Polymorphic Delimited Continuations

Kenichi Asai[1] and Yukiyoshi Kameyama[2]

[1] Department of Information Science, Ochanomizu University
asai@is.ocha.ac.jp
[2] Department of Computer Science, University of Tsukuba
kameyama@acm.org

Abstract. This paper presents a polymorphic type system for a language with delimited control operators, shift and reset. Based on the monomorphic type system by Danvy and Filinski, the proposed type system allows pure expressions to be polymorphic. Thanks to the explicit presence of answer types, our type system satisfies various important properties, including strong type soundness, existence of principal types and an inference algorithm, and strong normalization. Relationship to CPS translation as well as extensions to impredicative polymorphism are also discussed. These technical results establish the foundation of polymorphic delimited continuations.
Keywords: Type System, Delimited Continuation, Control Operator, CPS Translation, Predicative/Impredicative Polymorphism.

Note. This technical report is an extended version (with proofs) of the paper: "Polymorphic Delimited Continuations", Kenichi Asai and Yukiyoshi Kameyama, Proc. Fifth Asian Symposium on Programming Languages and Systems, Lecture Notes in Computer Science, 2007.

1 Introduction

Delimited continuation operators enable us to manipulate control of programs in a concise manner without transforming them into continuation-passing style (CPS). In particular, shift and reset, introduced by Danvy and Filinski [5], have strong connection to CPS, and thus most of the control effects compatible with CPS can be expressed using shift and reset [7]. They have been used, for example, to program backtracking [5], A-normalization in direct style [1], let-insertion in partial evaluation [1], and type-safe "printf" in direct style [2].

Despite the increasing interest in the use of delimited continuations in typed programming languages, there has been little work that investigates their basic properties without sacrificing their expressive power. The original type system for shift and reset by Danvy and Filinski [4] is the only type system that allows modification of answer types but is restricted to monomorphic types. Polymorphism in the presence of `call/cc` has been discussed in the context of ML [10] but strong type soundness [21] does not hold for their type system. Gunter, Rémy, and Riecke [9] proposed typed `cupto` operator with strong type soundness theorem as well as various properties, but their type system is restricted to a fixed answer type for each prompt. As such, none of the above type systems can type check, for instance, the "printf" program written with shift and reset.

To establish the basic properties of shift and reset without sacrificing their expressive power, we present in this paper a polymorphic type system, an extension of the monomorphic

type system by Danvy and Filinski, and show that it satisfies a number of basic properties needed to use them in ordinary programming languages. In particular, we show strong type soundness, existence of principal types and an efficient type inference algorithm, and strong normalization among others. The polymorphism does not break the semantic foundation of the original monomorphic type system: CPS translation is naturally defined for our polymorphic calculus and preserves types and equivalence. Because of its natural connection to CPS, our framework can be extended to a calculus with impredicative polymorphism [8].

Unrestricted polymorphism in the presence of control operators leads to an unsound type system [10]. We introduce and employ a new criteria called "purity" restriction instead of more restrictive value restriction. An expression is said to be pure if it has no control-effects [18]. By allowing pure expressions to be polymorphic, an interesting non-value term can be given a polymorphic type.

Based on these results, we have implemented a prototypical type inference algorithm, and applied it to many interesting programs to obtain their principal types.

The rest of this paper is organized as follows: Section 2 illustrates a few programming examples to give intuition about the type structure for shift and reset. In Section 3, we formalize a predicatively polymorphic calculus for shift and reset, and prove its properties such as type soundness. We then study a CPS translation for our calculus in Section 4. In Section 5, we extend our study to cover impredicative polymorphism under two evaluation strategies. In Section 6, we compare our work with related work and give conclusion.

Proofs of the theorems in this paper can be found in the appendix.

2 Programming Examples

Polymorphism is inevitable in programming [17]. A simple example of polymorphism is found in list manipulating functions: a reverse function works for a list of elements of any type. In this section, we introduce the control operators, `shift` and `reset`, and show examples of polymorphism that involves control operators.

2.1 List Append: Answer Type Modification

Consider the following program [4] written in OCaml syntax:

```
let rec append lst = match lst with
    [] -> shift (fun k -> k)
  | a :: rest -> a :: append rest
```

This program is a curried version of list append, written with control operators. Here, `shift` captures its current continuation and passes it to its argument (typically a one-argument function `fun k ->` ...) in the empty context. Unlike `callcc`, however, continuations are captured only up to its enclosing `reset` (hence called *delimited* continuations).

When `append` is invoked in a delimited context as follows:

```
let append123 = reset (fun () -> append [1; 2; 3])
```

`append` recursively stores each element of its argument into the control stack. When all the elements are stacked, the control stack could be thought of as a term with a hole: $1 :: 2 :: 3 :: \bullet$, waiting for the value for the `[]` case. Then, `shift (fun k -> k)` captures it,

turns it into an ordinary function $\lambda x.1 :: 2 :: 3 :: x$, and returns it. The returned continuation `append123` is the partially applied append function: given a list, it appends 1, 2, and 3 to it in the reversed order.

When `shift` is used in a program, it typically has an impact on the answer type of its enclosing context. Before `shift (fun k -> k)` is executed, the context $1 :: 2 :: 3 :: \bullet$ was supposed to return a list (given a list for $\bullet$). In other words, the answer type of this context was a list. After `shift (fun k -> k)` is executed, however, what is returned is the captured continuation $\lambda x.1 :: 2 :: 3 :: x$ of type `int list -> int list`. In other words, execution of `shift (fun k -> k)` modifies the answer type from `'a list` to `'a list -> 'a list`, where `'a` is the type of the elements of the list.

To accommodate this behavior, Danvy and Filinski used a function type of the form `S / A -> T / B` [4]. It is the type of a function from `S` to `T`, but modifies the answer type from `A` to `B` when applied. Using this notation, `append` has the type `'a list / 'a list -> 'a list / ('a list -> 'a list)` for all `'a`: given a list of type `'a list`, `append` returns a list of type `'a list` to its immediate context; during this process, however, the answer type of the context is modified from `'a list` to `'a list -> 'a list`.

Gunter, Rémy, and Riecke mention the type of context (prompt) in their type system [9]. However, they fix the answer type and do not take the answer type modification into account, limiting the use of control operators. To characterize the full expressive power of `shift` and `reset`, it is necessary to cope with *two* answer types together with polymorphism.

2.2 List Prefix: Answer Type Polymorphism

Once answer types are included in a function type, polymorphism becomes more important in programming. First of all, the conventional function type `S -> T` is regarded as polymorphic in the answer type [18]: `S / 'a -> T / 'a` for a new type variable `'a`. This indicates that even a simple, apparently monomorphic, function like:

```
let add1 x = x + 1
```

has to be treated as polymorphic in the answer type. Otherwise, it cannot be used in different contexts as in:

```
reset (fun () -> add1 2; ()); reset (fun () -> add1 3; true)
```

The first occurrence of `add1` is used at type `int / unit -> int / unit` whereas the second one at type `int / bool -> int / bool`. To unify them, `add1` has to be given a polymorphic type: `int / 'a -> int / 'a`.

Answer type polymorphism plays an important role in captured continuations, too. Consider the following program [3]:

```
let rec visit lst = match lst with
    [] -> shift (fun h -> [])
  | a :: rest -> a :: shift (fun k -> (k []) :: reset (k (visit rest)))

let rec prefix lst = reset (visit lst)
```

When applied to a list, e.g., `[1; 2 3]`, `prefix` returns a list of its prefixes: `[[1]; [1; 2]; [1; 2; 3]]`. In this example, there are two occurrences of `shift`. Intuitively, the continuation captured by the second `shift` represents consing of elements read so far. It is applied

twice: once to an empty list to construct a current prefix and once to construct a list of longer prefixes. Finally, the first occurrence of `shift` initiates the construction of prefixes by returning an empty list of type `'a list list`, discarding the current continuation.

It is important that the captured continuation `k` is polymorphic in its answer type. A closer look at the function reveals that `k` is used in two different contexts: the first occurrence of `k` has type `'a list / 'a list list -> 'a list / 'a list list` whereas the second one has type `'a list / 'a list -> 'a list / 'a list`. This demonstrates that without answer type polymorphism in the captured continuations, the above program does not type check.

2.3 Printf

Finally, we present a type-safe printf program written in direct style with `shift` and `reset` (detailed in [2]). Given a representation of types:

```
let int x = string_of_int x
let str (x : string) = x
```

the following program achieves the behavior of printf in a type-safe manner:

```
let % to_str = shift (fun k -> fun x -> k (to_str x))
let sprintf p = reset p
```

Namely, the following programs are all well-typed:

```
sprintf (fun () -> "Hello world!")
sprintf (fun () -> "Hello " ^ % str ^ "!") "world"
sprintf (fun () -> "The value of " ^ % str ^ " is " ^ % int) "x" 3
```

and give `"Hello World!"` for the first two and `"The value of x is 3"` for the last. Depending on `%` appearing in the formatting text, `sprintf` returns a different type of values.

The dependent behavior of `sprintf` is well understood by examining its type: `(unit / string -> string / 'a) -> 'a`. The formatting text is represented as a thunk that modifies the final answer type into `'a` according to the occurrence of `%`. Then, the type of the return value of `sprintf` is polymorphic to this `'a`. The dependent behavior of `sprintf` is only achievable through the support of both the answer type modification and polymorphism.

3 Predicative Polymorphism with Shift/Reset

We now introduce polymorphic typed calculi for shift and reset, and study their properties such as type soundness. Following the literature, we distinguish two versions of polymorphism: *predicative* polymorphism (let-polymorphism) found in ML and *impredicative* polymorphism which is based on the second order lambda calculus (Girard's System F [8]). In this section, we give the predicative version $\lambda_{let}^{s/r}$. The impredicative version will be given in later sections.

$$
\begin{array}{rl}
v ::= & c \mid x \mid \lambda x.e \mid \texttt{fix}\ f.x.e & \text{value} \\
e ::= & v \mid e_1 e_2 \mid \mathcal{S}k.e \mid \langle e \rangle \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & \\
& \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \text{expression} \\
\alpha, \beta, \gamma, \delta ::= & t \mid b \mid (\alpha/\gamma \to \beta/\delta) & \text{monomorphic type} \\
A ::= & \alpha \mid \forall t.A & \text{polymorphic type}
\end{array}
$$

Fig. 1. Syntax of $\lambda_{let}^{s/r}$.

