\llbracket \mathcal{C}\llbracket \boldsymbol{M} \rrbracket\, \rho_\phi\, \lambda x.\, x \rrbracket\, \mu_{id} =_\beta \mathcal{I}\llbracket \boldsymbol{M} \rrbracket\, \rho_\phi\, \lambda x.\, x$.

10. BINDING-TIME ANALYSIS

The typing rules shown in Section 7 can be used to check that annotated terms are well-typed. However, it cannot be used to infer annotations from unannotated terms, because the typing rules are not syntax directed. For example, we have two rules for applications, (*app-s*) and (*app-d*). It is not clear which rule to use when we are given an unannotated term. In this section, we describe how to obtain annotations from unannotated terms. The method shown in this section is standard and not particular to shift and reset. The basic idea is to merge the rules so that the typing rules become syntax-directed.

We assume that the term is already typed in the type system shown in Section 6. Then, the task of the binding-time analysis is to annotate the type with either s (static) or d (dynamic). Define the binding time b and the binding-time annotated type τ as follows:

$$\begin{aligned}
b &= s \mid d \\
\tau &= o^b \mid (\tau/\tau \to \tau/\tau)^b
\end{aligned}$$

We write τ^b to mean that the top-level binding time of τ is b. Not all function types are well-formed: the type of arguments and results must be dynamic when a function itself is dynamic. Define an inequality between binding times as $s < d$. Then, the well-formedness of binding-time types is defined as follows.

- o^b is well-formed.
- $(\sigma^{b_\sigma}/\alpha^{b_\alpha} \to \tau^{b_\tau}/\beta^{b_\beta})^b$ is well-formed if $b \le b_\sigma, b \le b_\alpha, b \le b_\tau$, and $b \le b_\beta$.

The constraint of the form $b_1 \le b_2$ expresses that if $b_1 = d$, so must be b_2. (When $b_1 = s$, b_2 can be either s or d; the inequality holds regardless of the value of b_2.)

The typing rules for annotated terms can be made syntax-directed by merging the static and dynamic rules. Remember that in Section 7, we obtained dynamic rules for annotated terms by simply making all the function types dynamic. Thus, to merge static and dynamic rules, we simply use the static rules only, but with the additional constraints on binding times: if the type of a function becomes dynamic, so must be all of its components. The resulting binding-time inference rules are:

$$A[\boldsymbol{x} : \tau^b], \alpha \vdash \boldsymbol{x} : \tau^b, \alpha \quad (\mathit{var})$$

$$\frac{A[\boldsymbol{x} : \sigma], \alpha \vdash \boldsymbol{M} : \tau, \beta}{A, \delta \vdash \boldsymbol{\lambda x.\, M} : (\sigma^{b_\sigma}/\alpha^{b_\alpha} \to \tau^{b_\tau}/\beta^{b_\beta})^b, \delta} \; (\mathit{abs})$$

constraints for (abs): $\{b \le b_\sigma, b \le b_\alpha, b \le b_\tau, b \le b_\beta\}$

$$\frac{\begin{array}{l} A, \delta \vdash \boldsymbol{M_1} : (\sigma^{b_\sigma}/\alpha^{b_\alpha} \to \tau^{b_\tau}/\epsilon^{b_\epsilon})^b, \beta \\ A, \epsilon \vdash \boldsymbol{M_2} : \sigma, \delta \end{array}}{A, \alpha \vdash \boldsymbol{M_1 @ M_2} : \tau, \beta} \; (\mathit{app})$$

constraints for (app): $\{b \le b_\sigma, b \le b_\alpha, b \le b_\tau, b \le b_\epsilon\}$

$$\frac{A[\boldsymbol{k} : (\tau^{b_\tau}/\delta^{b_\delta} \to \alpha^{b_\alpha}/\delta^{b_\delta})^b], \sigma \vdash \boldsymbol{M} : \sigma, \beta}{A, \alpha \vdash \boldsymbol{\xi k.\, M} : \tau, \beta} \; (\mathit{shift})$$

constraints for (shift): $\{b \le b_\tau, b \le b_\delta, b \le b_\alpha\}$

$$\frac{A, \sigma \vdash \boldsymbol{M} : \sigma, \tau}{A, \alpha \vdash \langle \boldsymbol{M} \rangle : \tau, \alpha} \; (\mathit{reset})$$

The binding-time analysis goes as follows. The input term is typed according to the above type inference rules, collecting all the constraints. We use additional constraints saying that the top-level binding times must be all dynamic. In other words, we start the inference of a term $\boldsymbol{M}$ from the following rule:

$$A_\phi, \alpha^d \vdash \boldsymbol{M} : \tau^d, \beta^d$$

Then, all the constraints are solved by repeatedly propagating d through the constraints. Namely, if we have $b_1 \le b_2$ and $b_1 = d$, we set $b_2 = d$. Notice that setting $b_2 = d$ does not violate other constraints. When this process is finished, all the remaining binding times that are not set to d are set to s. It gives the least (most static) solution of the constraints.

When we reach the solution, we convert the obtained annotated type τ into the type $||\tau||$ used in Section 7 as follows:

$$\begin{array}{rcl} ||o^s|| & = & s \\ ||o^d|| & = & d \\ ||(\sigma/\alpha \to \tau/\beta)^s|| & = & ||\sigma||/||\alpha|| \to ||\tau||/||\beta|| \\ ||(\sigma/\alpha \to \tau/\beta)^d|| & = & d \end{array}$$

With this conversion, we can choose static rules or dynamic rules to obtain annotations. Since the constraints on the function type ensure that there is no static information in $(\sigma/\alpha \to \tau/\beta)^d$, we can safely regard it as d without violating the rules for well-annotatedness.

Since the number of constraints generated is proportional to the length of the program, the constraints can be solved using Henglein's algorithm in almost linear time [10].

11. EXTENSION

The offline partial evaluator presented so far can handle only terminating programs because it is typed and does not have any ways to express general recursion. It is, however, easy to extend the framework to more realistic languages. In this section, we extend the input language to support constants (numbers), primitive operations (addition), a fixed point operator, and a conditional expression. The syntax for the input term is extended as follows:

$$\boldsymbol{M} = \ldots \mid \boldsymbol{n} \mid \boldsymbol{M{+}M} \mid \mathbf{fix}\, \boldsymbol{f(x).\, M} \mid \mathbf{if}\, \boldsymbol{M\; M\; M}$$

The annotated terms are extended accordingly by adding static (overline) and dynamic (underline) versions. The typing rules for annotated terms are:

$$A, \alpha \vdash \boldsymbol{n} : \mathsf{int}, \alpha \; [\overline{\boldsymbol{n}}] \quad (\mathit{num\text{-}s})$$

$$A, \alpha \vdash \boldsymbol{n} : d, \alpha \; [\underline{\boldsymbol{n}}] \quad (\mathit{num\text{-}d})$$

$$\frac{A, \delta \vdash \boldsymbol{M_1} : \mathsf{int}, \beta \; [\boldsymbol{W_1}] \quad A, \alpha \vdash \boldsymbol{M_2} : \mathsf{int}, \delta \; [\boldsymbol{W_2}]}{A, \alpha \vdash \boldsymbol{M_1{+}M_2} : \mathsf{int}, \beta \; [\boldsymbol{W_1} \overline{+} \boldsymbol{W_2}]} \; (\mathit{add\text{-}s})$$

$$\frac{A, \delta \vdash \boldsymbol{M_1} : d, \beta \; [\boldsymbol{W_1}] \quad A, \alpha \vdash \boldsymbol{M_2} : d, \delta \; [\boldsymbol{W_2}]}{A, \alpha \vdash \boldsymbol{M_1{+}M_2} : d, \beta \; [\boldsymbol{W_1} \underline{+} \boldsymbol{W_2}]} \; (\mathit{add\text{-}d})$$

