Chapter 5

Logical Relations for Call-by-value Delimited Continuations

Kenichi Asai[1]

Abstract: Logical relations, defined inductively on the structure of types, provide a powerful tool to characterize higher-order functions. They often enable us to prove correctness of a program transformer written with higher-order functions concisely. This paper demonstrates that the technique of logical relations can be used to characterize call-by-value functions as well as delimited continuations. Based on the traditional logical relations for call-by-name functions, logical relations for call-by-value functions are first defined, whose CPS variant is used to prove the correctness of an offline specializer for the call-by-value λ-calculus. They are then modified to cope with delimited continuations and are used to establish the correctness of an offline specializer for the call-by-value λ-calculus with delimited continuation constructs, shift and reset. This is the first correctness proof for such a specializer. Along the development, correctness of the continuation-based and shift/reset-based let-insertion and A-normalization is established.

5.1 INTRODUCTION

Whenever we build a program transformer, be it a compiler, an optimizer, or a specializer, we need to establish its correctness. We have to show that the semantics of a program does not change before and after the transformation. As a program transformer gets sophisticated, however, it becomes harder to prove its correctness. In particular, the non-trivial use of higher-order functions in the transformer makes the correctness proof particularly difficult. A simple structural

---

[1] Ochanomizu University, 2-1-1 Otsuka, Bunkyo-ku, Tokyo 112-8610, Japan; Email: `asai@is.ocha.ac.jp`

induction on the input program does not usually work, because we can not easily characterize their behavior.

The technique of logical relations [16] is one of the proof methods that is often used in such a case. With the help of types, it enables us to define a set of relations that captures necessary properties of higher-order functions. Notably, Wand [20] used this technique to prove correctness of an offline specializer [14] in which higher-order functions rather than closures were used for the representation of abstractions. However, the logical relations used by Wand were for call-by-name functions. They were used to prove the correctness of a specializer for the call-by-name λ-calculus, but are not directly applicable to the call-by-value languages.

In this paper, we demonstrate that the technique of logical relations can be used to characterize call-by-value functions as well as delimited continuations. We first modify Wand's logical relations so that we can use them for call-by-value functions. We then prove the correctness of an offline specializer for the call-by-value λ-calculus. It is written in continuation-passing style (CPS) and uses the continuation-based let-insertion to avoid computation elimination/duplication.

It is well-known that by using delimited continuation constructs, *shift* and *reset*, introduced by Danvy and Filinski [7], it is possible to implement the let-insertion in direct style [18]. We demonstrate that the correctness of this direct-style specializer with the shift/reset-based let-insertion can be also established by properly characterizing delimited continuations in logical relations.

Then, the specializer is extended to cope with shift and reset in the source language. To this end, the specialization-time delimited continuations are used to implement the delimited continuations in the source language. To characterize such delimited continuations, we define logical relations based on Danvy and Filinski's type system [6]. Thanks to the explicit reference to the types of continuations and the final result, we can establish the correctness of the specializer. This is the first correctness proof for the offline specializer for the call-by-value λ-calculus with shift and reset. The present author previously showed the correctness of a similar offline specializer [3], but it produced the result of specialization in CPS.

The contributions of this paper are summarized as follows:

- We show that the technique of logical relations can be used to characterize call-by-value functions as well as delimited continuations.
- We show for the first time the correctness of the offline specializer for the call-by-value λ-calculus with shift and reset.
- Along the development, we establish the correctness of the continuation-based let-insertion, the shift/reset-based let-insertion, the continuation-based A-normalization [13], and the shift/reset-based A-normalization.

The paper is organized as follows. After showing preliminaries in Section 5.2, the call-by-name specializer and its correctness proof by Wand are reviewed in Section 5.3. We then show the logical relations for call-by-value functions in

Section 5.4, and use (a CPS variant of) them to prove the correctness of a specializer for the call-by-value λ-calculus in Section 5.5. In Section 5.6, we transform the specializer into direct style and prove its correctness. Then, we further extend the specializer to cope with shift and reset. We show an interpreter and an A-normalizer in Section 5.7, a specializer in Section 5.8, a type system in Section 5.9, and logical relations with which the correctness is established in Section 5.10. Related work is in Section 5.11 and the paper concludes in Section 5.12. A complete proof of correctness of the offline specializer for shift and reset is found in the technical report [4].

ACKNOWLEDGMENTS

Most of the work has been done while the author was visiting Northeastern University. Special thanks to Mitch Wand for hosting my stay as well as support and encouragements. The use of de Bruijn levels to avoid the name generation problem was suggested by Olivier Danvy.

5.2 PRELIMINARIES

The metalanguage we use is a left-to-right λ-calculus extended with shift and reset as well as datatype constructors. The syntax is given as follows:

$$\begin{array}{rcl} M,K & = & x \mid \lambda x.M \mid M\,M \mid \xi k.M \mid \langle M \rangle \mid n \mid M+1 \mid \\ & & \mathrm{Var}(n) \mid \mathrm{Lam}(n,M) \mid \mathrm{App}(M,M) \mid \mathrm{Shift}(n,M) \mid \mathrm{Reset}(M) \mid \\ & & \overline{\mathrm{Lam}}(n,M) \mid \overline{\mathrm{App}}(M,M) \mid \overline{\mathrm{Shift}}(n,M) \mid \overline{\mathrm{Reset}}(M) \mid \\ & & \underline{\mathrm{Lam}}(n,M) \mid \underline{\mathrm{App}}(M,M) \mid \underline{\mathrm{Shift}}(n,M) \mid \underline{\mathrm{Reset}}(M) \end{array}$$

$\xi k.M$ and $\langle M \rangle$ represent shift and reset, respectively, and appear only later in the paper. Datatype constructors are for representing the input and output terms to our specializer. In this baselanguage, an integer n is used to represent a variable. For this purpose, the language contains an integer and an add-one operation. As usual, we use overline and underline to indicate static and dynamic terms, respectively. We assume that all the datatype constructors are strict. Among the metalanguage, a value (ranged over by a metavariable V) is either a variable, an abstraction, an integer, or one of constructors whose arguments are values.

When a specializer produces its output, it needs to generate fresh variables. To make the presentation simple, we use so-called the de Bruijn levels [9] (not indices). Define the following five strict operators:

$$\begin{array}{rcl} \mathrm{var}(m) & = & \lambda n.\,\mathrm{Var}(m) \\ \mathrm{lam}(f) & = & \lambda n.\,\mathrm{Lam}(n, f\,(n+1)) \\ \mathrm{app}(f_1,f_2) & = & \lambda n.\,\mathrm{App}(f_1\,n, f_2\,n) \\ \mathrm{shift}(f) & = & \lambda n.\,\mathrm{Shift}(n, f\,(n+1)) \\ \mathrm{reset}(f) & = & \lambda n.\,\mathrm{Reset}(f\,n) \end{array}$$

They are used to represent a term parameterized with a variable name. Given a term M in the de Bruijn level notation, we define the operation $\downarrow_n M$ of obtaining a concrete term as: $\downarrow_n M = M\,n$. Thus, we have:

$$\begin{array}{rl} & \downarrow_3 (\mathrm{app}(\mathrm{lam}(\lambda x.\,\mathrm{lam}(\lambda y.\,\mathrm{var}(x))), \mathrm{lam}(\lambda y.\,\mathrm{var}(y)))) \\ = & \mathrm{App}(\mathrm{Lam}(3, \mathrm{Lam}(4, \mathrm{Var}(3))), \mathrm{Lam}(3, \mathrm{Var}(3)))\ . \end{array}$$

Since we can freely transform a term with de Bruijn levels into the one without, we will use the former as the output of specializers.