3.1 Syntax and Operational Semantics

We assume that the sets of constants (denoted by c), variables (denoted by x, y, k, f), type variables (denoted by t), and basic types (denoted by b) are mutually disjoint, and that each constant is associated with a basic type. We assume `bool` is a basic type which has constants `true` and `false`.

The syntax of $\lambda_{let}^{s/r}$ is given by BNF in Figure 1. A value is either a constant, a variable, a lambda abstraction, or a fixpoint expression $\texttt{fix}\ f.x.e$ which represents a recursive function defined by the equation $f(x) = e$. The variables f and x are bound in $\texttt{fix}\ f.x.e$. An expression is either a value, an application, a shift expression, a reset expression, a let expression, or a conditional. The expressions $\mathcal{S}k.e$ and $\langle e \rangle$, resp., correspond to OCaml expressions `shift (fun k -> e)` and `reset (fun () -> e)`, resp. Types are similar to those in ML except that the function type is now annotated with answer types as $(\alpha/\gamma \to \beta/\delta)$. Free and bound variables (type variables, resp.) in expressions (types, resp.) are defined as usual, and $\texttt{FTV}(\alpha)$ denotes the set of free type variables in α.

We give call-by-value operational semantics for $\lambda_{let}^{s/r}$. First we define evaluation contexts (abbreviated as e-contexts), pure e-contexts, and redexes as follows:

$$
\begin{array}{rll}
E ::= & [\,] \mid vE \mid Ee \mid \langle E \rangle \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e & \text{e-context} \\
F ::= & [\,] \mid vF \mid Fe \mid \texttt{let}\ x = F\ \texttt{in}\ e \mid \texttt{if}\ F\ \texttt{then}\ e\ \texttt{else}\ e & \text{pure e-context} \\
R ::= & (\lambda x.e)v \mid \langle v \rangle \mid \langle F[\mathcal{S}k.e] \rangle \mid \texttt{let}\ x = v\ \texttt{in}\ e & \\
& \mid \texttt{if true then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{if false then}\ e_1\ \texttt{else}\ e_2 & \\
& \mid (\texttt{fix}\ f.x.e)v & \text{redex}
\end{array}
$$

A pure e-context F is an evaluation context such that no reset encloses the hole. Therefore, in the redex $\langle F[\mathcal{S}k.e] \rangle$, the outermost reset is guaranteed to be the one corresponding to this shift, i.e., no reset exists inbetween.

A one-step *evaluation* in $\lambda_{let}^{s/r}$ is $E[R] \rightsquigarrow E[e]$ where $R \rightsquigarrow e$ is an instance of reductions in Figure 2 where $e[v/x]$ denotes the ordinary capture-avoiding substitution. For example, `prefix [1; 2]` is reduced as follows. (We use `fix` implicitly through recursion, and assume

$$
\begin{aligned}
(\lambda x.e)v &\rightsquigarrow e[v/x] \\
(\texttt{fix}\ f.x.e)v &\rightsquigarrow e[\texttt{fix}\ f.x.e/f][v/x] \\
\langle v \rangle &\rightsquigarrow v \\
\langle F[\mathcal{S}k.e] \rangle &\rightsquigarrow \langle \texttt{let}\ k = \lambda x.\langle F[x] \rangle\ \texttt{in}\ e \rangle \\
\texttt{let}\ x = v\ \texttt{in}\ e &\rightsquigarrow e[v/x] \\
\texttt{if true then}\ e_1\ \texttt{else}\ e_2 &\rightsquigarrow e_1 \\
\texttt{if false then}\ e_1\ \texttt{else}\ e_2 &\rightsquigarrow e_2
\end{aligned}
$$

Fig. 2. Reduction rules for $\lambda_{let}^{s/r}$

that lists and other constructs are available in the language).

$$
\begin{aligned}
& \texttt{prefix [1; 2]} \\
\rightsquigarrow\ & \langle 1 :: \mathcal{S}k.(k\,\texttt{[]} :: \langle k\,(\texttt{visit [2]})\rangle)\rangle \\
\rightsquigarrow\ & \langle \texttt{let}\ k = \lambda x.\langle 1 :: x\rangle\ \texttt{in}\ k\,\texttt{[]} :: \langle k\,(\texttt{visit [2]})\rangle\rangle \\
\rightsquigarrow\ & \langle (\lambda x.\langle 1 :: x\rangle)\,\texttt{[]} :: \langle (\lambda x.\langle 1 :: x\rangle)(\texttt{visit [2]})\rangle\rangle \\
\rightsquigarrow^{+}\ & \langle \texttt{[1]} :: \langle (\lambda x.\langle 1 :: x\rangle)(2 :: \mathcal{S}k.(k\,\texttt{[]} :: \langle k\,(\texttt{visit []})\rangle))\rangle\rangle \\
\rightsquigarrow\ & \langle \texttt{[1]} :: \langle \texttt{let}\ k = \lambda x.\langle (\lambda x.\langle 1 :: x\rangle)(2 :: x)\rangle\ \texttt{in}\ k\,\texttt{[]} :: \langle k\,(\texttt{visit []})\rangle\rangle\rangle \\
\rightsquigarrow\ & \langle \texttt{[1]} :: \langle (\lambda x.\langle (\lambda x.\langle 1 :: x\rangle)(2 :: x)\rangle)\,\texttt{[]} :: \langle (\lambda x.\langle (\lambda x.\langle 1 :: x\rangle)(2 :: x)\rangle)(\texttt{visit []})\rangle\rangle\rangle \\
\rightsquigarrow^{+}\ & \langle \texttt{[1]} :: \langle \texttt{[1; 2]} :: \langle (\lambda x.\langle (\lambda x.\langle 1 :: x\rangle)(2 :: x)\rangle)(\mathcal{S}h.\texttt{[]})\rangle\rangle\rangle \\
\rightsquigarrow\ & \langle \texttt{[1]} :: \langle \texttt{[1; 2]} :: \texttt{let}\ h = \lambda x.\langle (\lambda x.\langle (\lambda x.\langle 1 :: x\rangle)(2 :: x)\rangle)x\rangle\ \texttt{in []}\rangle\rangle \\
\rightsquigarrow\ & \langle \texttt{[1]} :: \langle \texttt{[1; 2]} :: \texttt{[]}\rangle\rangle\ \rightsquigarrow^{+}\ \texttt{[[1]; [1; 2]]}
\end{aligned}
$$

The notion of *reduction* $\rightsquigarrow$ is defined as the compatible closure[3] of those in Figure 2, and $\rightsquigarrow^*$ (and $\rightsquigarrow^+$, resp.) denotes the reflexive-transitive (transitive, resp.) closure of $\rightsquigarrow$.

3.2 Type System

We begin with Danvy and Filinski's monomorphic type system for shift and reset [4]. Since the evaluation of an expression with shift and reset may modify answer types, a type judgment in their type system involves not only a type of an expression being typed, but also answer types before and after evaluation. Symbolically, a judgment takes the form:

$$\Gamma;\ \alpha \vdash e : \tau;\ \beta$$

which means that, under the type context Γ, the expression e has type τ and the evaluation of e changes the answer type from α to β. A rationale behind this formulation is that, the CPS counterpart of e has type $(\tau^* \to \alpha^*) \to \beta^*$ under the type context Γ^* in the simply typed lambda calculus, where $(\_)^*$ is the CPS translation for types and type contexts defined in the next section.

Introducing polymorphism into their type system is, however, not straightforward since the subject reduction property fails for the system with unrestricted uses of let-polymorphism

[3] A binary relation is compatible if it is closed under term-formation, for instance, whenever e_1 and e_2 are related by this relation, $\lambda x.e_1$ and $\lambda x.e_2$ are related.

and side effects such as references and control effects. In the literature, there are many proposals to solve this problem by restricting the let-expression `let` $x = e_1$ `in` e_2 or by changing its operational semantics, some of which are:

- Value restriction [20]: e_1 must be a value.
- Weak type variables [19]: the type variable in the type of e_1 can be generalized only when it is not related to side effects.
- Polymorphism by name [15]: the evaluation of e_1 is postponed until x is actually used in e_2, thus enforcing the call-by-name evaluation to e_1.

We take an alternative approach: we restrict that e_1 in `let` $x = e_1$ `in` e_2 must be free from control effects, that is, *pure*. Intuitively, an expression is pure when it is polymorphic in answer types.[4] In Danvy and Filinski's type system, we can define that e is pure if the judgment $\Gamma;\ \alpha \vdash e : \tau;\ \alpha$ is derivable for any type α. Typical examples of pure expressions are values but the expression $\langle e \rangle$ is also pure, since all control effects in e are delimited by reset. To represent purity of expressions, we introduce a new judgment form $\Gamma \vdash_p e : \tau$.

Now let us formally define the type system of $\lambda_{let}^{s/r}$. A *type context* (denoted by Γ) is a finite list of the form $x_1 : A_1, \cdots, x_n : A_n$ where the variables $x_1, \cdots, x_n$ are mutually distinct, and $A_1, \cdots, A_n$ are (polymorphic) types. *Judgments* are either one of the following forms:

$$\begin{array}{rl} \Gamma \vdash_p e : \tau & \text{judgment for pure expression} \\ \Gamma;\ \alpha \vdash e : \tau;\ \beta & \text{judgment for general expression} \end{array}$$

Figure 3 lists the type inference rules of $\lambda_{let}^{s/r}$ where $\tau \leq A$ in the rule (var) means the instantiation of type variables by monomorphic types. Namely, if $A \equiv \forall t_1. \cdots \forall t_n.\rho$ for some monomorphic type ρ, then $\tau \equiv \rho[\sigma_1, \cdots, \sigma_n/t_1, \cdots, t_n]$ for some monomorphic types $\sigma_1, \cdots, \sigma_n$. The type $\mathtt{Gen}(\sigma; \Gamma)$ in the rule (let) is defined by $\forall t_1. \cdots \forall t_n.\sigma$ where $\{t_1, \cdots, t_n\} = \mathtt{FTV}(\sigma) - \mathtt{FTV}(\Gamma)$.