$$\frac{A[\boldsymbol{f} : \sigma/\alpha \to \tau/\beta, \boldsymbol{x} : \sigma], \alpha \vdash \boldsymbol{M} : \tau, \beta \; [\boldsymbol{W}]}{A, \delta \vdash \mathbf{fix}\, \boldsymbol{f(x).\, M} : \sigma/\alpha \to \tau/\beta, \delta \; [\overline{\mathbf{fix}}\, \boldsymbol{f(x).\, W}]} \; (\mathit{fix\text{-}s})$$

$$\frac{A[\boldsymbol{f} : d, \boldsymbol{x} : d], d \vdash \boldsymbol{M} : d, d \; [\boldsymbol{W}]}{A, \delta \vdash \mathbf{fix}\, \boldsymbol{f(x).\, M} : d, \delta \; [\underline{\mathbf{fix}}\, \boldsymbol{f(x).\, W}]} \; (\mathit{fix\text{-}d})$$

$$\frac{\begin{array}{l} A, \delta \vdash \boldsymbol{M_1} : \mathsf{bool}, \beta \; [\boldsymbol{W_1}] \\ A, \alpha \vdash \boldsymbol{M_2} : \tau, \delta \; [\boldsymbol{W_2}] \quad A, \alpha \vdash \boldsymbol{M_3} : \tau, \delta \; [\boldsymbol{W_3}] \end{array}}{A, \alpha \vdash \mathbf{if}\, \boldsymbol{M_1\; M_2\; M_3} : \tau, \beta \; [\overline{\mathbf{if}}\, \boldsymbol{W_1\; W_2\; W_3}]} \; (\mathit{if\text{-}s})$$

$$\frac{\begin{array}{l} A, \delta \vdash \boldsymbol{M_1} : d, \beta \; [\boldsymbol{W_1}] \\ A, \alpha \vdash \boldsymbol{M_2} : \tau, \delta \; [\boldsymbol{W_2}] \quad A, \alpha \vdash \boldsymbol{M_3} : \tau, \delta \; [\boldsymbol{W_3}] \end{array}}{A, \alpha \vdash \mathbf{if}\, \boldsymbol{M_1\; M_2\; M_3} : \tau, \beta \; [\underline{\mathbf{if}}\, \boldsymbol{W_1\; W_2\; W_3}]} \; (\mathit{if\text{-}d})$$

These rules are a straightforward extension of the rules for the core language. We only note that they explicitly embed the order of execution. Arguments to primitive operators are evaluated from left to right. A predicate expression of a conditional is evaluated before then and else branches.

The specializer becomes:

$$\begin{aligned}
\mathcal{P}[\![\overline{\boldsymbol{n}}]\!]\,\rho\,\kappa &= \kappa\,(\mathcal{K}[\![\boldsymbol{n}]\!])\\
\mathcal{P}[\![\underline{\boldsymbol{n}}]\!]\,\rho\,\kappa &= \kappa\,(\boldsymbol{n})\\
\mathcal{P}[\![\boldsymbol{M}\overline{+}\boldsymbol{N}]\!]\,\rho\,\kappa &= \mathcal{P}[\![\boldsymbol{M}]\!]\,\rho\,\lambda m.\,\mathcal{P}[\![\boldsymbol{N}]\!]\,\rho\,\lambda n.\,\kappa\,(m+n)\\
\mathcal{P}[\![\boldsymbol{M}\underline{+}\boldsymbol{N}]\!]\,\rho\,\kappa &= \mathcal{P}[\![\boldsymbol{M}]\!]\,\rho\,\lambda m.\,\mathcal{P}[\![\boldsymbol{N}]\!]\,\rho\,\lambda n.\,\kappa\,(m\boldsymbol{+}n)\\
\mathcal{P}[\![\overline{\mathbf{fix}}\,\boldsymbol{f}(\boldsymbol{x}).\,\boldsymbol{M}]\!]\,\rho\,\kappa &= \kappa\,(\mathrm{fix}\,f(x).\,\lambda k.\,\mathcal{P}[\![\boldsymbol{M}]\!]\,\rho[x/\boldsymbol{x}, f/\boldsymbol{f}]\,k)
\end{aligned}$$

$$\begin{aligned}
&\mathcal{P}[\![\underline{\mathbf{fix}}\,\boldsymbol{f}(\boldsymbol{x}).\,\boldsymbol{M}]\!]\,\rho\,\kappa\\
&\quad = \kappa\,(\mathbf{fix}\,\boldsymbol{f}^{\diamond}(\boldsymbol{x}^{\diamond}).\,\boldsymbol{\lambda}\boldsymbol{k}^{\diamond}.\,\mathcal{P}[\![\boldsymbol{M}]\!]\,\rho[\boldsymbol{x}^{\diamond}/\boldsymbol{x}, \boldsymbol{f}^{\diamond}/\boldsymbol{f}]\,\lambda v.\,\boldsymbol{k}^{\diamond}\boldsymbol{@}v)\\
&\mathcal{P}[\![\overline{\mathbf{if}}\,\boldsymbol{P}\,\boldsymbol{M}\,\boldsymbol{N}]\!]\,\rho\,\kappa\\
&\quad = \mathcal{P}[\![\boldsymbol{P}]\!]\,\rho\,\lambda p.\,\mathrm{if}\ p\ \mathrm{then}\ \mathcal{P}[\![\boldsymbol{M}]\!]\,\rho\,\kappa\ \mathrm{else}\ \mathcal{P}[\![\boldsymbol{N}]\!]\,\rho\,\kappa\\
&\mathcal{P}[\![\underline{\mathbf{if}}\,\boldsymbol{P}\,\boldsymbol{M}\,\boldsymbol{N}]\!]\,\rho\,\kappa\\
&\quad = \mathcal{P}[\![\boldsymbol{P}]\!]\,\rho\,\lambda p.\,\mathbf{if}\,p\,(\mathcal{P}[\![\boldsymbol{M}]\!]\,\rho\,\kappa)\,(\mathcal{P}[\![\boldsymbol{N}]\!]\,\rho\,\kappa)
\end{aligned}$$

where $\mathcal{K}$ is used to interpret constants (*i.e.*, $\mathcal{K}[\![\boldsymbol{n}]\!] = n$). The specializer is also a simple extension to the one for the core language. We realize recursion and conditionals in the object language by the recursion and conditionals in the metalanguage, respectively.