Throughout this paper, we use three kinds of equalities between terms in the metalanguage: $=$ for definition or α-equality, $\sim_{\mathrm{n}}$ for β-equality under call-by-name semantics, and $\sim_{\mathrm{v}}$ for β-equality under call-by-value semantics. The call-by-value β-equality in the presence of shift and reset is defined by Kameyama and Hasegawa [15, Fig. 2].

5.3 SPECIALIZER FOR CALL-BY-NAME λ-CALCULUS

In this section, we review the specializer for the call-by-name λ-calculus and its correctness proof using the technique of logical relations presented by Wand [20].

A specializer reduces expressions that are known (or *static*) at specialization time and leaves unknown (or *dynamic*) expressions intact. Thus, it consists of two parts: an interpreter for static expressions and a residualizer for dynamic expressions. An interpreter for the input language is defined as follows:

$$\begin{array}{rcl} I_1 \llbracket \mathrm{Var}(n) \rrbracket\, \rho & = & \rho\,(n) \\ I_1 \llbracket \mathrm{Lam}(n, M) \rrbracket\, \rho & = & \lambda x.\, I_1 \llbracket M \rrbracket\, \rho[x/n] \\ I_1 \llbracket \mathrm{App}(M_1, M_2) \rrbracket\, \rho & = & (I_1 \llbracket M_1 \rrbracket\, \rho)\,(I_1 \llbracket M_2 \rrbracket\, \rho) \end{array}$$

where $\rho[x/n]$ is the same environment as ρ except that $\rho\,(n) = x$.

The residualizer is almost the identity function except for the use of de Bruijn levels to avoid name clashes:

$$\begin{array}{rcl} \mathcal{D}_1 \llbracket \mathrm{Var}(n) \rrbracket\, \rho & = & \rho\,(n) \\ \mathcal{D}_1 \llbracket \mathrm{Lam}(n, M) \rrbracket\, \rho & = & \mathrm{lam}(\lambda x.\, \mathcal{D}_1 \llbracket M \rrbracket\, \rho[\mathrm{var}(x)/n]) \\ \mathcal{D}_1 \llbracket \mathrm{App}(M_1, M_2) \rrbracket\, \rho & = & \mathrm{app}(\mathcal{D}_1 \llbracket M_1 \rrbracket\, \rho, \mathcal{D}_1 \llbracket M_2 \rrbracket\, \rho) \end{array}$$

An offline specializer is given by putting the interpreter and the residualizer together:

$$\begin{array}{rcl} \mathcal{P}_1 \llbracket \mathrm{Var}(n) \rrbracket\, \rho & = & \rho\,(n) \\ \mathcal{P}_1 \llbracket \overline{\mathrm{Lam}}(n, W) \rrbracket\, \rho & = & \lambda x.\, \mathcal{P}_1 \llbracket W \rrbracket\, \rho[x/n] \\ \mathcal{P}_1 \llbracket \underline{\mathrm{Lam}}(n, W) \rrbracket\, \rho & = & \mathrm{lam}(\lambda x.\, \mathcal{P}_1 \llbracket W \rrbracket\, \rho[\mathrm{var}(x)/n]) \\ \mathcal{P}_1 \llbracket \overline{\mathrm{App}}(W_1, W_2) \rrbracket\, \rho & = & (\mathcal{P}_1 \llbracket W_1 \rrbracket\, \rho)\,(\mathcal{P}_1 \llbracket W_2 \rrbracket\, \rho) \\ \mathcal{P}_1 \llbracket \underline{\mathrm{App}}(W_1, W_2) \rrbracket\, \rho & = & \mathrm{app}(\mathcal{P}_1 \llbracket W_1 \rrbracket\, \rho, \mathcal{P}_1 \llbracket W_2 \rrbracket\, \rho) \end{array}$$

The specializer goes wrong if the input term is not well-annotated. Well-annotatedness of a term is specified as a binding-time analysis that, given an unannotated term, produces a well-annotated term. Here, we show a type-based binding-time analysis. Define binding-time types of expressions as follows:

$$\tau \;=\; d \mid \tau \to \tau$$

An expression of type d denotes that the expression is dynamic, while an expression of type $\tau \rightarrow \tau$ shows that it is a static function. We use a judgment of the form $A \vdash M : \tau\ [W]$, which reads: under a type environment A, a term M has a binding-time type τ and is annotated as W. The binding-time analysis is defined by the following typing rules:

$$A[n:\tau] \vdash \mathrm{Var}(n) : \tau\ [\mathrm{Var}(n)]$$

$$\frac{A[n:\sigma] \vdash M : \tau\ [W]}{A \vdash \mathrm{Lam}(n,M) : \sigma \rightarrow \tau\ [\overline{\mathrm{Lam}}(n,W)]} \qquad \frac{A \vdash M_1 : \sigma \rightarrow \tau\ [W_1] \quad A \vdash M_2 : \sigma\ [W_2]}{A \vdash \mathrm{App}(M_1,M_2) : \tau\ [\overline{\mathrm{App}}(W_1,W_2)]}$$

$$\frac{A[n:d] \vdash M : d\ [W]}{A \vdash \mathrm{Lam}(n,M) : d\ [\underline{\mathrm{Lam}}(n,W)]} \qquad \frac{A \vdash M_1 : d\ [W_1] \quad A \vdash M_2 : d\ [W_2]}{A \vdash \mathrm{App}(M_1,M_2) : d\ [\underline{\mathrm{App}}(W_1,W_2)]}$$

To show the correctness of the specializer, Wand [20] uses the technique of logical relations. Define logical relations between terms in the metalanguage by induction on the structure of binding-time types as follows:

$$\begin{array}{lcl} (M,M') \in R_d & \Longleftrightarrow & I_1\, [\![\downarrow_n M]\!]\, \rho_{id} \sim_{\mathrm{n}} M' \text{ for any large } n \text{ (defined below)} \\ (M,M') \in R_{\sigma\rightarrow\tau} & \Longleftrightarrow & \forall (N,N') \in R_\sigma.\ (M\,N, M'\,N') \in R_\tau \end{array}$$

where $\rho_{id}(n) = z_n$ for all n. It relates free variables in the base- and metalanguage. Since the logical relations are defined on open terms, we need to relate free variables in the base- and metalanguage in some way. We choose here to relate a baselanguage variable Var(n) to a metalanguage variable z_n.

In the definition of R_d, M is a metalanguage term in the de Bruijn level notation that is either a value representing a baselanguage term or a term that is equal to (or evaluates to) a value representing a baselanguage term in the underlying semantics of the metalanguage (in this section, call-by-name).

The choice of n in R_d needs a special attention. Since M is possibly an open term, n has to be chosen so that it does not capture free variables in M. We ensure this property by the side condition "for any large n." n is defined to be large if n is greater than any free variables in the baselanguage term M.

For environments ρ and ρ', we say $(\rho,\rho') \models A$ iff $(\rho\,(n), \rho'\,(n)) \in R_{A(n)}$ for all $n \in dom(A)$, where $dom(A)$ is the domain of A. Then, we can show the following theorem by structural induction over types:

Theorem 5.1 (Wand [20]). *If* $A \vdash M : \tau\ [W]$ *and* $(\rho,\rho') \models A$, *then* $(\mathcal{P}_1\, [\![W]\!]\, \rho, I_1\, [\![M]\!]\, \rho') \in R_\tau$.