The type inference rules are a natural extension of the monomorphic type system by Danvy and Filinski [4]. Pure expressions are defined by one of the rules (fix), (fun), or (reset).[5] They can be freely turned into general expressions through the rule (exp). Pure expressions can be used polymorphically through the rule (let). It generalizes the standard let-polymorphism found in ML. We can allow a let expression `let` $x = e_1$ `in` e_2 even when e_1 is not pure, in which case it is macro-expanded to $(\lambda x.e_2)e_1$ where e_1 is treated monomorphically. Finally, the rule (shift) is extended to cope with the answer type polymorphism of captured continuations: k is given a polymorphic type $\forall t.(\tau/t \to \alpha/t)$.

Examples. We show the principal types for the examples shown in Section 2.

Using the type inference rules (augmented with rules for lists, etc.), we can deduce that `append` (rewritten with `fix`) has type `'a list / 'b -> 'a list / ('a list -> 'b)`,[6]

[4] Thielecke studied the relationship between answer type polymorphism and the absence of control in depth [18].

[5] We could have introduced a more general rule such as: if $\Gamma;\ t \vdash e : \tau;\ t$ is derivable for $t \notin \mathtt{FTV}(\Gamma, \tau)$, then $\Gamma \vdash_p e : \tau$. It would then allow expressions that are not syntactically values nor reset expressions but in fact pure, such as $\mathcal{S}k.k3$. We did not take this approach, because we can always insert reset around pure expressions to make them syntactically pure.

[6] This is the principal type for `append`. In the typical case where the call to `append` is immediately enclosed by `reset` as is the case for `append123`, `'b` is instantiated to `'a list`.

$$\frac{(x : A \in \Gamma \text{ and } \tau \leq A)}{\Gamma \vdash_p x : \tau}\ \text{var} \qquad \frac{(c \text{ is a constant of basic type } b)}{\Gamma \vdash_p c : b}\ \text{const}$$

$$\frac{\Gamma, f : (\sigma/\alpha \to \tau/\beta), x : \sigma;\ \alpha \vdash e : \tau;\ \beta}{\Gamma \vdash_p \texttt{fix}\ f.x.e : (\sigma/\alpha \to \tau/\beta)}\ \text{fix} \qquad \frac{\Gamma, x : \sigma;\ \alpha \vdash e : \tau;\ \beta}{\Gamma \vdash_p \lambda x.e : (\sigma/\alpha \to \tau/\beta)}\ \text{fun}$$

$$\frac{\Gamma;\ \gamma \vdash e_1 : (\sigma/\alpha \to \tau/\beta);\ \delta \quad \Gamma;\ \beta \vdash e_2 : \sigma;\ \gamma}{\Gamma;\ \alpha \vdash e_1 e_2 : \tau;\ \delta}\ \text{app} \qquad \frac{\Gamma \vdash_p e : \tau}{\Gamma;\ \alpha \vdash e : \tau;\ \alpha}\ \text{exp}$$

$$\frac{\Gamma, k : \forall t.(\tau/t \to \alpha/t);\ \sigma \vdash e : \sigma;\ \beta}{\Gamma;\ \alpha \vdash \mathcal{S}k.e : \tau;\ \beta}\ \text{shift} \qquad \frac{\Gamma; \sigma \vdash e : \sigma;\ \tau}{\Gamma \vdash_p \langle e \rangle : \tau}\ \text{reset}$$

$$\frac{\Gamma \vdash_p e_1 : \sigma \quad \Gamma, x : \texttt{Gen}(\sigma; \Gamma);\ \alpha \vdash e_2 : \tau;\ \beta}{\Gamma;\ \alpha \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau;\ \beta}\ \text{let}$$

$$\frac{\Gamma;\ \sigma \vdash e_1 : \texttt{bool};\ \beta \quad \Gamma;\ \alpha \vdash e_2 : \tau;\ \sigma \quad \Gamma;\ \alpha \vdash e_3 : \tau;\ \sigma}{\Gamma;\ \alpha \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau;\ \beta}\ \text{if}$$

Fig. 3. Type Inference Rules of $\lambda^{s/r}_{let}$.

where `'a list -> 'b` is a shorthand for `'a list / 'c -> 'b / 'c` for a new type variable `'c`. Given this type, the type of `append123`, i.e., `reset (fun () -> append [1; 2; 3])`, becomes `int list -> int list` (or `int list / 'c -> int list / 'c`). Since it is pure, `append123` can be given a polymorphic type in its answer type `'c`. Notice that `append123` is not bound to a value but an effectful expression enclosed by reset. If we employed value restriction, `append123` could not be polymorphic, and thus could only be used in a context with a fixed answer type.

Next, the principal type for `visit` is `'a list / 'b -> 'a list / 'b list`.[7] To deduce this type, we need to use the rule (shift) to give `k` a polymorphic type in its answer type. Then, the type of `prefix` becomes `'a list -> 'a list list`. In other words, it accepts a list of any type `'a`. Since it is pure (that is, answer type polymorphic), it can be used in any context.

Finally, the principal type for `%` is somewhat complicated:

```
('a / 'p -> 's / 'q) / 't -> 's / ('a / 'p -> 't / 'q)
```

In the typical case where `to_str` is pure (`'p`=`'q`) and has type `'a -> string`, and the output `'t` is `string`, the above type becomes:

```
('a -> string) / string -> string / ('a -> string)
```

This type describes the behavior of `%`: given a representation of a type (of type `'a -> string`), it changes the answer type from `string` to a function that receives a value of the specified type `'a`. Then, `sprintf` returns a function of this final answer type, thus accepting an argument depending on the occurrence of `%`.

3.3 Properties

We have introduced the polymorphic calculus $\lambda^{s/r}_{let}$ with shift and reset. We claim that our calculus provides a good foundation for studying the interaction between polymorphism and delimited continuations. To support this claim, we prove the following properties:

[7] Again, `'b` is typically instantiated to `'a list`.

– Subject reduction (type preservation).
– Progress and unique decomposition.
– Principal types and existence of a type inference algorithm.
– Preservation of types and equality through CPS translation.
– Confluence.
– Strong normalization for the subcalculus without `fix`.

We first show type soundness, i.e., subject reduction and progress.

Theorem 1 (Subject Reduction). *If $\Gamma;\ \alpha \vdash e_1 : \tau;\ \beta$ is derivable and $e_1 \rightsquigarrow^* e_2$, then $\Gamma;\ \alpha \vdash e_2 : \tau;\ \beta$ is derivable. Similarly, if $\Gamma \vdash_p e_1 : \tau$ is derivable and $e_1 \rightsquigarrow^* e_2$, then $\Gamma \vdash_p e_2 : \tau$ is derivable.*

The above theorem not only assures that a well-typed program does not go wrong (so-called *weak* type soundness [21]) but also guarantees that the evaluated term has the same type as the original term (*strong* type soundness [21]). This is the consequence of having answer types explicitly in our type system. We need three lemmas to prove this theorem.

Lemma 1 (Weakening of Type Context). *Suppose $\Gamma_1 \subset \Gamma_2$ and Γ_2 is a valid type context. If $\Gamma_1;\ \alpha \vdash e : \sigma;\ \beta$ is derivable, then $\Gamma_2;\ \alpha \vdash e : \sigma;\ \beta$ is derivable. Similarly for $\Gamma_1 \vdash_p e : \sigma$.*

Lemma 2 (Substitution for Monomorphic Variables). *Suppose $\Gamma_1 \subset \Gamma_2$, Γ_2 is a valid type context, and $\Gamma_1 \vdash_p v : \sigma$ is derivable.*

If $\Gamma_2, x : \sigma;\ \alpha \vdash e : \tau;\ \beta$ is derivable, then $\Gamma_2;\ \alpha \vdash e[v/x] : \tau;\ \beta$ is derivable. Similarly, if $\Gamma_2, x : \sigma \vdash_p e : \tau$ is derivable, then $\Gamma_2 \vdash_p e[v/x] : \tau$ is derivable.

Lemma 3 (Substitution for Polymorphic Variables). *Suppose $\Gamma_1 \subset \Gamma_2$, Γ_2 is a valid type context, and $\Gamma_1 \vdash_p v : \sigma$ is derivable.*

If $\Gamma_2, x : \texttt{Gen}(\sigma; \Gamma_1);\ \alpha \vdash e : \tau;\ \beta$ is derivable, then $\Gamma_2;\ \alpha \vdash e[v/x] : \tau;\ \beta$ is derivable. Similarly for $\Gamma_2, x : \texttt{Gen}(\sigma; \Gamma_1) \vdash_p e : \tau$.

We next prove the progress property, which states that evaluation of a program does not get stuck. Although a program is usually defined as an expression with no free variables, we need to refine it, since, for instance, $\mathcal{S}k.k3$ cannot be reduced further due to the absence of an enclosing reset. Here, we define a program to be an expression with a toplevel reset of the form $\langle e \rangle$ which has no free variables.

Theorem 2 (Progress and Unique Decomposition). *If $\vdash_p \langle e \rangle : \tau$ is derivable, then either e is a value, or $\langle e \rangle$ can be uniquely decomposed into the form $E[R]$ where E is an evaluation context and R is a redex.*

By Theorems 1 and 2, we can conclude that our type system is sound (Type Soundness).

Although our type system may look rather complex, we can smoothly extend Hindley-Milner type inference algorithm W to accommodate $\lambda_{let}^{s/r}$. The extended algorithm W' takes two arguments as its inputs: Γ (for a valid context) and e (for a raw expression) such that all free variables in e are contained in Γ. Then, W' either fails or returns a tuple $(\theta;\ \alpha, \tau, \beta)$ where θ is a substitution for type variables, and α, τ, and β are types.

Theorem 3 (Principal Type and Type Inference). *We can construct a type inference algorithm W' for $\lambda_{let}^{s/r}$ such that:*

1. W' always terminates.

2. if W' returns $(\theta;\ \alpha, \tau, \beta)$, then $\Gamma\theta;\ \alpha \vdash e : \tau;\ \beta$ is derivable. Moreover, for any $(\theta';\ \alpha', \tau', \beta')$ such that $\Gamma\theta';\ \alpha' \vdash e : \tau';\ \beta'$ is derivable, $(\Gamma\theta', \alpha', \tau', \beta') \equiv (\Gamma\theta, \alpha, \tau, \beta)\phi$ for some substitution ϕ.