In the rule for dynamic conditionals, we duplicate the continuation κ in the then and else branches. It leads to better specialization, but it can also cause duplication of code. To avoid the duplication (with the price of possibly less specialization), we could alternatively write it as follows:

$$\begin{aligned}
&\mathcal{P}[\![\underline{\mathbf{if}}\,\boldsymbol{P}\,\boldsymbol{M}\,\boldsymbol{N}]\!]\,\rho\,\kappa\\
=\ &\mathcal{P}[\![\boldsymbol{P}]\!]\,\rho\,\lambda p.\,\mathbf{let}\ \boldsymbol{k}^{\diamond} = \boldsymbol{\lambda}\boldsymbol{t}^{\diamond}.\,\kappa\,(\boldsymbol{t}^{\diamond})\ \mathbf{in}\\
&\qquad\qquad\quad \mathbf{if}\,p\,(\mathcal{P}[\![\boldsymbol{M}]\!]\,\rho\,\lambda v.\,\boldsymbol{k}^{\diamond}\boldsymbol{@}v)\,(\mathcal{P}[\![\boldsymbol{N}]\!]\,\rho\,\lambda v.\,\boldsymbol{k}^{\diamond}\boldsymbol{@}v)
\end{aligned}$$

$\mathbf{let}\ \boldsymbol{t} = M\ \mathbf{in}\ N$ is an abbreviation for $(\boldsymbol{\lambda}\boldsymbol{t}.\,N)\boldsymbol{@}M$ representing a dynamic let expression. We used two η-expansions here: first to turn a static continuation κ into a dynamic one $\boldsymbol{k}^{\diamond}$, and then to turn it back to a static one $\lambda v.\,\boldsymbol{k}^{\diamond}\boldsymbol{@}v$. Since κ is now residualized once in a let expression, it is never duplicated. During specialization of $\boldsymbol{M}$ and $\boldsymbol{N}$, a static continuation $\lambda v.\,\boldsymbol{k}^{\diamond}\boldsymbol{@}v$ is used, which does not pass the result to κ directly, but defers it until runtime. Since the specialization results for $\boldsymbol{M}$ and $\boldsymbol{N}$ are now residualized, they have to be classified as dynamic. Thus, we need to use the following typing rule in which three occurrences of τ in (*if-d*) is replaced with d:

$$\frac{\begin{array}{l}A, \delta \vdash \boldsymbol{M_1} : d, \beta\ [\boldsymbol{W_1}]\\ A, \alpha \vdash \boldsymbol{M_2} : d, \delta\ [\boldsymbol{W_2}] \quad A, \alpha \vdash \boldsymbol{M_3} : d, \delta\ [\boldsymbol{W_3}]\end{array}}{A, \alpha \vdash \mathbf{if}\,\boldsymbol{M_1}\,\boldsymbol{M_2}\,\boldsymbol{M_3} : d, \beta\ [\underline{\mathbf{if}}\,\boldsymbol{W_1}\,\boldsymbol{W_2}\,\boldsymbol{W_3}]}\ (\textit{if-d}')$$

12. EXAMPLES

We are now ready to specialize the examples shown earlier. For the append program, the binding-time analysis produces the following annotation, assuming `x` is static. We use `_` to indicate dynamic expressions.

```
(define (append x)
  (letrec ((app (lambda (x)
                  (if (null? x)
                    (_shift k k)
                    (_cons (car x)
                           (app (cdr x)))))))
    (reset (app x))))
```

Since `x` is static, `(null? x)`, `(car x)`, and `(cdr x)` all become static and we can unfold recursion completely. `(shift k k)` is classified as dynamic because the captured continuation has to be residualized as a specialized output term. This in turn makes the return value of `app` dynamic and hence `cons` is classified as dynamic. This classification is similar to the classification of the ordinary append program where `x` is static and `y` is dynamic:

```
(define (append x y)
  (if (null? x)
    y
    (_cons (car x) (append (cdr x) y))))
```

When we partially evaluate the above program with respect to a static list `'(1 2 3)`, we obtain the following result:

```
> (pe '((lambda (x)
          (letrec ((app (lambda (x)
                          (if (null? x)
                            (_shift k k)
                            (_cons (car x)
                                (app (cdr x)))))))
            (reset (app x))))
        '(1 2 3))
      empty-env
      init-cont)
(lambda (v40) (lambda (k41)
  (k41 (cons 1 (cons 2 (cons 3 v40))))))
```

Although the result is in CPS, we can see that the static parts are completely reduced. The resulting program simply conses 1, 2, and 3 to the argument and returns the appended list to its continuation.

For the reverse program, the binding-time analysis produces the following annotation, assuming that the input list is a static list of dynamic elements.

```
(define (reverse lst)
  (letrec ((rev (lambda (lst)
                  (if (null? lst)
                    (_quote ())
                    (shift k
                      (_cons (car lst)
                        (k (rev (cdr lst)))))))))
    (reset (rev lst))))
```

Since the list structure itself is static, we can safely unfold recursion at specialization time. Furthermore, capturing and invoking of continuations are also classified as static. Together, reversing is performed completely at specialization time. For example, if we partially evaluate this program with respect to a list `(x 2 3)` where `x` is unknown, we obtain the following:

```
> (pe '(_lambda (x) (reverse (list x 2 3)))
      empty-env
      init-cont)
(lambda (x42) (lambda (k43)
  (k43 (cons 3 (cons 2 (cons x42 '()))))))
```

13. RELATED WORK

This work extends our earlier work [1] where we presented online partial evaluators for λ-terms with shift and reset. There, we showed the partial evaluator that produces output not only in CPS but also in direct style. We have not investigated the offline partial evaluator whose output is in direct style.

Thiemann [14] presented an offline partial evaluator for Scheme including call/cc. Based on a standard type system, he reduces call/cc when the body of call/cc is completely static.

The correctness proof for offline partial evaluators using logical relations appears in Jones et al. [11, Chapter 8]. It has also been used by Wand [17] to prove the correctness of an offline partial evaluator for pure λ-calculus. The present work follows this approach and shows that the technique is applicable to specializers written in CPS.

The CPS transformer and the partial evaluator presented in this paper are based on the higher-order one-pass CPS transformation [6]. They are written mostly in CPS but make use of non-tail calls. The same technique is used to obtain transformations to the A-normal forms [8] and monadic normal forms [3].

Danvy and Nielsen [7] presented another one-pass CPS transformation that is first-order. It would be interesting to see how the proof method would change if we adopted it.

14. CONCLUSION

This paper has presented an offline partial evaluator for the λ-calculus with the delimited continuation constructs *shift* and *reset*. Based on Danvy and Filinski's type system for shift and reset, we first presented a type system that specifies well-annotated terms. We then showed a specializer that receives an annotated term and produces the output in CPS. The correctness of our partial evaluator was established using the technique of logical relations. Thanks to the explicit reference to the type of continuations, we could establish the correctness using the standard structural induction, regardless of the fact that the specializer itself is written in CPS. The paper also showed an efficient constraint-based binding-time analysis as well as how to extend the present work to richer language constructs, such as recursion and conditionals.

We are now investigating how to construct an offline partial evaluator that produces the output in direct style, as well as its correctness proof.

Acknowledgments

I am grateful to Mitchell Wand who kindly accepted my visit to Northeastern University. Throughout this research, I received many valuable comments, criticisms, advice, suggestions, and encouragements. Without his support, this work could not have been completed. I would like to thank Olivier Danvy for his comments and support. Finally, the comments from reviewers helped improve the paper.

15. REFERENCES

[1] Asai, K. "Online Partial Evaluation for Shift and Reset," *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '02)*, pp. 19–30 (January 2002).

[2] Asai, K. "Reflecting on the Metalevel Interpreter Written in Direct Style," Presented at the International Lisp Conference 2003 (ILC 2003), New York City, 12 pages (October 2003).

[3] Danvy, O. "A New One-Pass Transformation into Monadic Normal Form," *Proceedings of the 12th International Conference on Compiler Construction (CC2003), LNCS 2622*, pp. 77–89 (April 2003).

[4] Danvy, O., and A. Filinski "A Functional Abstraction of Typed Contexts," Technical Report 89/12, DIKU, University of Copenhagen (July 1989).

[5] Danvy, O., and A. Filinski "Abstracting Control," *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160 (June 1990).

[6] Danvy, O., and A. Filinski "Representing Control, a Study of the CPS Transformation," *Mathematical Structures in Computer Science*, Vol. 2, No. 4, pp. 361–391 (December 1992).

[7] Danvy, O., and L. R. Nielsen "A First-Order One-Pass CPS Transformation," *Theoretical Computer Science*, Vol. 308, Nos. 1–3, pp. 239–257 (November 2003).