By instantiating it to an empty environment ρ_ϕ, we obtain the following corollary, which establishes the correctness of specialization.

Corollary 5.2 (Wand [20]). *If* $\vdash M : d\ [W]$, *then* $I_1\, [\![\downarrow_0 (\mathcal{P}_1\, [\![W]\!]\, \rho_\phi)]\!]\, \rho_{id} \sim_{\mathrm{n}} I_1\, [\![M]\!]\, \rho_\phi$.

5.4 LOGICAL RELATIONS FOR CALL-BY-VALUE λ-CALCULUS

Define logical relations for the call-by-value λ-calculus as follows:

$$\begin{array}{lcl} (M,M') \in R_d & \Longleftrightarrow & I_1\,\llbracket \downarrow_n M \rrbracket\, \rho_{id} \sim_{\mathrm{v}} M' \text{ for any large } n \\ (M,M') \in R_{\sigma\to\tau} & \Longleftrightarrow & \forall (V,V') \in R_\sigma.\ (M\,V, M'\,V') \in R_\tau \end{array}$$

There are two differences from the logical relations in the previous section. First, call-by-value equality $\sim_{\mathrm{v}}$ is used instead of call-by-name equality $\sim_{\mathrm{n}}$ in the definition of R_d. Secondly, M and M' are allowed to be in $R_{\sigma\to\tau}$ if they transform only related *values* (rather than arbitrary terms) into related terms.

If we could prove Theorem 5.1 with this definition of R_τ, we would have obtained as a corollary the correctness of the specializer in the call-by-value semantics. However, the proof fails for static applications. In fact, the specializer is not correct under the call-by-value semantics.

5.5 SPECIALIZER IN CPS

The correctness under the call-by-value semantics does not hold for the specializer in Section 5.3 because it may discard a non-terminating computation. The standard method to recover the correctness is to perform *let-insertion* [5]. Since let-insertion requires explicit manipulation of continuations, we first rewrite our specializer into CPS as follows:

$$\begin{array}{rcl} \mathcal{P}_2\,\llbracket \mathrm{Var}(n) \rrbracket\, \rho\, \kappa & = & \kappa\,(\rho\,(n)) \\ \mathcal{P}_2\,\llbracket \overline{\mathrm{Lam}}(n,W) \rrbracket\, \rho\, \kappa & = & \kappa\,(\lambda x.\, \lambda k.\, \mathcal{P}_2\,\llbracket W \rrbracket\, \rho[x/n]\, k) \\ \mathcal{P}_2\,\llbracket \underline{\mathrm{Lam}}(n,W) \rrbracket\, \rho\, \kappa & = & \kappa\,(\mathrm{lam}(\lambda x.\, \mathcal{P}_2\,\llbracket W \rrbracket\, \rho[\mathrm{var}(x)/n]\, \lambda x.x)) \\ \mathcal{P}_2\,\llbracket \overline{\mathrm{App}}(W_1,W_2) \rrbracket\, \rho\, \kappa & = & \mathcal{P}_2\,\llbracket W_1 \rrbracket\, \rho\, \lambda m.\, \mathcal{P}_2\,\llbracket W_2 \rrbracket\, \rho\, \lambda n.\, m\, n\, \kappa \\ \mathcal{P}_2\,\llbracket \underline{\mathrm{App}}(W_1,W_2) \rrbracket\, \rho\, \kappa & = & \mathcal{P}_2\,\llbracket W_1 \rrbracket\, \rho\, \lambda m.\, \mathcal{P}_2\,\llbracket W_2 \rrbracket\, \rho\, \lambda n.\, \kappa\,(\mathrm{app}(m,n)) \end{array}$$

We then replace the last rule with the following:

$$\begin{array}{rcl} \mathcal{P}_2\,\llbracket \underline{\mathrm{App}}(W_1,W_2) \rrbracket\, \rho\, \kappa & = & \mathcal{P}_2\,\llbracket W_1 \rrbracket\, \rho\, \lambda m.\, \mathcal{P}_2\,\llbracket W_2 \rrbracket\, \rho\, \lambda n. \\ & & \qquad \mathrm{let}(\mathrm{app}(m,n), \mathrm{lam}(\lambda t.\, \kappa\,(\mathrm{var}(t)))) \end{array}$$

where $\mathrm{let}(M_1, \mathrm{lam}(\lambda t.\, M_2))$ is an abbreviation for $\mathrm{app}(\mathrm{lam}(\lambda t.\, M_2), M_1)$. Whenever an application is residualized, we insert a let-expression to residualize it exactly once with a unique name t, and continue the rest of the specialization with this name. Since the residualized application is not passed to the continuation κ, it will never be discarded even if κ discards its argument.

The let-insertion technique can be regarded as performing A-normalization [13] on the fly during specialization. If we extract the rules for variables, dynamic abstractions, and dynamic applications from $\mathcal{P}_2$, we obtain the following one-pass A-normalizer written in CPS [13]:

$$\begin{array}{rcl} \mathcal{A}_1\,\llbracket \mathrm{Var}(n) \rrbracket\, \rho\, \kappa & = & \kappa\,(\rho\,(n)) \\ \mathcal{A}_1\,\llbracket \mathrm{Lam}(n,M) \rrbracket\, \rho\, \kappa & = & \kappa\,(\mathrm{lam}(\lambda x.\, \mathcal{A}_1\,\llbracket M \rrbracket\, \rho[\mathrm{var}(x)/n]\, \lambda x.x)) \\ \mathcal{A}_1\,\llbracket \mathrm{App}(M_1,M_2) \rrbracket\, \rho\, \kappa & = & \mathcal{A}_1\,\llbracket M_1 \rrbracket\, \rho\, \lambda m.\, \mathcal{A}_1\,\llbracket M_2 \rrbracket\, \rho\, \lambda n. \\ & & \qquad \mathrm{let}(\mathrm{app}(m,n), \mathrm{lam}(\lambda t.\, \kappa\,(\mathrm{var}(t)))) \end{array}$$

We now want to show the correctness of the specializer $\mathcal{P}_2$ under the call-by-value semantics. Namely, we want to show $I_1\,[\![\downarrow_0 (\mathcal{P}_2\,[\![W]\!]\,\rho_\phi\,\lambda x.x)]\!]\,\rho_{id} \sim_{\mathrm{v}} I_1\,[\![M]\!]\,\rho_\phi$ along the similar story as we did in Section 5.3. Let us define the base case R_d as follows:

$$(M,M') \in R_d \quad \Longleftrightarrow \quad I_1\,[\![\downarrow_n M]\!]\,\rho_{id} \sim_{\mathrm{v}} M' \text{ for any large } n \ .$$

Then, we want to show $(\mathcal{P}_2\,[\![W]\!]\,\rho_\phi\,\lambda x.x, I_1\,[\![M]\!]\,\rho_\phi) \in R_d$ with a suitable definition of $R_{\sigma\to\tau}$. To prove it, we first generalize the statement to make induction work. Rather than proving only the case where environments and continuations are the empty ones, we prove something like:

$$(\mathcal{P}_2\,[\![W]\!]\,\rho\,\lambda v.K, (\lambda v'.K')\,(I_1\,[\![M]\!]\,\rho')) \in R_\tau$$

for some suitable ρ, ρ', $\lambda v.K$, and $\lambda v'.K'$. Since I_1 is written in direct style, we introduce its continuation as a form of a direct application.