3. if W' fails, then $\Gamma\theta;\ \alpha \vdash e : \tau;\ \beta$ is not derivable for any $(\theta;\ \alpha, \tau, \beta)$.

We have implemented a prototypical type inference algorithm system for our language based on this theorem. The principal types shown in Section 3.2 are all inferred by it.

Finally, we can show confluence for $\lambda_{let}^{s/r}$, and strong normalization for the subcalculus without `fix`. This is in contrast to `cupto` operator, where strong normalization does not hold.[8]

Theorem 4 (Confluence and Strong Normalization).

1. The reduction $\rightsquigarrow$ in $\lambda_{let}^{s/r}$ is confluent.

2. The reduction $\rightsquigarrow$ in $\lambda_{let}^{s/r}$ without `fix` is strongly normalizing.

4 CPS translation of $\lambda_{let}^{s/r}$

The semantics of control operators have often been given through a CPS translation. In their first proposal, Danvy and Filinski gave the precise semantics of shift and reset in terms of a CPS translation [5, 6]. In this section, we show that it can be naturally extended to polymorphic setting.

Harper and Lillibridge [11] were the first to systematically study CPS translations in polymorphic language with control operators. They introduced CPS translations from $F\omega$+`call/cc` to $F\omega$, and proved that, under a condition similar to the value restriction, a call-by-value CPS translation preserves types and semantics (equality). We follow Harper and Lillibridge to give a type-and-equality preserving CPS translation for polymorphic calculi with shift and reset.

The CPS translation for $\lambda_{let}^{s/r}$ is a Plotkin-style, call-by-value translation, and is defined in Figures 4 and 5, where the variables κ, m and n are fresh. The target calculus (the image) of the translation is λ_{let}, the minimum lambda calculus with let-polymorphism and conditional expressions.[9]

The type $(\alpha/\gamma \to \beta/\delta)$ is translated to the type of a function which, given a parameter of type α^* and a continuation of type $\beta^* \to \gamma^*$ returns a value of type δ^*. For instance, the type of the `visit` function (in the `prefix` example) `'a list / 'b -> 'a list / 'b list` is CPS translated to `'a list -> ('a list -> 'b) -> 'b list`.

The translation of reset is the same as that in Danvy and Filinski's. For shift, we use a let-expression rather than substitution, so that the captured continuation $\lambda n\kappa'.\kappa'(\kappa n)$ may be used polymorphically in the body $[\![e]\!](\lambda m.m)$. This is essential to retain enough polymorphism for delimited continuations.

[8] See `http://okmij.org/ftp/Computation/Continuations.html#cupto-nontermination`.

[9] λ_{let} may be obtained from $\lambda_{let}^{s/r}$ by eliminating shift, reset, and answer types α and β in $\Gamma;\ \alpha \vdash e : \tau;\ \beta$ and $(\sigma/\alpha \to \tau/\beta)$. Since all expressions are pure in λ_{let}, we do not distinguish two kinds of judgments.

$$
\begin{aligned}
b^* &= b \quad \text{for a basic type } b \\
t^* &= t \quad \text{for a type variable } t \\
((\alpha/\gamma \to \beta/\delta))^* &= \alpha^* \to (\beta^* \to \gamma^*) \to \delta^* \\
(\forall t.A)^* &= \forall t.A^* \\
(\Gamma, x : A)^* &= \Gamma^*, x : A^*
\end{aligned}
$$

Fig. 4. CPS translation for types and type contexts.

$$
\begin{aligned}
c^* &= c \\
v^* &= v \\
(\lambda x.e)^* &= \lambda x.[\![e]\!] \\
(\mathtt{fix}\ f.x.e)^* &= \mathtt{fix}\ f.x.[\![e]\!] \\
[\![v]\!] &= \lambda\kappa.\kappa v^* \\
[\![e_1 e_2]\!] &= \lambda\kappa.[\![e_1]\!](\lambda m.[\![e_2]\!](\lambda n.mn\kappa)) \\
[\![\mathcal{S}k.e]\!] &= \lambda\kappa.\mathtt{let}\ k = \lambda n\kappa'.\kappa'(\kappa n)\ \mathtt{in}\ [\![e]\!](\lambda m.m) \\
[\![\langle e \rangle]\!] &= \lambda\kappa.\kappa([\![e]\!](\lambda m.m)) \\
[\![\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2]\!] &= \lambda\kappa.\mathtt{let}\ x = [\![e_1]\!](\lambda m.m)\ \mathtt{in}\ [\![e_2]\!]\kappa \\
[\![\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3]\!] &= \lambda\kappa.[\![e_1]\!](\lambda m.\mathtt{if}\ m\ \mathtt{then}\ [\![e_2]\!]\kappa\ \mathtt{else}\ [\![e_3]\!]\kappa)
\end{aligned}
$$

Fig. 5. CPS translation for values and expressions.

The translation of the let expression $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$ needs care to take polymorphism into account. We use a let-expression to express the polymorphism in the source term, and supply the identity continuation $\lambda m.m$ to the CPS transform $[\![e_1]\!]$. This is typable in the target calculus, since a pure expression is translated to an expression of type $\forall t.((\tau \to t) \to t)$.

We can prove that the CPS translation preserves types and equality:

Theorem 5 (Preservation of Types). *If $\Gamma;\ \alpha \vdash e : \tau;\ \beta$ is derivable in $\lambda_{let}^{s/r}$, then $\Gamma^* \vdash [\![e]\!] : (\tau^* \to \alpha^*) \to \beta^*$ is derivable in λ_{let}.*

If $\Gamma \vdash_p e : \tau$ is derivable in $\lambda_{let}^{s/r}$, then, $\Gamma^* \vdash [\![e]\!] : (\tau^* \to \gamma) \to \gamma$ is derivable for an arbitrary type γ in λ_{let}.

Theorem 6 (Preservation of Equality). *If $\Gamma;\ \alpha \vdash e_1 : \tau;\ \beta$ is derivable and $e_1 \rightsquigarrow^* e_2$ in $\lambda_{let}^{s/r}$, then $[\![e_1]\!] = [\![e_2]\!]$ in λ_{let} where $=$ is the least congruence relation which contains $\rightsquigarrow$ in λ_{let}.*[10]

5 Impredicative Polymorphism with Shift and Reset

The second order lambda calculus (Girard's System F) is a solid foundation for advanced concepts in programming languages, since its *impredicative polymorphism* is strictly more

[10] The reduction $\rightsquigarrow$ in λ_{let} is the reduction $\rightsquigarrow$ restricted to the expressions in λ_{let}.

$\alpha, \beta, \gamma, \delta ::= \cdots \mid \forall t.\alpha$	$::= \cdots \mid \forall t.\alpha$	type
$v ::= \cdots \mid \Lambda t.e$	$::= \cdots \mid \Lambda t.v$	value
$e ::= \cdots \mid e\{\alpha\}$	$::= \cdots \mid \Lambda t.e \mid e\{\alpha\}$	expression
$\lambda_2^{s/r,Std}$(standard)	$\lambda_2^{s/r,ML}$(ML-like)	

Fig. 6. Syntax of $\lambda_2^{s/r,Std}$ and $\lambda_2^{s/r,ML}$.

$$\frac{\Gamma \vdash_p e : \tau}{\Gamma \vdash_p \Lambda t.e : \forall t.\tau}\ \text{tabs}, t \notin \mathtt{FTV}(\Gamma) \qquad \frac{\Gamma;\ \alpha \vdash e : \forall t.\tau;\ \beta}{\Gamma;\ \alpha \vdash e\{\sigma\} : \tau[\sigma/t];\ \beta}\ \text{tapp}$$

Fig. 7. Type inference rules for new constructs.

expressive than the predicative one. In this section, we study an extension of (call-by-value version of) System F with shift and reset. It is an *explicitly typed* calculus rather than an implicitly typed calculus like $\lambda_{let}^{s/r}$. Hence, we add two constructs to the expressions: $\Lambda t.e$ for type-abstraction and $e\{\alpha\}$ for type-application. Following Harper and Lillibridge [11], we consider two calculi with impredicative polymorphism that differ in evaluation strategies. The first calculus, $\lambda_2^{s/r,Std}$, adopts the "standard" strategy: $\Lambda t.e$ is treated as a value, and hence we do not evaluate under Λ. The second one, $\lambda_2^{s/r,ML}$, adopts the "ML-like" strategy: $\Lambda t.e$ is a value only when e is a value, and hence we evaluate under Λ.

The syntax of $\lambda_2^{s/r,Std}$ and $\lambda_2^{s/r,ML}$ extends that of $\lambda_{let}^{s/r}$ with the new constructs listed in Figure 6. We annotate bound variables with types, for instance, $\lambda x : \alpha.\ e$. We eliminate let expressions, since they can be macro-defined: for instance, the expression $\mathtt{let}\ f = \lambda x.x\ \mathtt{in}\ (ff)0$ in $\lambda_{let}^{s/r}$ is represented[11] as $(\lambda f : \forall t.(t \to t).\ f\{\mathtt{int} \to \mathtt{int}\}(f\{\mathtt{int}\})0)(\Lambda t.\lambda x : t.\ x)$. Monomorphic and polymorphic types are merged, since the type quantifier $\forall$ may occur at any place in types. The definitions for values and expressions reflect the difference between the two calculi.

The type inference rules for new constructs are common to $\lambda_2^{s/r,Std}$ and $\lambda_2^{s/r,ML}$, and are given in Figure 7. As can be seen by the rule (tabs), the body e in $\Lambda t.e$ is restricted to a pure expression. For $\lambda_2^{s/r,ML}$, this restriction is necessary[12] to ensure the type soundness due to a similar reason as Harper and Lillibridge [11] who proposed to put a kind of value restriction when abstracting types. Unfortunately, their calculus under the value restriction is not very interesting, since the standard and ML-like strategies completely agree on the restricted calculus. We relax the restriction so that e in $\Lambda t.e$ may be an arbitrary pure expression, which makes the two strategies differ on some expressions.