[8] Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen "The Essence of Compiling with Continuations," *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, pp. 237–247 (June 1993).

[9] Gasbichler, M., and M. Sperber "Final Shift for Call/cc: Direct Implementation of Shift and Reset," *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, pp. 271–282 (October 2002).

[10] Henglein, F. "Efficient Type Inference for Higher-Order Binding-Time Analysis," In J. Hughes, editor, *Functional Programming Languages and Computer Architecture (LNCS 523)*, pp. 448–472 (August 1991).

[11] Jones, N. D., C. K. Gomard, and P. Sestoft *Partial Evaluation and Automatic Program Generation*, New York: Prentice-Hall (1993).

[12] Kelsey, R., W. Clinger, and J. Rees (editors) "Revised^5 Report on the Algorithmic Language Scheme," *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, pp. 7–105, Kluwer Academic Publishers (August 1998).

[13] Lawall, J. L., and O. Danvy "Continuation-Based Partial Evaluation," *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pp. 227–238 (June 1994).

[14] Thiemann, P. "Towards Partial Evaluation of Full Scheme," *Proceedings of Reflection'96*, pp. 105–115 (April 1996).

[15] Thiemann, P. J. "Cogen in Six Lines," *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pp. 180–189 (May 1996).

[16] Thiemann, P. "Combinators for Program Generation," *Journal of Functional Programming*, Vol. 9, No. 5, pp. 483–525, Cambridge University Press (September 1999).

[17] Wand, M. "Specifying the Correctness of Binding-Time Analysis," *Journal of Functional Programming*, Vol. 3, No. 3, pp. 365–387, Cambridge University Press (July 1993).

APPENDIX

A. PROOF

LEMMA 1 (ADMISSIBILITY). *If $M =_\beta M'$ and $N =_\beta N'$, then $(M, N) \in R_\tau \Longleftrightarrow (M', N') \in R_\tau$.*

Proof. By induction on the length $|\tau|$ of τ, defined by $|s| = 0$, $|d| = 0$, and $|\sigma \to \tau| = |\tau| + 1$. ☐

Theorem 2. *If* $A, \alpha \vdash \boldsymbol{M} : \tau, \beta\ [\boldsymbol{W}]$, $(\rho, \rho') \models A$, *and* $(\kappa, \kappa') \in R_{\tau^* \to \alpha^*}$, *then* $(\mathcal{P}\llbracket \boldsymbol{W} \rrbracket \rho\, \kappa, \mathcal{I}\llbracket \boldsymbol{M} \rrbracket \rho'\, \kappa') \in R_{\beta^*}$.

Proof. By induction on the structure of the proof of $A, \alpha \vdash \boldsymbol{M} : \tau, \beta\ [\boldsymbol{W}]$.

Case 1 (Variable). Assume

$$A[\boldsymbol{x} : \tau], \alpha \vdash \boldsymbol{x} : \tau, \alpha\ [\boldsymbol{x}] \quad (\mathit{var})$$

We need to show $(\mathcal{P}\llbracket \boldsymbol{x} \rrbracket \rho\, \kappa, \mathcal{I}\llbracket \boldsymbol{x} \rrbracket \rho'\, \kappa') \in R_{\alpha^*}$ assuming $(\rho, \rho') \models A[\boldsymbol{x} : \tau]$ and $(\kappa, \kappa') \in R_{\tau^* \to \alpha^*}$. From the definition of $\mathcal{P}$ and $\mathcal{I}$ and the Admissibility Lemma, we have:

$$\begin{array}{rl} & (\mathcal{P}\llbracket \boldsymbol{x} \rrbracket \rho\, \kappa, \mathcal{I}\llbracket \boldsymbol{x} \rrbracket \rho'\, \kappa') \in R_{\alpha^*} \\ \iff & (\kappa\, (\rho\, (\boldsymbol{x})), \kappa'\, (\rho'\, (\boldsymbol{x}))) \in R_{\alpha^*} \end{array}$$

which holds because $(\rho, \rho') \models A[\boldsymbol{x} : \tau]$ and $(\kappa, \kappa') \in R_{\tau^* \to \alpha^*}$.

Case 2 (Static Abstraction). Assume

$$\frac{A[\boldsymbol{x} : \sigma], \alpha \vdash \boldsymbol{M} : \tau, \beta\ [\boldsymbol{W}]}{A, \delta \vdash \boldsymbol{\lambda x.\, M} : \sigma/\alpha \to \tau/\beta, \delta\ [\overline{\boldsymbol{\lambda}}\boldsymbol{x.\, W}]}\ (\mathit{abs\text{-}s})$$

We need to show $(\mathcal{P}\llbracket \overline{\boldsymbol{\lambda}}\boldsymbol{x.\, W} \rrbracket \rho\, \kappa, \mathcal{I}\llbracket \boldsymbol{\lambda x.\, M} \rrbracket \rho'\, \kappa') \in R_{\delta^*}$ assuming $(\rho, \rho') \models A$ and $(\kappa, \kappa') \in R_{(\sigma/\alpha \to \tau/\beta)^* \to \delta^*}$ (or equivalently $(\kappa, \kappa') \in R_{(\sigma^* \to (\tau^* \to \alpha^*) \to \beta^*) \to \delta^*}$). From the definition of $\mathcal{P}$ and $\mathcal{I}$, we have:

$$\begin{array}{rl} & (\mathcal{P}\llbracket \overline{\boldsymbol{\lambda}}\boldsymbol{x.\, W} \rrbracket \rho\, \kappa, \mathcal{I}\llbracket \boldsymbol{\lambda x.\, M} \rrbracket \rho'\, \kappa') \in R_{\delta^*} \\ \iff & (\kappa\, (\lambda x.\, \lambda k.\, \mathcal{P}\llbracket \boldsymbol{W} \rrbracket \rho[x/\boldsymbol{x}]\, k), \\ & \ \kappa'\, (\lambda x.\, \lambda k.\, \mathcal{I}\llbracket \boldsymbol{M} \rrbracket \rho'[x/\boldsymbol{x}]\, k)) \in R_{\delta^*} \end{array}$$

Since $(\kappa, \kappa') \in R_{(\sigma^* \to (\tau^* \to \alpha^*) \to \beta^*) \to \delta^*}$, it suffices to show that

$$\begin{array}{l} (\lambda x.\, \lambda k.\, \mathcal{P}\llbracket \boldsymbol{W} \rrbracket \rho[x/\boldsymbol{x}]\, k, \\ \ \lambda x.\, \lambda k.\, \mathcal{I}\llbracket \boldsymbol{M} \rrbracket \rho'[x/\boldsymbol{x}]\, k) \in R_{\sigma^* \to (\tau^* \to \alpha^*) \to \beta^*}\quad . \end{array}$$

So assume that $(W_1, M_1) \in R_{\sigma^*}$ and $(K, L) \in R_{\tau^* \to \alpha^*}$. We want to show

$$\begin{array}{l} ((\lambda x.\, \lambda k.\, \mathcal{P}\llbracket \boldsymbol{W} \rrbracket \rho[x/\boldsymbol{x}]\, k)\, W_1\, K, \\ \ (\lambda x.\, \lambda k.\, \mathcal{I}\llbracket \boldsymbol{M} \rrbracket \rho'[x/\boldsymbol{x}]\, k)\, M_1\, L) \in R_{\beta^*} \end{array}$$

which β-reduces to

$$(\mathcal{P}\llbracket \boldsymbol{W} \rrbracket \rho[W_1/\boldsymbol{x}]\, K, \mathcal{I}\llbracket \boldsymbol{M} \rrbracket \rho'[M_1/\boldsymbol{x}]\, L) \in R_{\beta^*}\quad .$$

This statement holds by the induction hypothesis, because $(\rho[W_1/\boldsymbol{x}], \rho'[M_1/\boldsymbol{x}]) \models A[\boldsymbol{x} : \sigma]$ and $(K, L) \in R_{\tau^* \to \alpha^*}$.