Now, how can we define $R_{\sigma\to\tau}$? Unlike Section 5.3, it is not immediately clear how to define $R_{\sigma\to\tau}$ because the specializer is written in CPS. We need to relate $\mathcal{P}_2\,[\![W]\!]\,\rho\,\lambda v.K$ and $(\lambda v'.K')\,(I_1\,[\![M]\!]\,\rho')$ properly. To do so, we need to characterize precisely the two continuations, $\lambda v.K$ and $\lambda v'.K'$, and the final results. Going back to the definition of $\mathcal{P}_2$, we notice two things:

- $\mathcal{P}_2\,[\![W]\!]\,\rho\,\lambda v.K$ as a whole returns a dynamic expression.
- $\lambda v.K$ returns a dynamic expression, given some value v.

In ordinary CPS programs, the return type of continuations is polymorphic. It can be of any type, usually referred to as a type *Answer*. Here, we used continuations in a non-standard way, however. We instantiated the *Answer* type into a type of dynamic expressions and used it to construct dynamic expressions.

Taking into account that the type of dynamic expressions is d, the above observation leads us to the following definition of logical relations:

$$\begin{array}{lll}
(M,M') \in R_d & \Longleftrightarrow & I_1\,[\![\downarrow_n M]\!]\,\rho_{id} \sim_{\mathrm{v}} M' \text{ for any large } n \\
(M,M') \in R_{\sigma\to\tau} & \Longleftrightarrow & \forall (V,V') \in R_\sigma.\ \forall (\lambda v.K, \lambda v'.K') \models \tau \rightsquigarrow d. \\
& & (M\,V\,\lambda v.K, (\lambda v'.K')\,(M'\,V')) \in R_d
\end{array}$$

where $(\lambda v.K, \lambda v'.K') \models \tau \rightsquigarrow d$ is simultaneously defined as follows:

$$(\lambda v.K, \lambda v'.K') \models \tau \rightsquigarrow d \quad \Longleftrightarrow \quad \forall (V,V') \in R_\tau.\ ((\lambda v.K)\,V, (\lambda v'.K')\,V') \in R_d$$

Intuitively, $(\lambda v.K, \lambda v'.K') \models \tau \rightsquigarrow d$ means that $\lambda v.K$ and $\lambda v'.K'$ are related continuations that, given related values of type τ, produce related results of type d. Using this definition, $(M,M') \in R_{\sigma\to\tau}$ states that M and M' are related if they produce related results of type d, given related values of type σ and related continuations of type $\tau \rightsquigarrow d$. In the following, we use $\rightsquigarrow$ for the type of continuations.

With this definition of logical relations, we can prove the correctness of $\mathcal{P}_2$ under the call-by-value semantics.

Theorem 5.3. *If* $A \vdash M : \tau\ [W]$, $(\rho, \rho') \models A$, *and* $(\lambda v. K, \lambda v'. K') \models \tau \rightsquigarrow d$, *then* $(\mathcal{P}_2\, [\![W]\!]\, \rho\, \lambda v. K, (\lambda v'. K')\, (I_1\, [\![M]\!]\, \rho')) \in R_d$.

The proof of this theorem is by induction on the structure of the proof of $A \vdash M : \tau\ [W]$. Even though $\mathcal{P}_2$ is written in CPS, the induction does work thanks to the explicit reference to the types of continuations and the final result. The proof proceeds in a CPS manner. In particular, the cases for (both static and dynamic) applications go from left to right. We use the induction hypotheses for the function part and the argument part in this order.

By instantiating the theorem to the case where both the environment and the continuation are empty, we obtain the following corollary that establishes the correctness of a specializer using the continuation-based let-insertion:

Corollary 5.4. *If* $\vdash M : d\ [W]$, *then* $I_1\, [\![\downarrow_0 (\mathcal{P}_2\, [\![W]\!]\, \rho_\phi\, \lambda x. x)]\!]\, \rho_{id} \sim_{\mathrm{v}} I_1\, [\![M]\!]\, \rho_\phi$.

If we annotate the input to the specializer completely dynamic, the specializer behaves exactly the same as the A-normalizer. Thus, the theorem can be instantiated to the following corollary, which proves the correctness of the continuation-based A-normalization.

Corollary 5.5. $I_1\, [\![\downarrow_0 (\mathcal{A}_1\, [\![M]\!]\, \rho_\phi\, \lambda x. x)]\!]\, \rho_{id} \sim_{\mathrm{v}} I_1\, [\![M]\!]\, \rho_\phi$ *for any closed* M.

5.6 SPECIALIZER IN DIRECT STYLE

In this section, we present a specializer written in direct style and show its correctness under the call-by-value semantics. Since we have already established the correctness of a specializer written in CPS in the previous section, the development in this section is easy. Roughly speaking, we transform the results in the previous section *back to direct style* [8]. During this process, we use the first-class delimited continuation constructs, *shift* and *reset*, to cope with non-standard use of continuations. Intuitively, shift captures the current continuation up to its enclosing reset [7]. Here is the definition of the specializer written in direct style:

$$\begin{array}{rcl}
\mathcal{P}_3\, [\![\mathrm{Var}(n)]\!]\, \rho & = & \rho\,(n) \\
\mathcal{P}_3\, [\![\overline{\mathrm{Lam}}(n, W)]\!]\, \rho & = & \lambda x. \mathcal{P}_3\, [\![W]\!]\, \rho[x/n] \\
\mathcal{P}_3\, [\![\underline{\mathrm{Lam}}(n, W)]\!]\, \rho & = & \mathrm{lam}(\lambda x. \langle \mathcal{P}_3\, [\![W]\!]\, \rho[\mathrm{var}(x)/n] \rangle) \\
\mathcal{P}_3\, [\![\overline{\mathrm{App}}(W_1, W_2)]\!]\, \rho & = & (\mathcal{P}_3\, [\![W_1]\!]\, \rho)\, (\mathcal{P}_3\, [\![W_2]\!]\, \rho) \\
\mathcal{P}_3\, [\![\underline{\mathrm{App}}(W_1, W_2)]\!]\, \rho & = & \xi\kappa. \mathrm{let}(\mathrm{app}(\mathcal{P}_3\, [\![W_1]\!]\, \rho, \mathcal{P}_3\, [\![W_2]\!]\, \rho), \mathrm{lam}(\lambda t. \kappa\,(\mathrm{var}(t))))
\end{array}$$

As in the previous section, we obtain the one-pass A-normalizer written in direct style with shift and reset [3] by extracting dynamic rules from $\mathcal{P}_3$:

$$\begin{array}{rcl}
\mathcal{A}_2\, [\![\mathrm{Var}(n)]\!]\, \rho & = & \rho\,(n) \\
\mathcal{A}_2\, [\![\mathrm{Lam}(x, M)]\!]\, \rho & = & \mathrm{lam}(\lambda x. \langle \mathcal{A}_2\, [\![M]\!]\, \rho[\mathrm{var}(x)/n] \rangle) \\
\mathcal{A}_2\, [\![\mathrm{App}(M_1, M_2)]\!]\, \rho & = & \xi\kappa. \mathrm{let}(\mathrm{app}(\mathcal{A}_2\, [\![M_1]\!]\, \rho, \mathcal{A}_2\, [\![M_2]\!]\, \rho), \mathrm{lam}(\lambda t. \kappa\,(\mathrm{var}(t))))
\end{array}$$

To define suitable logical relations for the specializer written in direct style (with shift and reset), we need to correctly handle delimited continuations. This is done by observing the exact correspondence between continuations in the previous section and delimited continuations in this section. In particular, we type the result of the delimited continuations as d.