Operational semantics is defined in Figure 8 with a new reduction rule:

$$(\Lambda t.e)\{\alpha\} \rightsquigarrow e[\alpha/t]$$

where $e[\alpha/t]$ denotes the capture-avoiding substitution for types. For $\lambda_2^{s/r,ML}$, the subexpression e in the reduction rule is restricted to a value.

[11] We assume that 0 is a constant of type `int`.

[12] For $\lambda_2^{s/r,Std}$, the restriction is not necessary, and we could have defined a more liberal type system. In the present paper, however, we choose a uniform, simpler syntax.

$E ::= \cdots \mid E\{\alpha\}$	$::= \cdots \mid E\{\alpha\} \mid \Lambda t.E$	e-context
$F ::= \cdots \mid F\{\alpha\}$	$::= \cdots \mid F\{\alpha\}$	pure e-context
$R ::= \cdots \mid (\Lambda t.e)\{\alpha\}$	$::= \cdots \mid (\Lambda t.v)\{\alpha\}$	redex
$\lambda_2^{s/r,Std}$	$\lambda_2^{s/r,ML}$	

Fig. 8. Evaluation Contexts and Redexes.

Polymorphism in $\lambda_2^{s/r,Std}$ is a generalization of Leroy's "polymorphism by name" [15]: consider the expression $\texttt{let}\ f = \langle e \rangle\ \texttt{in}\ (ff)0$ for an expression e of type $t \to t$ and a constant 0 of type $\texttt{int}$. It is represented by $(\lambda f : \forall t.(t \to t).(f\{\texttt{int} \to \texttt{int}\})(f\{\texttt{int}\})0)(\Lambda t.\langle e \rangle)$ in $\lambda_2^{s/r,Std}$, and it is easy to see that the evaluation of e is postponed until a type is applied to $\Lambda t.\langle e \rangle$.

Polymorphism in $\lambda_2^{s/r,ML}$ is a generalization of that for ML. Taking the same example, the outermost β-redex is computed only after $\langle e \rangle$ is computed and returns a value. Then, the variable f is substituted for the value of $\Lambda t.\langle e \rangle$, and the body $(f\{\texttt{int} \to \texttt{int}\})(f\{\texttt{int}\})0$ is computed.

We can show type soundness for $\lambda_2^{s/r,Std}$ and $\lambda_2^{s/r,ML}$.

Theorem 7 (Type Soundness). *Subject reduction property and progress property hold for* $\lambda_2^{s/r,Std}$ *and* $\lambda_2^{s/r,ML}$.

We define a CPS transformation for $\lambda_2^{s/r,Std}$ and $\lambda_2^{s/r,ML}$ in Figures 9 and 10. The target calculus of the translation is System F augmented with basic types, constants, `fix` and conditionals. Equality of the target calculus is the least congruence relation which includes call-by-value $\beta\eta$-equality, β-equality for types $((\Lambda t.e)\{\alpha\} = e[\alpha/t])$, and equality for `fix` and conditionals. Since the target calculus is explicitly typed, the CPS translation for expressions is annotated by types as $[\![e]\!]_{\alpha,\tau,\beta}$, which is well-defined when $\Gamma;\ \alpha \vdash e : \tau;\ \beta$ is derivable for some Γ. It is interesting to see how the difference of evaluation strategies affect the difference of CPS translations in Figures 9 and 10.

Note that the CPS translation for $\lambda_2^{s/r,ML}$ is a natural extension of that for $\lambda_{let}^{s/r}$: for instance, $[\![\texttt{let}\ f = \langle e \rangle\ \texttt{in}\ (ff)0]\!]$ in $\lambda_{let}^{s/r}$ is equal (up to the call-by-value $\beta\eta$-equality) to $[\![(\lambda f : \forall t.(t \to t).(f\{\texttt{int} \to \texttt{int}\})(f\{\texttt{int}\})0)(\Lambda t.\langle e \rangle)]\!]$ in $\lambda_2^{s/r,Std}$.

We can show that the CPS transformations for the two calculi preserve types and equality. Let T be $\lambda_2^{s/r,Std}$ or $\lambda_2^{s/r,ML}$.

$$
\begin{aligned}
(\forall t.\tau)^* &= \forall t.\forall s.((\tau^* \to s) \to s) \text{ for a fresh type variable } s \\
[\![\Lambda t.e]\!]_{\alpha,\forall t.\tau,\alpha} &= \lambda\kappa : ((\forall t.\tau)^* \to \alpha).\kappa(\Lambda t.\Lambda s.[\![e]\!]_{s,\tau,s}) \\
[\![e\{\sigma\}]\!]_{\alpha,\tau[\sigma/t],\beta} &= \lambda\kappa : ((\tau[\sigma/t])^* \to \alpha^*).[\![e]\!]_{\alpha,\forall t.\tau,\beta}(\lambda u : (\forall t.\tau)^*.u\{\sigma^*\}\{\alpha^*\}\kappa)
\end{aligned}
$$

Fig. 9. CPS translation for $\lambda_2^{s/r,Std}$.

$$
\begin{aligned}
(\forall t.\tau)^* &= \forall t.\tau^* \\
[\![\Lambda t.e]\!]_{\alpha,\forall t.\tau,\alpha} &= \lambda\kappa : ((\forall t.\tau^*) \to \alpha).\kappa(\Lambda t.[\![e]\!]_{\tau,\tau,\tau}(\lambda m : \tau^*.m)) \\
[\![e\{\sigma\}]\!]_{\alpha,\tau[\sigma/t],\beta} &= \lambda\kappa : ((\tau[\sigma/t])^* \to \alpha^*).\ [\![e]\!]_{\alpha,\forall t.\tau,\beta}(\lambda u : \forall t.\tau^*.\kappa(u\{\sigma^*\}))
\end{aligned}
$$

Fig. 10. CPS translation for $\lambda_2^{s/r,ML}$.

Theorem 8 (Preservation of Types and Equality).

1. *If $\Gamma;\ \alpha \vdash e : \tau;\ \beta$ is derivable in T, then $\Gamma^* \vdash [\![e]\!]_{\alpha,\tau,\beta} : (\tau^* \to \alpha^*) \to \beta^*$ is derivable in the target calculus.*
2. *If $\Gamma \vdash_p e : \tau$ is derivable in T, then $\Gamma^* \vdash [\![e]\!]_{s,\tau,s} : (\tau^* \to s) \to s$ is derivable for any type variable s in the target calculus.*
3. *If $\Gamma;\ \alpha \vdash e : \tau;\ \beta$ is derivable in T, and $e \rightsquigarrow^* e'$, then $[\![e]\!]_{\alpha,\tau,\beta} = [\![e']\!]_{\alpha,\tau,\beta}$ under the equality of the target calculus.*

6 Conclusion

We have introduced predicative and impredicative polymorphic typed calculi for shift and reset, and investigated their properties such as type soundness and relationship to CPS translations. We have extended Danvy and Filinski's monomorphic type system for shift and reset to polymorphic one, and have shown that a number of pleasant properties hold for the polymorphic calculi. We have shown that our calculi have a natural representation for the "purity" of expressions, and that the purity restriction suffices for the type systems to be sound, thus generalizing value restriction used in Standard ML and OCaml.

In the literature, a number of authors have tackled the unsoundness problem of polymorphism and effects [19, 15, 20, 11]. We have proposed a simple solution based on the notion of "purity", which is, in the presence of the reset operator, less restrictive than the notion of "syntactic values" in ML. We have also investigated two evaluation strategies for impredicative calculi, each of which generalizes ML's and Leroy's solutions for the unsoundness problem.

Several authors have studied polymorphic calculi with control operators for delimited continuations. Introducing polymorphism into a calculus with shift and reset has been implicit by Danvy who gave many programming examples (see, for instance, [3]). In fact, his interesting examples encouraged us to formulate the calculi in the present paper. Filinski [7] implemented shift and reset in SML/NJ, thus enabling one to write polymorphic functions with shift and reset. However, the expressivity of his system is limited since the answer type is fixed once and for all. The same goes for the calculus with `cupto` by Gunter et al. [9].

Kiselyov et al. [14] have implemented shift and reset in OCaml, and their examples made use of let-polymorphism. However, their paper did not give formal accounts for polymorphism. As far as we know, the present paper is the first to provide a systematic study on the interaction of polymorphism and control operators for delimited continuations.

Although we believe that our calculus serves as a good foundation for studying polymorphic delimited continuations calculi, this is only the first step; we need deeper understanding and better theories. The first author of the present paper has studied logical relations based on Danvy and Filinski's monomorphic type system [1], but it is not apparent if his result extends to the polymorphic case. Hasegawa [12] studied parametricity principle for the second order, call-by-name $\lambda\mu$-calculus (similar to System F + `call/cc`), and obtained the notion of "focal parametricity". Although he works in call-by-name, we hope to find some connection between our work and his results in the future. A recent work by Mogelberg and Simpson [16] treats a similar notion in call-by-value.

Acknowledgments. We would like to thank Olivier Danvy and Masahito Hasegawa for helpful comments and suggestions. This work was partly supported by JSPS Grant-in-Aid for Scientific Research (C) 18500005 and 16500004.