Case 3 (Dynamic Abstraction). Assume

$$\frac{A[\boldsymbol{x} : d], d \vdash \boldsymbol{M} : d, d\ [\boldsymbol{W}]}{A, \delta \vdash \boldsymbol{\lambda x.\, M} : d, \delta\ [\underline{\boldsymbol{\lambda}}\boldsymbol{x.\, W}]}\ (\mathit{abs\text{-}d})$$

We need to show $(\mathcal{P}\llbracket \underline{\boldsymbol{\lambda}}\boldsymbol{x.\, W} \rrbracket \rho\, \kappa, \mathcal{I}\llbracket \boldsymbol{\lambda x.\, M} \rrbracket \rho'\, \kappa') \in R_{\delta^*}$ assuming $(\rho, \rho') \models A$ and $(\kappa, \kappa') \in R_{d^* \to \delta^*}$. From the definition of $\mathcal{P}$ and $\mathcal{I}$, we have:

$$\begin{array}{rl} & (\mathcal{P}\llbracket \underline{\boldsymbol{\lambda}}\boldsymbol{x.\, W} \rrbracket \rho\, \kappa, \mathcal{I}\llbracket \boldsymbol{\lambda x.\, M} \rrbracket \rho'\, \kappa') \in R_{\delta^*} \\ \iff & (\kappa\, (\boldsymbol{\lambda x^{\diamond}.\, \lambda k^{\diamond}.}\, \mathcal{P}\llbracket \boldsymbol{W} \rrbracket \rho[\boldsymbol{x}^{\diamond}/\boldsymbol{x}]\, \lambda v.\, \boldsymbol{k}^{\diamond}\boldsymbol{@}v), \\ & \ \kappa'\, (\lambda x^{\diamond}.\, \lambda k^{\diamond}.\, \mathcal{I}\llbracket \boldsymbol{M} \rrbracket \rho'[x^{\diamond}/\boldsymbol{x}]\, k^{\diamond})) \in R_{\delta^*} \end{array}$$

Since $(\kappa, \kappa') \in R_{d^* \to \delta^*}$ and $d^* = d$, it suffices to show that

$$\begin{array}{l} (\boldsymbol{\lambda x^{\diamond}.\, \lambda k^{\diamond}.}\, \mathcal{P}\llbracket \boldsymbol{W} \rrbracket \rho[\boldsymbol{x}^{\diamond}/\boldsymbol{x}]\, \lambda v.\, \boldsymbol{k}^{\diamond}\boldsymbol{@}v, \\ \ \lambda x^{\diamond}.\, \lambda k^{\diamond}.\, \mathcal{I}\llbracket \boldsymbol{M} \rrbracket \rho'[x^{\diamond}/\boldsymbol{x}]\, k^{\diamond}) \in R_d\quad . \end{array}$$

Now, $(\boldsymbol{x}^{\diamond}, x^{\diamond}) \in R_d$ holds, so let $\rho_1 = \rho[\boldsymbol{x}^{\diamond}/\boldsymbol{x}]$ and $\rho'_1 = \rho'[x^{\diamond}/\boldsymbol{x}]$; then $(\rho_1, \rho'_1) \models A[\boldsymbol{x} : d]$. Furthermore, let $\kappa_1 = \lambda v.\, \boldsymbol{k}^{\diamond}\boldsymbol{@}v$ and $\kappa'_1 = k^{\diamond}$; then we have $(\kappa_1, \kappa'_1) \in R_{d \to d}$ (since for any $(a, a') \in R_d$, we have

$$\begin{array}{lll} & \mathcal{E}\llbracket \kappa_1\, (a) \rrbracket \mu_{id} & \\ =_\beta & \mathcal{E}\llbracket \boldsymbol{k}^{\diamond}\boldsymbol{@}a \rrbracket \mu_{id} & \text{definition of } \kappa_1 \\ = & (\mathcal{E}\llbracket \boldsymbol{k}^{\diamond} \rrbracket \mu_{id})\, (\mathcal{E}\llbracket a \rrbracket \mu_{id}) & \text{definition of } \mathcal{E} \\ = & k^{\diamond}\, (\mathcal{E}\llbracket a \rrbracket \mu_{id}) & \text{definition of } \mathcal{E} \\ =_\beta & k^{\diamond}\, (a') & (a, a') \in R_d \\ = & \kappa'_1\, (a') & \text{definition of } \kappa'_1 \end{array}$$

and thus $(\kappa_1\, (a), \kappa'_1\, (a')) \in R_d$). Therefore, by the induction hypothesis, we know that

$$(\mathcal{P}\llbracket \boldsymbol{W} \rrbracket \rho_1\, \kappa_1, \mathcal{I}\llbracket \boldsymbol{M} \rrbracket \rho'_1\, \kappa'_1) \in R_d$$

that is,

$$\mathcal{E}\llbracket \mathcal{P}\llbracket \boldsymbol{W} \rrbracket \rho_1\, \kappa_1 \rrbracket \mu_{id} =_\beta \mathcal{I}\llbracket \boldsymbol{M} \rrbracket \rho'_1\, \kappa'_1 \quad (1)$$

Now, to establish

$$\begin{array}{l} (\boldsymbol{\lambda x^{\diamond}.\, \lambda k^{\diamond}.}\, \mathcal{P}\llbracket \boldsymbol{W} \rrbracket \rho[\boldsymbol{x}^{\diamond}/\boldsymbol{x}]\, \lambda v.\, \boldsymbol{k}^{\diamond}\boldsymbol{@}v, \\ \ \lambda x^{\diamond}.\, \lambda k^{\diamond}.\, \mathcal{I}\llbracket \boldsymbol{M} \rrbracket \rho'[x^{\diamond}/\boldsymbol{x}]\, k^{\diamond}) \in R_d \end{array}$$

we calculate:

$$\begin{array}{lll} & \mathcal{E}\llbracket \boldsymbol{\lambda x^{\diamond}.\, \lambda k^{\diamond}.}\, \mathcal{P}\llbracket \boldsymbol{W} \rrbracket \rho[\boldsymbol{x}^{\diamond}/\boldsymbol{x}]\, \lambda v.\, \boldsymbol{k}^{\diamond}\boldsymbol{@}v \rrbracket \mu_{id} & \\ = & \lambda x^{\diamond}.\, \lambda k^{\diamond}.\, \mathcal{E}\llbracket \mathcal{P}\llbracket \boldsymbol{W} \rrbracket \rho[\boldsymbol{x}^{\diamond}/\boldsymbol{x}]\, \lambda v.\, \boldsymbol{k}^{\diamond}\boldsymbol{@}v \rrbracket \mu_{id}[x^{\diamond}/\boldsymbol{x}^{\diamond}, k^{\diamond}/\boldsymbol{k}^{\diamond}] & \\ & & \text{definition of } \mathcal{E} \text{ twice} \\ = & \lambda x^{\diamond}.\, \lambda k^{\diamond}.\, \mathcal{E}\llbracket \mathcal{P}\llbracket \boldsymbol{W} \rrbracket \rho_1\, \kappa_1 \rrbracket \mu_{id} & \text{definition of } \rho_1,\ \kappa_1, \text{ and } \mu_{id} \\ =_\beta & \lambda x^{\diamond}.\, \lambda k^{\diamond}.\, \mathcal{I}\llbracket \boldsymbol{M} \rrbracket \rho'_1\, \kappa'_1 & \text{Equation (1)} \\ = & \lambda x^{\diamond}.\, \lambda k^{\diamond}.\, \mathcal{I}\llbracket \boldsymbol{M} \rrbracket \rho'[x^{\diamond}/\boldsymbol{x}]\, k^{\diamond} & \text{definition of } \rho'_1 \text{ and } \kappa'_1 \end{array}$$