Logical relations for the direct-style specializer with delimited continuations are defined as follows:

$$\begin{array}{lcl}(M,M') \in R_d & \Longleftrightarrow & I_1\,[\![\downarrow_n M]\!]\,\rho_{id} \sim_{\mathrm{v}} M' \text{ for any large } n \\ (M,M') \in R_{\sigma\to\tau} & \Longleftrightarrow & \forall(V,V') \in R_\sigma.\forall(\lambda v.K,\lambda v'.K') \models \tau \leadsto d. \\ & & (\langle(\lambda v.K)\,(M\,V)\rangle,(\lambda v'.K')\,(M'\,V')) \in R_d\end{array}$$

where $(\lambda v.K,\lambda v'.K') \models \tau \leadsto d$ is simultaneously defined as follows:

$$(\lambda v.K,\lambda v'.K') \models \tau \leadsto d \quad \Longleftrightarrow \quad \forall(V,V') \in R_\tau.\ (\langle(\lambda v.K)\,V\rangle,(\lambda v'.K')\,V') \in R_d$$

Then, the correctness of the specializer is stated as follows:

Theorem 5.6. *If* $A \vdash M : \tau\ [W]$, $(\rho,\rho') \models A$, *and* $(\lambda v.K,\lambda v'.K') \models \tau \leadsto d$, *then* $(\langle(\lambda v.K)\,(\mathcal{P}_3\,[\![W]\!]\,\rho)\rangle,(\lambda v'.K')\,(I_1\,[\![M]\!]\,\rho')) \in R_d$.

Although both the specializer and the interpreter are written in direct style, the proof proceeds in a CPS manner. In particular, the cases for applications go from left to right, naturally reflecting the call-by-value semantics.

By instantiating the theorem to the case where both the environment and the continuation are empty, we obtain the following corollary that establishes the correctness of a specializer using the shift/reset-based let-insertion:

Corollary 5.7. *If* $\vdash M : d\ [W]$, *then* $I_1\,[\![\downarrow_0 \langle\mathcal{P}_3\,[\![W]\!]\,\rho_\phi\rangle]\!]\,\rho_{id} \sim_{\mathrm{v}} I_1\,[\![M]\!]\,\rho_\phi$.

As before, if we annotate the input to the specializer completely dynamic, the specializer behaves exactly the same as the A-normalizer. Thus, the theorem can be instantiated to the following corollary, which proves the correctness of the direct-style A-normalization.

Corollary 5.8. $I_1\,[\![\downarrow_0 \langle\mathcal{A}_2\,[\![M]\!]\,\rho_\phi\rangle]\!]\,\rho_{id} \sim_{\mathrm{v}} I_1\,[\![M]\!]\,\rho_\phi$ *for any closed* M.

5.7 INTERPRETER AND A-NORMALIZER FOR SHIFT AND RESET

So far, shift and reset appeared only in the metalanguage. In the following sections, we develop a specializer written in direct style that can handle shift and reset in the baselanguage. We first define an interpreter, a residualizer, and an A-normalizer for the call-by-value λ-calculus with shift and reset. We then try to combine the interpreter and the A-normalizer to obtain a specializer in the next section. Here is the interpreter written in direct style:

$$\begin{array}{rcl}I_2\,[\![\mathrm{Var}(n)]\!]\,\rho & = & \rho\,(n) \\ I_2\,[\![\mathrm{Lam}(n,M)]\!]\,\rho & = & \lambda x.\,I_2\,[\![M]\!]\,\rho[x/n] \\ I_2\,[\![\mathrm{App}(M_1,M_2)]\!]\,\rho & = & (I_2\,[\![M_1]\!]\,\rho)\,(I_2\,[\![M_2]\!]\,\rho) \\ I_2\,[\![\mathrm{Shift}(n,M)]\!]\,\rho & = & \xi k.\,I_2\,[\![M]\!]\,\rho[k/n] \\ I_2\,[\![\mathrm{Reset}(M)]\!]\,\rho & = & \langle I_2\,[\![M]\!]\,\rho\rangle\end{array}$$

We used shift and reset operations themselves to interpret shift and reset expressions. A residualizer is defined as follows:

$$\begin{aligned}
\mathcal{D}_2 \llbracket \mathrm{Var}(n) \rrbracket \rho &= \rho\,(n) \\
\mathcal{D}_2 \llbracket \mathrm{Lam}(n, M) \rrbracket \rho &= \mathrm{lam}(\lambda x.\, \mathcal{D}_2 \llbracket M \rrbracket \rho[\mathrm{var}(x)/n]) \\
\mathcal{D}_2 \llbracket \mathrm{App}(M_1, M_2) \rrbracket \rho &= \mathrm{app}(\mathcal{D}_2 \llbracket M_1 \rrbracket \rho, \mathcal{D}_2 \llbracket M_2 \rrbracket \rho) \\
\mathcal{D}_2 \llbracket \mathrm{Shift}(n, M) \rrbracket \rho &= \mathrm{shift}(\lambda k.\, \mathcal{D}_2 \llbracket M \rrbracket \rho[\mathrm{var}(k)/n]) \\
\mathcal{D}_2 \llbracket \mathrm{Reset}(M) \rrbracket \rho &= \mathrm{reset}(\mathcal{D}_2 \llbracket M \rrbracket \rho)
\end{aligned}$$

It simply renames bound variables and keeps other expressions unchanged. As before, this residualizer is not suitable for specializers. We instead use the following A-normalizer:

$$\begin{aligned}
\mathcal{A}_3 \llbracket \mathrm{Var}(n) \rrbracket \rho &= \rho\,(n) \\
\mathcal{A}_3 \llbracket \mathrm{Lam}(n, M) \rrbracket \rho &= \mathrm{lam}(\lambda x.\, \langle \mathcal{A}_3 \llbracket M \rrbracket \rho[\mathrm{var}(x)/n] \rangle) \\
\mathcal{A}_3 \llbracket \mathrm{App}(M_1, M_2) \rrbracket \rho &= \xi k.\, \mathrm{let}(\mathrm{app}(\mathcal{A}_3 \llbracket M_1 \rrbracket \rho, \mathcal{A}_3 \llbracket M_2 \rrbracket \rho), \mathrm{lam}(\lambda t.\, k\,(\mathrm{var}(t)))) \\
\mathcal{A}_3 \llbracket \mathrm{Shift}(n, M) \rrbracket \rho &= \mathrm{shift}(\lambda k.\, \langle \mathcal{A}_3 \llbracket M \rrbracket \rho[\mathrm{var}(k)/n] \rangle) \\
\mathcal{A}_3 \llbracket \mathrm{Reset}(M) \rrbracket \rho &= \mathrm{reset}(\langle \mathcal{A}_3 \llbracket M \rrbracket \rho \rangle)
\end{aligned}$$

It replaces all the application expressions in the body of abstractions, shift expressions, and reset operations with a sequence of let-expressions.