References

1. K. Asai. Logical Relations for Call-by-value Delimited Continuations. In *Trends in Functional Programming*, volume 6, pages 63–78, 2007.
2. K. Asai. On Typing Delimited Continuations: Three New Solutions to the Printf Problem. 2007. Submitted. See `http://pllab.is.ocha.ac.jp/~asai/papers/`.
3. O. Danvy. *An Analytical Approach to Program as Data Objects*. DSc thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark, 2006.
4. O. Danvy and A. Filinski. A Functional Abstraction of Typed Contexts. Technical Report 89/12, DIKU, University of Copenhagen, July 1989.
5. O. Danvy and A. Filinski. Abstracting Control. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, 1990.
6. O. Danvy and A. Filinski. Representing Control: a Study of the CPS Transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.
7. A. Filinski. Representing Monads. In *POPL*, pages 446–457, 1994.
8. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
9. C. A. Gunter, D. Remy, and J. G. Riecke. A Generalization of Exceptions and Control in ML-Like Languages. In *FPCA*, pages 12–23, 1995.
10. R. Harper, B. F. Duba, and D. MacQueen. Typing First-Class Continuations in ML. *J. Funct. Program.*, 3(4):465–484, 1993.
11. R. Harper and M. Lillibridge. Explicit polymorphism and CPS conversion. In *POPL*, pages 206–219, 1993.
12. M. Hasegawa. Relational parametricity and control. *Logical Methods in Computer Science*, 2(3), 2006.
13. Y. Kameyama and M. Hasegawa. A sound and complete axiomatization for delimited continuations. In *ICFP*, pages 177–188, 2003.
14. O. Kiselyov, C. c. Shan, and A. Sabry. Delimited dynamic binding. In *ICFP*, pages 26–37, 2006.
15. X. Leroy. Polymorphism by name for references and continuations. In *POPL*, pages 220–231, 1993.
16. R.E. Mogelberg and A. Simpson. Relational parametricity for computational effects. In *LICS*, 2007.

17. C. Strachey. Fundamental concepts in programming languages. International Summer School in Computer Programming, Copenhagen, Denmark, August 1967.
18. H. Thielecke. From Control Effects to Typed Continuation Passing. In *POPL*, pages 139–149, New York, January 2003. ACM Press.
19. M. Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1):1–34, 1990.
20. A. K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.
21. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

A Proofs of Theorems and Lemmas

In this appendix, we give detailed proofs of theorems and lemmas for interested readers.

Proof (Lemma 1). This lemma can be proved by induction on the derivation of $\Gamma_1; \alpha \vdash e : \sigma; \beta$. The only tricky case is the rule (let) in which the type variables in $\mathtt{FTV}(\Gamma_2) - \mathtt{FTV}(\Gamma_1)$ may clash with the type variables being generalized by this rule, i.e., those type variables in $\mathtt{FTV}(\sigma) - \mathtt{FTV}(\Gamma)$ in this rule. However, since these type variables are so called eigen variables, and can be systematically renamed to fresh type variables, we can manage this case.

The other cases can be proved easily.

Lemmas 2 and 3 are proved by straightforward induction on the derivation, and the proof is omitted.

Proof (Theorem 1). It suffices to prove that, for each reduction rule $R \rightsquigarrow e$, if $\Gamma;\ \alpha \vdash R : \tau;\ \beta$ is derivable, then $\Gamma;\ \alpha \vdash e : \tau;\ \beta$ can be derived.

The cases when R is $(\lambda x.e_1)v$ or $\mathtt{let}\ x = v\ \mathtt{in}\ e_1$ can be proved using Lemmas 2 and 3. The cases when R is $\langle v \rangle$ or conditionals are easy to prove.

The remaining case is when $R \equiv \langle F[\mathcal{S}k.e_1] \rangle$ and $e \equiv \langle \mathtt{let}\ k = \lambda x.\langle F[x] \rangle\ \mathtt{in}\ e_1 \rangle$. We consider the following set of reductions:

$$\begin{aligned}
(\mathcal{S}k.e_1)e_2 &\to \mathcal{S}k'.\mathtt{let}\ k = \lambda u.\langle k'(ue_2) \rangle\ \mathtt{in}\ e_1 \\
v(\mathcal{S}k.e_1) &\to \mathcal{S}k'.\mathtt{let}\ k = \lambda u.\langle k'(vu) \rangle\ \mathtt{in}\ e_1 \\
\langle \mathcal{S}k.e \rangle &\to \langle \mathtt{let}\ k = \lambda u.u\ \mathtt{in}\ e \rangle \\
\langle (\lambda u.\langle F[u] \rangle)e \rangle &\to \langle F[e] \rangle \\
(\lambda x.x)e &\to e
\end{aligned}$$

It is not difficult to see that the reduction $R \rightsquigarrow e$ can be "decomposed" into these reductions. In other words, we have:

$$\langle F[\mathcal{S}k.e_1] \rangle \to^* \langle \mathtt{let}\ k = \lambda x.\langle F[x] \rangle\ \mathtt{in}\ e_1 \rangle$$

which can be proved by induction on the pure evaluation context F. Then, it remains to show the subject reduction property for these reductions, but all of them can be carried out easily.

Proof (Theorem 2). We can prove that, if $;\ \alpha \vdash e_1 : \tau;\ \beta$ is derived (i.e., e_1 has no free variables), e_1 is in one of the following forms: (1) a value v, (2) $E[R]$ for some evaluation

context E and redex R, or (3) $F[\mathcal{S}k.e_2]$ for a pure evaluation context F, a variable k and an expression e_2.

Suppose $\vdash_p \langle e \rangle : \tau$ is derived. Then we have $;\alpha \vdash \langle e \rangle : \tau;\ \alpha$, and hence $\langle e \rangle$ must be one of the three forms above. Since $\langle e \rangle$ has an outermost reset, the case (3) cannot happen, and we obtain the conclusion of the theorem.

The type inference algorithm stated in Theorem 3 is given in Section B of the appendix.

Proof (Theorem 4). Confluence can be proved by Takahashi's parallel reduction which is omitted. (Note we do not have overlapping redexes.)

To prove strong normalization for $\lambda_{let}^{s/r}$ without `fix`, we define a refined CPS translation, which produces fewer administrative redexes (administrative redexes are those redexes generated at the translation time) than the CPS translation we gave in the main text of the paper. There are several such CPS translations since Plotkin's colon translation, and we use a slightly modified version of Danvy and Filinski's two-level lambda calculus [6].

In the two-level lambda calculus, λ and application (explicitly written as @ here) are classified[13] into "static" and "dynamic", depending on whether they constitute administrative redexes (i.e., they are generated at the translation time) or source redexes (i.e., they exist in the source expression). The static lambda's and @'s are annotated by overlines as $\overline{\lambda}$ and $\overline{@}$, while the dynamic ones as $\underline{\lambda}$ and $\underline{@}$.

The target of the modified CPS translation consists of dynamic constructs and those constructs which are not lambda's and application symbols. (We will later define the precise grammar of the target language after defining the refined CPS translation.)

The modified CPS translation $[\![e, K]\!]$ takes an expression e and an extra argument K (representing the continuation), and returns an expression, which is defined by:

$$
\begin{aligned}
[\![v, K]\!] &= K\overline{@}v^* \\
[\![e_1@e_2, K]\!] &= [\![e_1, \overline{\lambda}m_1.[\![e_2, \overline{\lambda}m_2.(m_1\underline{@}m_2)\underline{@}(\underline{\lambda}n.\ K\overline{@}n)]\!]]\!] \\
[\![\langle e \rangle, K]\!] &= K\overline{@}[\![e, \overline{\lambda}m.m]\!] \\
[\![\mathcal{S}k.e, K]\!] &= \texttt{let}\ m_1 = \texttt{true}\ \texttt{in} \\
&\qquad \texttt{let}\ k = \underline{\lambda}n.\underline{\lambda}\kappa'.\kappa'\underline{@}(K\overline{@}n)\ \texttt{in}\ [\![e, \overline{\lambda}m.m]\!] \\
[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2, K]\!] &= \texttt{let}\ x = [\![e_1, \overline{\lambda}m.m]\!]\ \texttt{in}\ [\![e_2, K]\!] \\
[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, K]\!] &= [\![e_1, \overline{\lambda}m.\texttt{if}\ m\ \texttt{then}\ [\![e_2, K]\!]\ \texttt{else}\ [\![e_3, K]\!]]\!] \\
x^* &= x \\
c^* &= c \\
(\lambda x.e)^* &= \underline{\lambda}x.\underline{\lambda}\kappa.[\![e, \overline{\lambda}m.\ \kappa\underline{@}m]\!]
\end{aligned}
$$

where m,m_1,m_2,n,κ, and κ' are fresh variables. In the definition for shift, we have added a redundant redex $\texttt{let}\ m_1 = \texttt{true}\ \texttt{in} \cdots$ for the purpose of SN proof.

The complete CPS transform of an expression e is defined by $\mathcal{C}[\![e]\!] \equiv \underline{\lambda}\kappa.[\![e, \overline{\lambda}x.\kappa\underline{@}x]\!]$.

[13] In fact, Danvy and Filinski classified every construct into two classes, but here we need to classify only lambda's and applications, and other constructs are implicitly classified as "dynamic".

The grammar of the target language of the refined CPS translation is defined as follows:

$$
\begin{aligned}
d ::= {} & c \mid x \mid n \mid n \underline{@} d \mid (d \underline{@} d) \underline{@} (\underline{\lambda} n.d) \mid \underline{\lambda} x.\underline{\lambda}\kappa.d \\
& \mid \texttt{let}\ x = d\ \texttt{in}\ d \mid \texttt{let}\ n = d\ \texttt{in}\ d \mid \texttt{if}\ d\ \texttt{then}\ d\ \texttt{else}\ d \\
& \mid s \\
s ::= {} & K \overline{@} d \\
K ::= {} & \overline{\lambda} n.d \quad \text{where } n \in \texttt{FV}(d)
\end{aligned}
$$

In the definition, x is a source variable (which appears in the source expression of the CPS translation), n is an auxiliary variable (which is introduced by the CPS translation), d is a dynamic expression, s is a static expression, and K is a (static) continuation.

It is important to note the side condition for K, $n \in \texttt{FV}(d)$, which means that the continuation K does not discard its argument n.

Let λ^{+}_{let} be the typed calculus λ_{let} with the function space being replaced by two function spaces $\alpha \overline{\rightarrow} \beta$ (for static functions) and $\alpha \underline{\rightarrow} \beta$ (for dynamic ones), and, accordingly, lambda abstractions and applications being replaced by two versions.

Then, we can easily prove the following for any expression e typable in $\lambda^{s/r}_{let}$:

- $\mathcal{C}[\![e]\!]$ follows the grammar above (including the side condition of the arguments of continuations).
- $\mathcal{C}[\![e]\!]$ typechecks in λ^{+}_{let}.
- Subject reduction w. r. t. static β-reduction holds. Namely, if we reduce a static β-redex $(\overline{\lambda} x.e_1) \overline{@} e_2$ to $e_1[e_2/x]$, then an expression typable in λ^{+}_{let} reduces to an expression still typable in λ^{+}_{let}. Also if an expression before a static β-reduction satisfies the side condition of the arguments of continuations, so does the expression after the reduction.