Case 4 (Static Application). Assume

$$\frac{\begin{array}{l} A, \delta \vdash \boldsymbol{M_1} : \sigma/\alpha \to \tau/\epsilon, \beta\ [\boldsymbol{W_1}] \\ A, \epsilon \vdash \boldsymbol{M_2} : \sigma, \delta\ [\boldsymbol{W_2}] \end{array}}{A, \alpha \vdash \boldsymbol{M_1 @ M_2} : \tau, \beta\ [\boldsymbol{W_1}\overline{\boldsymbol{@}}\boldsymbol{W_2}]}\ (\mathit{app\text{-}s})$$

We must show $(\mathcal{P}\llbracket \boldsymbol{W_1}\overline{\boldsymbol{@}}\boldsymbol{W_2} \rrbracket \rho\, \kappa, \mathcal{I}\llbracket \boldsymbol{M_1 @ M_2} \rrbracket \rho'\, \kappa') \in R_{\beta^*}$ assuming $(\rho, \rho') \models A$ and $(\kappa, \kappa') \in R_{\tau^* \to \alpha^*}$. From the definition of $\mathcal{P}$ and $\mathcal{I}$, we have:

$$\begin{array}{rl} & (\mathcal{P}\llbracket \boldsymbol{W_1}\overline{\boldsymbol{@}}\boldsymbol{W_2} \rrbracket \rho\, \kappa, \mathcal{I}\llbracket \boldsymbol{M_1 @ M_2} \rrbracket \rho'\, \kappa') \in R_{\beta^*} \\ \iff & (\mathcal{P}\llbracket \boldsymbol{W_1} \rrbracket \rho\, \lambda w_1.\, \mathcal{P}\llbracket \boldsymbol{W_2} \rrbracket \rho\, \lambda w_2.\, w_1\, w_2\, \kappa, \\ & \ \mathcal{I}\llbracket \boldsymbol{M_1} \rrbracket \rho'\, \lambda m_1.\, \mathcal{I}\llbracket \boldsymbol{M_2} \rrbracket \rho'\, \lambda m_2.\, m_1\, m_2\, \kappa') \in R_{\beta^*} \end{array}$$

Since $(\rho, \rho') \models A$, we are done by the induction hypothesis if we can prove

$$\begin{array}{l} (\lambda w_1.\, \mathcal{P}\llbracket \boldsymbol{W_2} \rrbracket \rho\, \lambda w_2.\, w_1\, w_2\, \kappa, \\ \ \lambda m_1.\, \mathcal{I}\llbracket \boldsymbol{M_2} \rrbracket \rho'\, \lambda m_2.\, m_1\, m_2\, \kappa') \in R_{(\sigma/\alpha \to \tau/\epsilon)^* \to \delta^*}\quad . \end{array}$$

So take any $(w'_1, m'_1) \in R_{(\sigma/\alpha \to \tau/\epsilon)^*}$. We then need to show

$$(\mathcal{P}\llbracket \boldsymbol{W_2} \rrbracket \rho\, \lambda w_2.\, w'_1\, w_2\, \kappa, \mathcal{I}\llbracket \boldsymbol{M_2} \rrbracket \rho'\, \lambda m_2.\, m'_1\, m_2\, \kappa') \in R_{\delta^*}\quad .$$

Since $(\rho, \rho') \models A$, we are again done by the induction hypothesis if we can prove

$$(\lambda w_2.\, w'_1\, w_2\, \kappa, \lambda m_2.\, m'_1\, m_2\, \kappa') \in R_{\sigma^* \to \epsilon^*}\quad .$$

So take any $(w'_2, m'_2) \in R_{\sigma^*}$. We then need to show

$$(w'_1\, w'_2\, \kappa, m'_1\, m'_2\, \kappa') \in R_{\epsilon^*}$$

which holds because $(\sigma/\alpha \to \tau/\epsilon)^* = \sigma^* \to (\tau^* \to \alpha^*) \to \epsilon^*$ and $(\kappa, \kappa') \in R_{\tau^* \to \alpha^*}$.

Case 5 (Dynamic Application). Assume

$$\frac{A, \delta \vdash \boldsymbol{M_1} : d, \beta\ [\boldsymbol{W_1}] \quad A, d \vdash \boldsymbol{M_2} : d, \delta\ [\boldsymbol{W_2}]}{A, d \vdash \boldsymbol{M_1 @ M_2} : d, \beta\ [\boldsymbol{W_1}\underline{\mathbf{@}}\boldsymbol{W_2}]}\ (\textit{app-d})$$

We need to show $(\mathcal{P}\llbracket \boldsymbol{W_1}\underline{\mathbf{@}}\boldsymbol{W_2}\rrbracket\,\rho\,\kappa, \mathcal{I}\llbracket \boldsymbol{M_1 @ M_2}\rrbracket\,\rho'\,\kappa') \in R_{\beta^*}$ assuming $(\rho, \rho') \models A$ and $(\kappa, \kappa') \in R_{d^* \to d^*}$. From the definition of $\mathcal{P}$ and $\mathcal{I}$, we have:

$$\begin{array}{ll} & (\mathcal{P}\llbracket \boldsymbol{W_1}\underline{\mathbf{@}}\boldsymbol{W_2}\rrbracket\,\rho\,\kappa, \mathcal{I}\llbracket \boldsymbol{M_1 @ M_2}\rrbracket\,\rho'\,\kappa') \in R_{\beta^*} \\ \iff & (\mathcal{P}\llbracket \boldsymbol{W_1}\rrbracket\,\rho\,\lambda w_1.\, \mathcal{P}\llbracket \boldsymbol{W_2}\rrbracket\,\rho\,\lambda w_2.\, w_1 \mathbf{@} w_2 \mathbf{@} \boldsymbol{\lambda t^\diamond.}\, \kappa\,(\boldsymbol{t}^\diamond), \\ & \ \mathcal{I}\llbracket \boldsymbol{M_1}\rrbracket\,\rho'\,\lambda m_1.\, \mathcal{I}\llbracket \boldsymbol{M_2}\rrbracket\,\rho'\,\lambda m_2.\, m_1\, m_2\, \kappa') \in R_{\beta^*} \end{array}$$

Since $(\rho, \rho') \models A$, we are done by the induction hypothesis if we can prove

$$\begin{array}{l} (\lambda w_1.\, \mathcal{P}\llbracket \boldsymbol{W_2}\rrbracket\,\rho\,\lambda w_2.\, w_1 \mathbf{@} w_2 \mathbf{@} \boldsymbol{\lambda t^\diamond.}\, \kappa\,(\boldsymbol{t}^\diamond), \\ \ \lambda m_1.\, \mathcal{I}\llbracket \boldsymbol{M_2}\rrbracket\,\rho'\,\lambda m_2.\, m_1\, m_2\, \kappa') \in R_{d^* \to \delta^*} \quad . \end{array}$$