5.8 SPECIALIZER FOR SHIFT AND RESET

In this section, we show a specializer for the call-by-value λ-calculus with shift and reset. Our first attempt is to combine the interpreter and the A-normalizer as we did before for the calculi without shift and reset:

$$\begin{aligned}
\mathcal{P}_4 \llbracket \mathrm{Var}(n) \rrbracket \rho &= \rho\,(n) \\
\mathcal{P}_4 \llbracket \overline{\mathrm{Lam}}(n, W) \rrbracket \rho &= \lambda x.\, \mathcal{P}_4 \llbracket W \rrbracket \rho[x/n] \\
\mathcal{P}_4 \llbracket \underline{\mathrm{Lam}}(n, W) \rrbracket \rho &= \mathrm{lam}(\lambda x.\, \langle \mathcal{P}_4 \llbracket W \rrbracket \rho[\mathrm{var}(x)/n] \rangle) \\
\mathcal{P}_4 \llbracket \overline{\mathrm{App}}(W_1, W_2) \rrbracket \rho &= (\mathcal{P}_4 \llbracket W_1 \rrbracket \rho)\,(\mathcal{P}_4 \llbracket W_2 \rrbracket \rho) \\
\mathcal{P}_4 \llbracket \underline{\mathrm{App}}(W_1, W_2) \rrbracket \rho &= \xi k.\, \mathrm{let}(\mathrm{app}(\mathcal{P}_4 \llbracket W_1 \rrbracket \rho, \mathcal{P}_4 \llbracket W_2 \rrbracket \rho), \mathrm{lam}(\lambda t.\, k\,(\mathrm{var}(t)))) \\
\mathcal{P}_4 \llbracket \overline{\mathrm{Shift}}(n, W) \rrbracket \rho &= \xi k.\, \mathcal{P}_4 \llbracket W \rrbracket \rho[k/n] \\
\mathcal{P}_4 \llbracket \underline{\mathrm{Shift}}(n, W) \rrbracket \rho &= \mathrm{shift}(\lambda k.\, \langle \mathcal{P}_4 \llbracket W \rrbracket \rho[\mathrm{var}(k)/n] \rangle) \\
\mathcal{P}_4 \llbracket \overline{\mathrm{Reset}}(W) \rrbracket \rho &= \langle \mathcal{P}_4 \llbracket W \rrbracket \rho \rangle \\
\mathcal{P}_4 \llbracket \underline{\mathrm{Reset}}(W) \rrbracket \rho &= \mathrm{reset}(\langle \mathcal{P}_4 \llbracket W \rrbracket \rho \rangle)
\end{aligned}$$

Although this specializer does seem to work for carefully annotated inputs, it is hard to specify the well-annotated term as a simple binding-time analysis. The difficulty comes from the inconsistency between the specialization-time continuation and the runtime continuation.

In the rule for the static shift, a continuation is grabbed at specialization time, which means that we implicitly assume the grabbed continuation coincides with the actual continuation at runtime. This was actually true for the interpreter: we implemented shift in the baselanguage using shift in the metalanguage. In the specializer, however, the specialization-time continuation does not always coincide with the actual continuation. To be more specific, in the rule for dynamic abstractions, we specialize the body W in a static reset (*i.e.*, in the empty continuation) to perform A-normalization, but the actual continuation at the time when W is executed is not necessarily the empty one. Rather, it is the one when the abstraction is applied at runtime.

Given that the specialization-time continuation is not always consistent with the actual one, we have to make sure that the continuation is captured statically only when it represents the actual one. Furthermore, we have to make sure that whenever shift is residualized, its enclosing reset is also residualized. One way to express this information in the type system would be to split all the typing rules into two, one for the case when the specialization-time continuation and the actual continuation coincide (or, the continuation is known, static) and the other for the case when they do not (the continuation is unknown, dynamic). We could then statically grab the continuation only when it represents the actual one.

However, this solution leads to an extremely weak specialization. Unless an enclosing reset is known at specialization time, we cannot grab continuations statically. Thus, under dynamic abstractions, no shift operation is possible at specialization time. Furthermore, because we use a type-based binding-time analysis, it becomes impossible to perform *any* specialization under dynamic abstractions. Remember that a type system does not tell us what subexpressions appear in a given expression, but only the type of the given expression. From a type system, we cannot distinguish the expression that does not contain any shift expressions from the one that does. Thus, even if W_1 turns out to have a static function type in $\mathrm{App}(W_1, W_2)$ (and thus it appears that this application can be performed statically), we cannot actually perform this application, because the toplevel operator of W_1 might be a shift operation that passes a function to the grabbed continuation. In other words, we cannot determine the binding-time of $\mathrm{App}(W_1, W_2)$ from the binding-time of W_1, which makes it difficult to construct a simple type-based binding-time analysis.

The solution we employ takes a different approach. We maintain the consistency between specialization-time continuations and actual ones *all the time*. In other words, we make the continuation always static. The modified specializer is presented as follows:

$$
\begin{array}{rcl}
\mathcal{P}_5\,[\![\mathrm{Var}(n)]\!]\,\rho & = & \rho\,(n) \\
\mathcal{P}_5\,[\![\overline{\mathrm{Lam}}(n,W)]\!]\,\rho & = & \lambda x.\,\mathcal{P}_5\,[\![W]\!]\,\rho[x/n] \\
\mathcal{P}_5\,[\![\underline{\mathrm{Lam}}(n,W)]\!]\,\rho & = & \mathrm{lam}(\lambda x.\,\mathrm{shift}(\lambda k.\,\langle \mathrm{reset}(\mathrm{app}(\mathrm{var}(k), \mathcal{P}_5\,[\![W]\!]\,\rho[\mathrm{var}(x)/n]))\rangle)) \\
\mathcal{P}_5\,[\![\overline{\mathrm{App}}(W_1,W_2)]\!]\,\rho & = & (\mathcal{P}_5\,[\![W_1]\!]\,\rho)\,(\mathcal{P}_5\,[\![W_2]\!]\,\rho) \\
\mathcal{P}_5\,[\![\underline{\mathrm{App}}(W_1,W_2)]\!]\,\rho & = & \xi k.\,\mathrm{reset}(\mathrm{let}(\mathrm{app}(\mathcal{P}_5\,[\![W_1]\!]\,\rho, \mathcal{P}_5\,[\![W_2]\!]\,\rho), \mathrm{lam}(\lambda t.\,k\,(\mathrm{var}(t))))) \\
\mathcal{P}_5\,[\![\overline{\mathrm{Shift}}(n,W)]\!]\,\rho & = & \xi k.\,\mathcal{P}_5\,[\![W]\!]\,\rho[k/n] \\
\mathcal{P}_5\,[\![\underline{\mathrm{Shift}}(n,W)]\!]\,\rho & = & \xi k.\,\mathcal{P}_5\,[\![W]\!]\,\rho[\mathrm{lam}(\lambda v.\,\langle k\,(\mathrm{var}(v))\rangle)/n] \\
\mathcal{P}_5\,[\![\overline{\mathrm{Reset}}(W)]\!]\,\rho & = & \langle \mathcal{P}_5\,[\![W]\!]\,\rho \rangle
\end{array}
$$

There are four changes from $\mathcal{P}_4$. The first and the most important change is in the rule for dynamic abstractions. Rather than specializing the body W of a dynamic abstraction in the empty context, we specialize it in the context $\mathrm{reset}(\mathrm{app}(\mathrm{var}(k),\cdot))$. This specialization-time continuation $\mathrm{reset}(\mathrm{app}(\mathrm{var}(k),\cdot))$ turns out to be consistent with the runtime continuation, because the variable k is bound in the dynamic shift placed directly under the dynamic abstraction and represents the continuation when the abstraction is applied at runtime.