The reduction of static β-redex $(\overline{\lambda} x.e_1) \overline{@} e_2 \rightarrow e_1[e_2/x]$ is Church-Rosser and strongly normalizing in the target language, and we can take the unique normal form of each expression in the target language, which does not have any static constructs $\overline{\lambda}$ and $\overline{@}$. Let us write the unique normal form (w.r.t. static β reduction) of a term e by $\texttt{NF}(e)$.

We are going to prove the key property that, for any reduction $e_1 \rightsquigarrow e_2$ in $\lambda^{s/r}_{let}$ other than the reset-value reduction ($\langle v \rangle \rightsquigarrow v$), and any continuation K generated by the grammar above, we have $\texttt{NF}([\![e_1, K]\!]) \rightsquigarrow^{+} \texttt{NF}([\![e_2, K]\!])$ in the target calculus λ_{let}[14].

- If the reduction is the call-by-value β reduction (the first reduction in Figure 2), or reductions for let (the fifth reduction) or conditional (the sixth and seventh reductions), then the key property can be proved easily.
- For the fourth reduction $\langle F[\mathcal{S}k.e] \rangle \rightsquigarrow \langle \texttt{let}\ k = \lambda x.\langle F[x] \rangle\ \texttt{in}\ e \rangle$, we first prove that $\texttt{NF}([\![F[e], K]\!]) \equiv \texttt{NF}([\![e, \overline{\lambda} m.[\![F[m], K]\!]]\!])$ for any pure e-context F, any expression e, and any continuation K, which can be proved by induction on F.

[14] Since we consider only normal forms wrt static β-redexes, all reductions in the target calculus are dynamic β-reduction, let-reductions, or reductions for conditionals.

Then we can prove:

$$
\begin{aligned}
&\mathtt{NF}([\![\langle F[\mathcal{S}k.e]\rangle, K]\!]) \\
&\equiv \mathtt{NF}(K\overline{@}[\![F[\mathcal{S}k.e], \overline{\lambda}m.m]\!]) \\
&\equiv \mathtt{NF}(K\overline{@}[\![\mathcal{S}k.e, \overline{\lambda}m'.[\![F[m'], \overline{\lambda}m.m]\!]]\!]) \\
&\equiv \mathtt{NF}(K\overline{@}(\mathtt{let}\ m_1 = \mathtt{true}\ \mathtt{in}\ \mathtt{let}\ k = \underline{\lambda}n.\underline{\lambda}\kappa'.\kappa'\underline{@}((\overline{\lambda}m'.[\![F[m'], \overline{\lambda}m.m]\!])\overline{@}n)\ \mathtt{in}\ [\![e, \overline{\lambda}m.m]\!])) \\
&\equiv \mathtt{NF}(K\overline{@}(\mathtt{let}\ m_1 = \mathtt{true}\ \mathtt{in}\ \mathtt{let}\ k = \underline{\lambda}n.\underline{\lambda}\kappa'.\kappa'\underline{@}[\![F[n], \overline{\lambda}m.m]\!]\ \mathtt{in}\ [\![e, \overline{\lambda}m.m]\!])) \\
&\rightsquigarrow^{+} \mathtt{NF}(K\overline{@}\mathtt{let}\ k = \underline{\lambda}n.\underline{\lambda}\kappa'.\kappa'\underline{@}[\![F[n], \overline{\lambda}m.m]\!]\ \mathtt{in}\ [\![e, \overline{\lambda}m.m]\!])
\end{aligned}
$$

Note that, since K does not discard its argument, the reduction $\mathtt{let}\ m_1 = \mathtt{true}\ \mathtt{in}\ e_1 \rightsquigarrow e_1$ is preserved, and at least one step reduction occurs during this sequence. We also have:

$$
\begin{aligned}
&\mathtt{NF}([\![\langle \mathtt{let}\ k = \lambda x.\langle F[x]\rangle\ \mathtt{in}\ e\rangle, K]\!]) \\
&\equiv \mathtt{NF}(K\overline{@}\mathtt{let}\ k = \underline{\lambda}x.\underline{\lambda}\kappa.\kappa\underline{@}[\![F[x], \overline{\lambda}m.m]\!]\ \mathtt{in}\ [\![e, \overline{\lambda}m.m]\!])
\end{aligned}
$$

and therefore the resulting expressions are the same up to α-equivalence, hence we have:

$$
\mathtt{NF}([\![\langle F[\mathcal{S}k.e]\rangle, K]\!]) \rightsquigarrow^{+} \mathtt{NF}([\![\langle \mathtt{let}\ k = \lambda x.\langle F[x]\rangle\ \mathtt{in}\ e\rangle, K]\!])
$$

We also note that the reset-value reduction $\langle v\rangle \rightsquigarrow v$ will be CPS translated to identity, that is, $\mathtt{NF}([\![\langle v\rangle, K]\!]) \equiv \mathtt{NF}([\![v, K]\!])$ for a value v and a continuation K.

In summary we have:

- If $e_1 \rightsquigarrow e_2$ by the reduction other than reset-value, then $\mathtt{NF}(\mathcal{C}[\![e_1]\!]) \rightsquigarrow^{+} \mathtt{NF}(\mathcal{C}[\![e_2]\!])$.
- If $e_1 \rightsquigarrow e_2$ by the reset-value reduction, $\mathtt{NF}(\mathcal{C}[\![e_1]\!]) \equiv \mathtt{NF}(\mathcal{C}[\![e_2]\!])$.

We now prove the strong normalization for $\lambda_{let}^{s/r}$. Suppose there is an infinite reduction sequence $e_1 \rightsquigarrow e_2 \rightsquigarrow \cdots$ in $\lambda_{let}^{s/r}$. Since the reset-value reduction ($\langle v\rangle \rightsquigarrow v$) cannot be applied to an expression infinite many times, the reduction sequence must contain infinitely many reductions which are not reset-value. Then by the property above, we have an infinite sequence $\mathtt{NF}(\mathcal{C}[\![e_1]\!]) \rightsquigarrow^{+} \mathtt{NF}(\mathcal{C}[\![e_2]\!]) \rightsquigarrow^{+} \cdots$. But the target calculus λ_{let} is a strongly normalizing calculus, we get contradiction. Hence, $\lambda_{let}^{s/r}$ does not have an infinite reduction sequence.

Proof (Theorem 5). The two statements in the theorem are proved by simultaneous induction on the derivation of $\Gamma;\ \alpha \vdash e : \tau;\ \beta$ and that of $\Gamma \vdash_p e : \tau$.

We only prove the case when the last rule of the derivation is the rule (let). Then $e \equiv (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2)$, and we have derivations for $\Gamma \vdash_p e_1 : \sigma$ and $\Gamma, x : \mathtt{Gen}(\sigma, \Gamma);\ \alpha \vdash e_2 : \tau;\ \beta$. By induction hypothesis, we can derive $\Gamma^* \vdash [\![e_1]\!] : (\sigma^* \to \sigma^*) \to \sigma^*$, and then $\Gamma^* \vdash [\![e_1]\!](\lambda m.m) : \sigma^*$. Again by induction hypothesis, we can derive $\Gamma^*, x : (\mathtt{Gen}(\sigma, \Gamma))^* \vdash [\![e_2]\!] : (\tau^* \to \alpha^*) \to \beta^*$. Since $(\mathtt{Gen}(\sigma, \Gamma))^* \equiv \mathtt{Gen}(\sigma^*, \Gamma^*)$, we can apply the rule (let) to obtain $\Gamma^* \vdash [\![\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2]\!] : (\tau^* \to \alpha^*) \to \beta^*$.

Proof (Theorem 6). This theorem can be proved in the same manner as in Kameyama and Hasegawa [13], where the soundness and completeness of the CPS translation with respect to the equality were proved.

Proof (Theorem 7). To prove the subject reduction property, we need to consider two new cases.

(Case-1: $(\Lambda t.e)\{\alpha\} \rightsquigarrow e[\alpha/t]$)

We first note that $\lambda_2^{s/r,Std}$ and $\lambda_2^{s/r,ML}$ both enjoy the type substitution property which means, if $\Gamma;\ \alpha \vdash e : \tau;\ \beta$ is derivable then, for any substitution θ for type variables, we can derive $\Gamma\theta;\ \alpha\theta \vdash e\theta : \tau\theta;\ \beta\theta$. Then, the subject reduction property for Case-1 is easily proved.

(Case-2: $\langle F[\mathcal{S}k.e_1]\rangle \rightsquigarrow \langle \texttt{let}\ k = \lambda x.\langle F[x]\rangle\ \texttt{in}\ e_2\rangle$)

In $\lambda_{let}^{s/r}$, this case has been proved by "decomposing" this big reduction into several smaller reductions. In $\lambda_2^{s/r,Std}$ and $\lambda_2^{s/r,ML}$, we have a new small reduction after this "decomposition" as:

$$(\mathcal{S}k.e_1)\{\alpha\} \rightarrow \mathcal{S}k'.\texttt{let}\ k = \lambda u.\langle k'(u\{\alpha\})\rangle\ \texttt{in}\ e_1$$

To prove the subject reduction property for this case is again straightforward.

Progress property can be proved in the same way as that for $\lambda_{let}^{s/r}$.

Proof (Theorem 8). Due to the modularity of the CPS translations, we only have to prove the cases for new constructs.

The proof of the type-preservation part of the theorem is quite standard for both calculi. The only problem was how to *define* them correctly.

The proof of the equation-preservation part of the theorem needs to treat two new reductions that appeared in the proof of Theorem 7 above. In fact, we need to prove four new cases (two new reductions for two calculi).

We frequently omit the type annotations in the following.