So take any $(w_1', m_1') \in R_{d^*}$. We then need to show

$$\begin{array}{l} (\mathcal{P}\llbracket \boldsymbol{W_2}\rrbracket\,\rho\,\lambda w_2.\, w_1' \mathbf{@} w_2 \mathbf{@} \boldsymbol{\lambda t^\diamond.}\, \kappa\,(\boldsymbol{t}^\diamond), \\ \ \mathcal{I}\llbracket \boldsymbol{M_2}\rrbracket\,\rho'\,\lambda m_2.\, m_1'\, m_2\, \kappa') \in R_{\delta^*} \quad . \end{array}$$

Since $(\rho, \rho') \models A$, we are again done by the induction hypothesis if we can prove

$$(\lambda w_2.\, w_1' \mathbf{@} w_2 \mathbf{@} \boldsymbol{\lambda t^\diamond.}\, \kappa\,(\boldsymbol{t}^\diamond), \lambda m_2.\, m_1'\, m_2\, \kappa') \in R_{d^* \to d^*} \quad .$$

So take any $(w_2', m_2') \in R_{d^*}$. We then need to show

$$(w_1' \mathbf{@} w_2' \mathbf{@} \boldsymbol{\lambda t^\diamond.}\, \kappa\,(\boldsymbol{t}^\diamond), m_1'\, m_2'\, \kappa') \in R_{d^*} \quad .$$

We calculate:

$$\begin{array}{lll} & \mathcal{E}\llbracket w_1' \mathbf{@} w_2' \mathbf{@} \boldsymbol{\lambda t^\diamond.}\, \kappa\,(\boldsymbol{t}^\diamond)\rrbracket\,\mu_{id} & \\ = & (\mathcal{E}\llbracket w_1'\rrbracket\,\mu_{id})\,(\mathcal{E}\llbracket w_2'\rrbracket\,\mu_{id})\,(\mathcal{E}\llbracket \boldsymbol{\lambda t^\diamond.}\, \kappa\,(\boldsymbol{t}^\diamond)\rrbracket\,\mu_{id}) & \\ & & \text{definition of } \mathcal{E} \text{ twice} \\ =_\beta & m_1'\,(\mathcal{E}\llbracket w_2'\rrbracket\,\mu_{id})\,(\mathcal{E}\llbracket \boldsymbol{\lambda t^\diamond.}\, \kappa\,(\boldsymbol{t}^\diamond)\rrbracket\,\mu_{id}) & (w_1', m_1') \in R_{d^*} \\ =_\beta & m_1'\, m_2'\,(\mathcal{E}\llbracket \boldsymbol{\lambda t^\diamond.}\, \kappa\,(\boldsymbol{t}^\diamond)\rrbracket\,\mu_{id}) & (w_2', m_2') \in R_{d^*} \\ = & m_1'\, m_2'\,(\lambda t^\diamond.\, \mathcal{E}\llbracket \kappa\,(\boldsymbol{t}^\diamond)\rrbracket\,\mu_{id}[t^\diamond/\boldsymbol{t}^\diamond]) & \text{definition of } \mathcal{E} \\ = & m_1'\, m_2'\,(\lambda t^\diamond.\, \mathcal{E}\llbracket \kappa\,(\boldsymbol{t}^\diamond)\rrbracket\,\mu_{id}) & \text{definition of } \mu_{id} \end{array}$$

Thus, we are done if we can prove $\lambda t^\diamond.\, \mathcal{E}\llbracket \kappa\,(\boldsymbol{t}^\diamond)\rrbracket\,\mu_{id} =_\beta \kappa'$.

Now, $(\boldsymbol{t}^\diamond, t^\diamond) \in R_d$ holds, so from $(\kappa, \kappa') \in R_{d \to d}$, we have $(\kappa\,(\boldsymbol{t}^\diamond), \kappa'\,(t^\diamond)) \in R_d$. The definition of R_d gives us $\mathcal{E}\llbracket \kappa\,(\boldsymbol{t}^\diamond)\rrbracket\,\mu_{id} =_\beta \kappa'\,(t^\diamond)$, and hence $\lambda t^\diamond.\, \mathcal{E}\llbracket \kappa\,(\boldsymbol{t}^\diamond)\rrbracket\,\mu_{id} =_\beta \lambda t^\diamond.\, \kappa'\,(t^\diamond)$. So we need to show $\lambda t^\diamond.\, \kappa'\,(t^\diamond) =_\beta \kappa'$, but it holds because simple investigation on $\mathcal{I}$ shows that κ' always has the form $\lambda x.\, \cdots$.

Case 6 (Static Shift). Assume

$$\frac{A[\boldsymbol{k} : \tau/\delta \to \alpha/\delta], \sigma \vdash \boldsymbol{M} : \sigma, \beta\ [\boldsymbol{W}]}{A, \alpha \vdash \boldsymbol{\xi k.\, M} : \tau, \beta\ [\overline{\boldsymbol{\xi}}\boldsymbol{k.\, W}]}\ (\textit{shift-s})$$

We need to show $(\mathcal{P}\llbracket \overline{\boldsymbol{\xi}}\boldsymbol{k.\, W}\rrbracket\,\rho\,\kappa, \mathcal{I}\llbracket \boldsymbol{\xi k.\, M}\rrbracket\,\rho'\,\kappa') \in R_{\beta^*}$ assuming $(\rho, \rho') \models A$ and $(\kappa, \kappa') \in R_{\tau^* \to \alpha^*}$. From the definition of $\mathcal{P}$ and $\mathcal{I}$, we have:

$$\begin{array}{ll} & (\mathcal{P}\llbracket \overline{\boldsymbol{\xi}}\boldsymbol{k.\, W}\rrbracket\,\rho\,\kappa, \mathcal{I}\llbracket \boldsymbol{\xi k.\, M}\rrbracket\,\rho'\,\kappa') \in R_{\beta^*} \\ \iff & (\mathcal{P}\llbracket \boldsymbol{W}\rrbracket\,\rho[\lambda v.\, \lambda k.\, k\,(\kappa\,(v))/\boldsymbol{k}]\,\lambda x.\, x, \\ & \ \mathcal{I}\llbracket \boldsymbol{M}\rrbracket\,\rho'[\lambda v.\, \lambda k.\, k\,(\kappa'\,(v))/\boldsymbol{k}]\,\lambda x.\, x) \in R_{\beta^*} \end{array}$$

Since $(\lambda x.\, x, \lambda x.\, x) \in R_{\sigma^* \to \sigma^*}$, we are done by the induction hypothesis if we can prove

$$\begin{array}{r} (\rho[\lambda v.\, \lambda k.\, k\,(\kappa\,(v))/\boldsymbol{k}], \rho'[\lambda v.\, \lambda k.\, k\,(\kappa'\,(v))/\boldsymbol{k}]) \\ \models A[\boldsymbol{k} : \tau/\delta \to \alpha/\delta] \end{array}$$

or in other words,

$$\begin{array}{rl} (\lambda v.\, \lambda k.\, k\,(\kappa\,(v)), \lambda v.\, \lambda k.\, k\,(\kappa'\,(v))) & \in R_{(\tau/\delta \to \alpha/\delta)^*} \\ & = R_{\tau^* \to (\alpha^* \to \delta^*) \to \delta^*} \quad . \end{array}$$

So take any $(v_1, v_2) \in R_{\tau^*}$ and $(k_1, k_2) \in R_{\alpha^* \to \delta^*}$. We need to prove

$$(k_1\,(\kappa\,(v_1)), k_2\,(\kappa'\,(v_2))) \in R_{\delta^*}$$

which holds since $(\kappa, \kappa') \in R_{\tau^* \to \alpha^*}$.