The second change is in the rule for dynamic applications where dynamic reset is inserted around the residualized let-expression. The third change is in the rule for dynamic shift. Rather than residualizing a dynamic shift, which requires residualization of the corresponding reset, the current continuation is grabbed and it is turned into a dynamic expression via η-expansion. Finally, the rule for dynamic reset is removed since all the shift operations are taken care of during specialization time, and there is no need to residualize reset. (This does not necessarily mean that the result of specialization does not contain any reset expressions. Reset is residualized in the rule for dynamic abstractions and applications.)

These changes not only define a correct specializer but result in a quite powerful one. It can now handle *partially static continuations*. Consider the term $\underline{\mathrm{Lam}}(f,\underline{\mathrm{Lam}}(x,\underline{\mathrm{App}}(\mathrm{Var}(f),\overline{\mathrm{Shift}}(k,\overline{\mathrm{App}}(\mathrm{Var}(k),\overline{\mathrm{App}}(\mathrm{Var}(k),\mathrm{Var}(x)))))))$. (This term is well-annotated in the type system shown in the next section.) When we specialize this term, the continuation k grabbed by $\overline{\mathrm{Shift}}(k,\cdots)$ is partially static: we know that the first thing to do when k is applied is to pass its argument to f, but the computation that should be performed after that is unknown. It is the continuation when $\underline{\mathrm{Lam}}(x,\cdots)$ is applied to an argument. Even in this case, $\mathcal{P}_5$ can expand this partial continuation into the result of specialization. By naming the unknown continuation h, $\mathcal{P}_5$ produces the following output (after removing unnecessary dynamic shift and inlining the residualized let-expressions):

$$\begin{aligned}&\mathrm{lam}(\lambda f.\,\mathrm{lam}(\lambda x.\,\mathrm{shift}(\lambda h.\\&\quad\mathrm{reset}(\mathrm{app}(\mathrm{var}(h),\mathrm{app}(\mathrm{var}(f),\mathrm{reset}(\mathrm{app}(\mathrm{var}(h),\mathrm{app}(\mathrm{var}(f),\mathrm{var}(x)))))))))))\ .\end{aligned}$$

Observe that the partial continuation $\mathrm{reset}(\mathrm{app}(\mathrm{var}(h),\mathrm{app}(\mathrm{var}(f),\cdot)))$ is expanded twice in the result. If f were static, we could have been able to perform further specialization, exploiting the partially static information of the continuation.

On the other hand, the above changes cause an interesting side-effect to the result of specialization: all the residualized lambda abstractions now have a 'standardized' form $\mathrm{lam}(\lambda x.\,\mathrm{shift}(\lambda k.\,\cdots))$ (and this is the only place where shift is residualized). In particular, even when we specialize $\underline{\mathrm{Lam}}(x,W)$ where shift is not used during the evaluation of W, the residualized abstraction has typically the form $\mathrm{lam}(\lambda x.\,\mathrm{shift}(\lambda k.\,\mathrm{reset}(\mathrm{app}(\mathrm{var}(k),M))))$ where k does not occur free in M. (If let-expressions are inserted, the result becomes somewhat more complicated.) If we used $\mathcal{P}_3$ instead, we would have obtained the equivalent but simpler result: $\mathrm{lam}(\lambda x.M)$. In other words, $\mathcal{P}_5$ is not a conservative extension of $\mathcal{P}_3$.

A question then is whether it is possible to obtain the latter result on the fly using $\mathcal{P}_5$ with some extra work. We expect that it is not likely. As long as a simple type-based binding-time analysis is employed, it is impossible to tell if the execution of the body of a dynamic abstraction includes any shift operations. So, unless we introduce some extra mechanisms to keep track of this information, there is no way to avoid the insertion of a dynamic shift in the rule for dynamic abstractions. Then, rather than making the specializer complicated, we would employ a simple post-processing to remove unnecessary shift expressions.

5.9 TYPE SYSTEM FOR SHIFT AND RESET

Since our proof technique relies on the logical relations, we need to define a type system for the call-by-value λ-calculus with shift and reset to prove the correctness of $\mathcal{P}_5$. In this section, we briefly review Danvy and Filinski's type system [6]. More thorough explanation is found in [3, 6].

In the presence of first-class (delimited) continuations, we need to explicitly specify the types of continuations and the final result. For this purpose, Danvy and Filinski use a judgment of the form $A, \alpha \vdash M : \tau, \beta \; [W]$. It reads: under the type assumption A, an expression M has a type τ in a continuation of type $\tau \rightsquigarrow \alpha$ and the final result is of type β. Since we use this type system as the static part of our binding-time analysis, we decorate it with $[W]$ to indicate that M is annotated as W. If M does not contain any shift operations, the types α and β are always the same, namely, the *Answer* type. In the presence of shift and reset, however, they can be different and of any type.

The type of functions also needs to include the types of continuations and the final result. It has the form: $\sigma/\alpha \to \tau/\beta$. It is a type of functions that receive an argument of type σ and returns a value of type τ to a continuation of type $\tau \rightsquigarrow \alpha$ and the final result is of type β. As a result, types are specified as follows:

$$\tau \quad = \quad d \mid \tau/\tau \to \tau/\tau \;\; .$$

Here goes the type system:

$$A[n : \tau], \alpha \vdash \mathrm{Var}(n) : \tau, \alpha \; [\mathrm{Var}(n)]$$

$$\frac{A[n : \sigma], \alpha \vdash M : \tau, \beta \; [W]}{A, \delta \vdash \mathrm{Lam}(n, M) : \sigma/\alpha \to \tau/\beta, \delta \; [\overline{\mathrm{Lam}}(n, W)]} \qquad \frac{A, \sigma \vdash M : \sigma, \tau \; [W]}{A, \alpha \vdash \mathrm{Reset}(M) : \tau, \alpha \; [\overline{\mathrm{Reset}}(W)]}$$

$$\frac{\begin{array}{l} A, \delta \vdash M_1 : \sigma/\alpha \to \tau/\varepsilon, \beta \; [W_1] \\ A, \varepsilon \vdash M_2 : \sigma, \delta \; [W_2] \end{array}}{A, \alpha \vdash \mathrm{App}(M_1, M_2) : \tau, \beta \; [\overline{\mathrm{App}}(W_1, W_2)]} \qquad \frac{A[n : \tau/\delta \to \alpha/\delta], \sigma \vdash M : \sigma, \beta \; [W]}{A, \alpha \vdash \mathrm{Shift}(n, M) : \tau, \beta \; [\overline{\mathrm{Shift}}(n, W)]}$$

The above type system is a generalization of the standard type system where types of continuations are made explicit. In Section 5.6, the result type of continuations and the type of final results were always d. In the above type system, it means that a judgment had always the form $A, d \vdash M : \tau, d \; [W]$ and the function type had always the form $\sigma/d \to \tau/d$. So if we write them as $A \vdash M : \tau \; [W]$ and $\sigma \to \tau$, respectively, we obtain exactly the same type system as the one for the ordinary λ-calculus (the three static rules shown in Section 5.3).