(Case-1 for $\lambda_2^{s/r,Std}$: $(\Lambda t.e)\{\alpha\} \rightsquigarrow e[\alpha/t]$)

$$\begin{aligned}
[\![(\Lambda t.e)\{\alpha\}]\!]_{\beta,\tau[\alpha/t],\beta} &\equiv \lambda\kappa.(\lambda\kappa'.\kappa'(\Lambda t.\Lambda s.[\![e]\!]))(\lambda u.u\{\alpha^*\}\{\beta^*\}\kappa) \\
&\rightsquigarrow \lambda\kappa.(\lambda u.u\{\alpha^*\}\{\beta^*\}\kappa)(\Lambda t.\Lambda s.[\![e]\!]) \\
&\rightsquigarrow^+ \lambda\kappa.[\![e]\!][\alpha^*/t][\beta^*/s]\kappa \\
&\equiv \lambda\kappa.[\![e]\!][\alpha^*/t]\kappa \\
&= [\![e[\alpha/t]]\!]_{\beta,\tau[\alpha/t],\beta}
\end{aligned}$$

From the third last to the second last line, we used the fact that e does not contain the type variable s freely. From the second last to the last line, we used η-equality.

(Case-2 for $\lambda_2^{s/r,Std}$: $\langle F[\mathcal{S}k.e_1]\rangle \rightsquigarrow \langle \texttt{let}\ k = \lambda x.\langle F[x]\rangle\ \texttt{in}\ e_2\rangle$)

Again we pick up the following smaller reduction by decomposing the big reduction.

$$(\mathcal{S}k.e_1)\{\alpha\} \rightarrow \mathcal{S}k'.\texttt{let}\ k = \lambda u.\langle k'(u\{\alpha\})\rangle\ \texttt{in}\ e_1$$

$$
\begin{aligned}
&[\![(\mathcal{S}k.e_1)\{\alpha\}]\!] \\
&\equiv \lambda\kappa.(\lambda\kappa_1.\texttt{let } k = \lambda n\kappa_2.\kappa_2(\kappa_1 n) \texttt{ in } [\![e_1]\!](\lambda m.m))(\lambda u.u\{\sigma^*\}\{\alpha^*\}\kappa) \\
&\rightsquigarrow^+ \lambda\kappa.\texttt{let } k = \lambda n\kappa_2.\kappa_2(n\{\sigma^*\}\{\alpha^*\}\kappa) \texttt{ in } [\![e_1]\!](\lambda m.m) \\
&[\![\mathcal{S}k'.\texttt{let } k = \lambda u.\langle k'(u\{\alpha\})\rangle \texttt{ in } e_1]\!] \\
&\rightsquigarrow^+ \lambda\kappa.\texttt{let } k' = \lambda n\kappa_1.\kappa_1(\kappa n) \texttt{ in} \\
&\qquad \texttt{let } k = \lambda u.\lambda\kappa_3.\kappa_3(u\{\sigma^*\}\{\alpha^*\}(\lambda w.k'w(\lambda m.m))) \texttt{ in } [\![e_1]\!](\lambda m.m) \\
&\rightsquigarrow^+ \lambda\kappa.\texttt{let } k = \lambda u.\lambda\kappa_3.\kappa_3(u\{\sigma^*\}\{\alpha^*\}(\lambda w.\kappa w)) \texttt{ in } [\![e_1]\!](\lambda m.m)
\end{aligned}
$$

The CPS translations of the two terms are equal up to call-by-value η-equality: $\lambda w.\kappa w = \kappa$.

(Case-1 for $\lambda_2^{s/r,ML}$: $(\Lambda t.v)\{\alpha\} \rightsquigarrow v[\alpha/t]$)

$$
\begin{aligned}
[\![(\Lambda t.v)\{\sigma\}]\!]_{\beta,\tau[\sigma/t],\beta} &\equiv \lambda\kappa.(\lambda\kappa_1.\kappa_1(\Lambda t.(\lambda\kappa_2.\kappa_2(v)^*)(\lambda m.m)))(\lambda u.\kappa(u\{\sigma^*\})) \\
&\rightsquigarrow^+ \lambda\kappa.\kappa((\Lambda t.v^*)\{\sigma^*\}) \\
&\rightsquigarrow \lambda\kappa.\kappa(v^*[\sigma^*/t]) \\
&\equiv [\![v[\sigma/t]]\!]
\end{aligned}
$$

(Case-2 for $\lambda_2^{s/r,ML}$: $\langle F[\mathcal{S}k.e_1]\rangle \rightsquigarrow \langle \texttt{let } k = \lambda x.\langle F[x]\rangle \texttt{ in } e_2\rangle$)

Again we shall prove the following case.

$$(\mathcal{S}k.e_1)\{\alpha\} \rightarrow \mathcal{S}k'.\texttt{let } k = \lambda u.\langle k'(u\{\alpha\})\rangle \texttt{ in } e_1$$

$$
\begin{aligned}
&[\![(\mathcal{S}k.e_1)\{\alpha\}]\!] \\
&\equiv \lambda\kappa.(\lambda\kappa_1.\texttt{let } k = \lambda n\kappa_2.\kappa_2(\kappa_1 n) \texttt{ in } [\![e_1]\!](\lambda m.m))(\lambda u.\kappa(u\{\alpha^*\})) \\
&\rightsquigarrow^+ \lambda\kappa.\texttt{let } k = \lambda n\kappa_2.\kappa_2(\kappa(n\{\alpha^*\})) \texttt{ in } [\![e_1]\!](\lambda m.m) \\
&[\![\mathcal{S}k'.\texttt{let } k = \lambda u.\langle k'(u\{\alpha\})\rangle \texttt{ in } e_1]\!] \\
&\rightsquigarrow^+ \lambda\kappa.\texttt{let } k' = \lambda n\kappa_1.\kappa_1(\kappa n) \texttt{ in} \\
&\qquad \texttt{let } k = \lambda u.\lambda\kappa_3.\kappa_3((\lambda w.k'w(\lambda m.m))(u\{\alpha^*\})) \texttt{ in } [\![e_1]\!](\lambda m.m) \\
&\rightsquigarrow^+ \lambda\kappa.\texttt{let } k = \lambda u\kappa_3.\kappa_3(\lambda w.\kappa w)(u\{\alpha^*\}) \texttt{ in } [\![e_1]\!](\lambda m.m)
\end{aligned}
$$

Again using call-by-value η-equality, the CPS translations of the two terms are equal.

B Type Inference Algorithm for $\boldsymbol{\lambda_{let}^{s/r}}$

We can extend Hindley-Milner's type inference algorithm W to $\lambda_{let}^{s/r}$. The extended algorithm is called W' here.

W' receives two parameters as inputs: a valid context Γ and a raw expression e such that all free variables in e are contained in Γ.

$W'(\Gamma, e)$ either succeeds or fails. If it succeeds, it returns a quadruple $(\theta;\ \alpha, \tau, \beta)$ where θ is a substitution for type variables, α, τ and β are types such that $\Gamma\theta;\ \alpha \vdash e : \tau;\ \beta$ is derivable.

We define a few important cases for W'. In the following, it should be silently understood that, if any subcomputation (a recursive call to W' or unification) fails, then the whole computation fails.

(Case 1: e is a variable x)

Since x appears in Γ, we can assume $(x : \forall t_1.\cdots\forall t_n.\sigma) \in \Gamma$ with monomorphic σ. Choose fresh type variables $s_0, s_1, \cdots, s_n$. Then the algorithm returns $([\,];\ s_0, \sigma[s_1, \cdots, s_n/t_1, \cdots, t_n], s_0)$ where $[\,]$ denotes the empty substitution.

(Case 2: e is $e_1 e_2$)

Compute $W'(\Gamma, e_1)$ to obtain $(\theta_1;\ \alpha_1, \tau_1, \beta_1)$, and compute $W'(\Gamma\theta_1, e_2)$ to obtain $(\theta_2;\ \alpha_2, \tau_2, \beta_2)$.

Choose fresh type variables t_1 and t_2, and unify: $\tau_1\theta_2 = (\tau_2/t_1 \to t_2/\alpha_2)\theta_2$ and $\alpha_1\theta_2 = \beta_2$. Let θ_3 be the most general unifier of these equations, and return $(\theta_3\circ\theta_2\circ\theta_1;\ t_1\theta_3, t_2\theta_3, \beta_1\theta_2\theta_3)$.

(Case 3: e is $\mathcal{S}k.e_1$)

Choose fresh type variables t_1, t_2 and t_3, and let Γ' be $\Gamma, k : \forall t_1.(t_2/t_1 \to t_3/t_1)$. Compute $W'(\Gamma', e)$ to obtain $(\theta_1;\ \alpha, \tau, \beta)$. Unify $\alpha = \tau$, and let θ_2 be its most general unifier. Then return $(\theta_2 \circ \theta_1;\ t_3\theta, t_2\theta, \beta\theta)$.

(Case 4: e is $\langle e_1 \rangle$)

Compute $W'(\Gamma, e_1)$ to obtain $(\theta_1;\ \alpha, \tau, \beta)$. Unify $\alpha = \tau$ and let θ_2 be its most general unifier. Choose a fresh type variable t. Returns $(\theta_2 \circ \theta_1;\ t, \beta\theta_2, t)$.

(Case 5: e is $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$)

If e_1 is not pure, i.e., not a syntactic value or a reset expression, then it should be understood as a monomorphic let, or an abbreviation of $(\lambda x.e_2)e_1$, then compute $W'(\Gamma, (\lambda x.e_2)e_1)$.

If e_1 is pure, it is a polymorphic let expression. Compute $W'(\Gamma, e_1)$ to obtain $(\theta_1;\ \alpha_1, \tau_1, \beta_1)$. (It is automatically guaranteed that, $\alpha_1 \equiv \beta_1$ is a type variable which does not appear in any other types.) Let $t_1, \cdots, t_n$ be the type variables in $\texttt{FTV}(\tau) - \texttt{FTV}(\Gamma\theta_1)$. Let Γ' be $\Gamma\theta_1, x : \forall t_1.\cdots\forall t_n.\tau$. Compute $W'(\Gamma', e_2)$ to obtain $(\theta_3;\ \alpha_3, \tau_3, \beta_3)$. Then return $(\theta_3 \circ \theta_2 \circ \theta_1;\ \alpha_3, \tau_3, \beta_3)$.

Other cases are straightforward.

We can prove Theorem 3 in the same manner as that for W.