Case 7 (Dynamic Shift). Assume

$$\frac{A[\boldsymbol{k} : d], \sigma \vdash \boldsymbol{M} : \sigma, \beta\ [\boldsymbol{W}]}{A, d \vdash \boldsymbol{\xi k.\, M} : d, \beta\ [\underline{\boldsymbol{\xi}}\boldsymbol{k.\, W}]}\ (\textit{shift-d})$$

We need to show $(\mathcal{P}\llbracket \underline{\boldsymbol{\xi}}\boldsymbol{k.\, W}\rrbracket\,\rho\,\kappa, \mathcal{I}\llbracket \boldsymbol{\xi k.\, M}\rrbracket\,\rho'\,\kappa') \in R_{\beta^*}$ assuming $(\rho, \rho') \models A$ and $(\kappa, \kappa') \in R_{d^* \to d^*}$. From the definition of $\mathcal{P}$ and $\mathcal{I}$, we have:

$$\begin{array}{ll} & (\mathcal{P}\llbracket \underline{\boldsymbol{\xi}}\boldsymbol{k.\, W}\rrbracket\,\rho\,\kappa, \mathcal{I}\llbracket \boldsymbol{\xi k.\, M}\rrbracket\,\rho'\,\kappa') \in R_{\beta^*} \\ \iff & (\mathcal{P}\llbracket \overline{\boldsymbol{W}}\rrbracket\,\rho[\boldsymbol{\lambda v^\diamond.\, \lambda k^\diamond.\, k^\diamond} \mathbf{@}(\kappa\,(\boldsymbol{v}^\diamond))/\boldsymbol{k}]\,\lambda x.\, x, \\ & \ \mathcal{I}\llbracket \boldsymbol{M}\rrbracket\,\rho'[\lambda v^\diamond.\, \lambda k^\diamond.\, k^\diamond\,(\kappa'\,(v^\diamond))/\boldsymbol{k}]\,\lambda x.\, x) \in R_{\beta^*} \end{array}$$

Since $(\lambda x.\, x, \lambda x.\, x) \in R_{\sigma^* \to \sigma^*}$, we are done by the induction hypothesis if we can prove

$$\begin{array}{r} (\rho[\boldsymbol{\lambda v^\diamond.\, \lambda k^\diamond.\, k^\diamond} \mathbf{@}(\kappa\,(\boldsymbol{v}^\diamond))/\boldsymbol{k}], \rho'[\lambda v^\diamond.\, \lambda k^\diamond.\, k^\diamond\,(\kappa'\,(v^\diamond))/\boldsymbol{k}]) \\ \models A[\boldsymbol{k} : d] \end{array}$$

or in other words,

$$(\boldsymbol{\lambda v^\diamond.\, \lambda k^\diamond.\, k^\diamond} \mathbf{@}(\kappa\,(\boldsymbol{v}^\diamond)), \lambda v^\diamond.\, \lambda k^\diamond.\, k^\diamond\,(\kappa'\,(v^\diamond))) \in R_{d^*} \quad .$$

Now, $(\boldsymbol{v}^\diamond, v^\diamond) \in R_d$ holds, so from $(\kappa, \kappa') \in R_{d^* \to d^*}$, we have $(\kappa\,(\boldsymbol{v}^\diamond), \kappa'\,(v^\diamond)) \in R_d$, that is,

$$\mathcal{E}\llbracket \kappa\,(\boldsymbol{v}^\diamond)\rrbracket\,\mu_{id} =_\beta \kappa'\,(v^\diamond) \qquad (2)$$

To obtain $(\boldsymbol{\lambda v^\diamond.\, \lambda k^\diamond.\, k^\diamond} \mathbf{@}(\kappa\,(\boldsymbol{v}^\diamond)), \lambda v^\diamond.\, \lambda k^\diamond.\, k^\diamond\,(\kappa'\,(v^\diamond))) \in R_{d^*}$, we calculate:

$$\begin{array}{lll} & \mathcal{E}\llbracket \boldsymbol{\lambda v^\diamond.\, \lambda k^\diamond.\, k^\diamond} \mathbf{@}(\kappa\,(\boldsymbol{v}^\diamond))\rrbracket\,\mu_{id} & \\ = & \lambda v^\diamond.\, \lambda k^\diamond.\, \mathcal{E}\llbracket \boldsymbol{k}^\diamond \mathbf{@}(\kappa\,(\boldsymbol{v}^\diamond))\rrbracket\,\mu_{id}[v^\diamond/\boldsymbol{v}^\diamond, k^\diamond/\boldsymbol{k}^\diamond] & \\ & & \text{definition of } \mathcal{E} \text{ twice} \\ = & \lambda v^\diamond.\, \lambda k^\diamond.\, \mathcal{E}\llbracket \boldsymbol{k}^\diamond \mathbf{@}(\kappa\,(\boldsymbol{v}^\diamond))\rrbracket\,\mu_{id} & \text{definition of } \mu_{id} \\ = & \lambda v^\diamond.\, \lambda k^\diamond.\, (\mathcal{E}\llbracket \boldsymbol{k}^\diamond\rrbracket\,\mu_{id})\,(\mathcal{E}\llbracket \kappa\,(\boldsymbol{v}^\diamond)\rrbracket\,\mu_{id}) & \text{definition of } \mathcal{E} \\ = & \lambda v^\diamond.\, \lambda k^\diamond.\, k^\diamond\,(\mathcal{E}\llbracket \kappa\,(\boldsymbol{v}^\diamond)\rrbracket\,\mu_{id}) & \text{definition of } \mathcal{E} \\ =_\beta & \lambda v^\diamond.\, \lambda k^\diamond.\, k^\diamond\,(\kappa'\,(v^\diamond)) & \text{Equation (2)} \end{array}$$

Case 8 (Reset). Assume

$$\frac{A, \sigma \vdash \boldsymbol{M} : \sigma, \tau\ [\boldsymbol{W}]}{A, \alpha \vdash \langle \boldsymbol{M} \rangle : \tau, \alpha\ [\langle \boldsymbol{W} \rangle]}\ (\textit{reset})$$

We need to show $(\mathcal{P}\llbracket \langle \boldsymbol{W} \rangle\rrbracket\,\rho\,\kappa, \mathcal{I}\llbracket \langle \boldsymbol{M} \rangle\rrbracket\,\rho'\,\kappa') \in R_{\alpha^*}$ assuming $(\rho, \rho') \models A$ and $(\kappa, \kappa') \in R_{\tau^* \to \alpha^*}$. From the definition of $\mathcal{P}$ and $\mathcal{I}$, we have:

$$\begin{array}{ll} & (\mathcal{P}\llbracket \langle \boldsymbol{W} \rangle\rrbracket\,\rho\,\kappa, \mathcal{I}\llbracket \langle \boldsymbol{M} \rangle\rrbracket\,\rho'\,\kappa') \in R_{\alpha^*} \\ \iff & (\kappa\,(\mathcal{P}\llbracket \boldsymbol{W}\rrbracket\,\rho\,\lambda x.\, x), \kappa'\,(\mathcal{I}\llbracket \boldsymbol{M}\rrbracket\,\rho'\,\lambda x.\, x)) \in R_{\alpha^*} \end{array}$$

From $(\kappa, \kappa') \in R_{\tau^* \to \alpha^*}$, it suffices to show

$$(\mathcal{P}\llbracket \boldsymbol{W}\rrbracket\,\rho\,\lambda x.\, x, \mathcal{I}\llbracket \boldsymbol{M}\rrbracket\,\rho'\,\lambda x.\, x) \in R_{\tau^*}$$

which holds by the induction hypothesis since $(\lambda x.\, x, \lambda x.\, x) \in R_{\sigma^* \to \sigma^*}$.

This completes the cases for the proof. □