The dynamic rules can be obtained by simply replacing all the static function types with d (and types that occur within the function type). The dynamic rules are as follows:

$$\dfrac{A[n:d],d \vdash M : d,d\ [W]}{A,\delta \vdash \mathrm{Lam}(n,M) : d,\delta\ [\underline{\mathrm{Lam}}(n,W)]} \qquad \dfrac{A[n:d],\sigma \vdash M : \sigma,\beta\ [W]}{A,d \vdash \mathrm{Shift}(n,M) : d,\beta\ [\underline{\mathrm{Shift}}(n,W)]}$$

$$\dfrac{A,\delta \vdash M_1 : d,\beta\ [W_1] \quad A,d \vdash M_2 : d,\delta\ [W_2]}{A,d \vdash \mathrm{App}(M_1,M_2) : d,\beta\ [\underline{\mathrm{App}}(W_1,W_2)]}$$

5.10 LOGICAL RELATIONS FOR SHIFT AND RESET

In this section, we define the logical relations for the call-by-value λ-calculus with shift and reset, which are used to prove the correctness of the specializer $\mathcal{P}_5$ presented in Section 5.8. They are the generalization of the logical relations in Section 5.6 in that the types of the final result and the result of continuations are not restricted to d.

$$\begin{array}{lcl}
(M,M') \in R_d & \iff & I_1\,[\![\downarrow_n M]\!]\,\rho_{id} \sim_{\mathrm{v}} M' \text{ for any large } n \\
(M,M') \in R_{\sigma/\alpha\to\tau/\beta} & \iff & \forall (V,V') \in R_\sigma . \forall (\lambda v.K, \lambda v'.K') \models \tau \rightsquigarrow \alpha . \\
 & & (\langle(\lambda v.K)\,(M\,V)\rangle, \langle(\lambda v'.K')\,(M'\,V')\rangle) \in R_\beta
\end{array}$$

where $(\lambda v.K, \lambda v'.K') \models \tau \rightsquigarrow \alpha$ is simultaneously defined as follows:

$$(\lambda v.K, \lambda v'.K') \models \tau \rightsquigarrow \alpha \quad \iff \quad \forall (V,V') \in R_\tau .\ (\langle(\lambda v.K)\,V\rangle, \langle(\lambda v'.K')\,V'\rangle) \in R_\alpha$$

Then, the correctness of the specializer is stated as follows:

Theorem 5.9. *If* $A,\alpha \vdash M : \tau,\beta\ [W]$, $(\rho,\rho') \models A$, *and* $(\lambda v.K, \lambda v'.K') \models \tau \rightsquigarrow \alpha$, *then* $(\langle(\lambda v.K)\,(\mathcal{P}_5\,[\![W]\!]\,\rho)\rangle, \langle(\lambda v'.K')\,(I_2\,[\![M]\!]\,\rho')\rangle) \in R_\beta$.

By instantiating the theorem to the case where both the environment and the continuation are empty, we obtain the following corollary that establishes the correctness of a direct-style specializer that can handle shift and reset:

Corollary 5.10. *If* $d \vdash M : d,d\ [W]$, *then* $I_2\,[\![\downarrow_0 \langle \mathcal{P}_5\,[\![W]\!]\,\rho_\phi\rangle]\!]\,\rho_{id} \sim_{\mathrm{v}} \langle I_2\,[\![M]\!]\,\rho_\phi\rangle$.

The complete proof of the theorem is found in the technical report [4].

5.11 RELATED WORK

This work extends our earlier work [3] where we presented offline specializers for λ-calculus with shift and reset that produced the output in CPS. The present work is a direct-style account of the previous work, but it contains non-trivial definition of logical relations for shift and reset. We also presented the *online* specializers for the λ-calculus with shift and reset [2]. However, their correctness has not been formally proved.

Thiemann [17] presented an offline partial evaluator for Scheme including call/cc. In his partial evaluator, call/cc is reduced if the captured continuation and the body of call/cc are both static. This is close to our first attempt in Section 5.8.

Our solution is more liberal and reduces more continuation-capturing constructs, but with a side-effect that all the residualized abstractions include a toplevel shift, which could be removed by a simple post-processing. More recently, Thiemann [19] showed a sophisticated effect-based type system to show the equivalence of the continuation-based let-insertion and the state-based let-insertion. His type system captures the information on the let-residualized code as an effect. It might be possible to extend his framework to avoid unnecessary shift at the front of dynamic abstractions on the fly.

The correctness proof for offline specializers using the technique of logical relations appears in Jones et al. [14, Chapter 8]. Wand [20] used it to prove the correctness of an offline specializer for the call-by-name λ-calculus. The present work is a non-trivial extension of his work to cope with delimited continuations. Wand's formulation was based on substitution, but we used the environment-based formulation, which is essentially the same but is more close to the implementation.

Filinski presented normalization-by-evaluation algorithms for the call-by-value λ-calculus [10] and the computational λ-calculus [11]. He showed their correctness denotationally using logical relations. The same framework is extended to the untyped λ-calculus by Filinski and Rohde [12].

The type system used in this paper is due to Danvy and Filinski [6]. A similar type system is studied by Ariola, Herbelin, and Sabry [1], which explicitly mentions the type of continuations.

5.12 CONCLUSION

This paper demonstrated that logical relations can be defined to characterize not only call-by-name higher-order functions but also call-by-value functions as well as delimited continuations. They were used to show the correctness of various offline specializers, including the one for the call-by-value λ-calculus with shift and reset. Along the development, we established the correctness of the continuation-based let-insertion, the shift/reset-based let-insertion, the continuation-based A-normalization, and the shift/reset-based A-normalization.

REFERENCES

[1] Ariola, Z. M., H. Herbelin, and A Sabry "A Type-Theoretic Foundation of Continuations and Prompts," *Proceedings of the ninth ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, pp. 40–53 (September 2004).

[2] 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).

[3] Asai, K. "Offline Partial Evaluation for Shift and Reset," *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '04)*, pp. 3–14 (August 2004).

[4] Asai, K. “Logical Relations for Call-by-value Delimited Continuations,” Technical Report OCHA-IS 06-1, Department of Information Sciences, Ochanomizu University (April 2006).

[5] Bondorf, A., and O. Danvy “Automatic autoprojection of recursive equations with global variables and abstract data types,” *Science of Computer Programming*, Vol. 16, pp. 151–195, Elsevier (1991).

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

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

[8] Danvy, O., and J. L. Lawall “Back to Direct Style II: First-Class Continuations,” *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pp. 299–310 (June 1992).

[9] de Bruijn, N. G. “Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem,” *Indagationes Mathematicae*, Vol. 34, pp. 381–392 (1972).

[10] Filinski, A. “A Semantic Account of Type-Directed Partial Evaluation,” In G. Nadathur, editor, *Principles and Practice of Declarative Programming (LNCS 1702)*, pp. 378–395 (September 1999).

[11] Filinski, A. “Normalization by Evaluation for the Computational Lambda-Calculus,” In S. Abramsky, editor, *Typed Lambda Calculi and Applications (LNCS 2044)*, pp. 151–165 (May 2001).

[12] Filinski, A., and H. K. Rohde “A Denotational Account of Untyped Normalization by Evaluation,” In I. Walukiewicz, editor, *Foundations of Software Science and Computation Structures (LNCS 2987)*, pp. 167–181 (March 2004).

[13] 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).

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

[15] Kameyama, Y., and M. Hasegawa “A Sound and Complete Axiomatization of Delimited Continuations,” *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming (ICFP’03)*, pp. 177–188 (August 2003).

[16] Mitchell, J. C. *Foundations for Programming Languages*, Cambridge: MIT Press (1996).

[17] Thiemann, P. “Towards Partial Evaluation of Full Scheme,” *Proceedings of Reflection’96*, pp. 105–115 (April 1996).

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

[19] Thiemann, P. “Continuation-Based Partial Evaluation without Continuation,” In R. Cousot, editor, *Static Analysis (LNCS 2694)*, pp. 366–382 (June 2003).

[20] 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).