Typed Dynamic Control Operators for Delimited Continuations

Yukiyoshi Kameyama and Takuo Yonezawa

Department of Computer Science, University of Tsukuba
kameyama@acm.org, yone@logic.cs.tsukuba.ac.jp

Abstract. We study the *dynamic* control operators for delimited continuations, `control` and `prompt`. Based on recent developments on purely functional CPS translations for them, we introduce a polymorphically typed calculus for these control operators which allows answer-type modification. We show that our calculus enjoys type soundness and is compatible with the CPS translation. We also show that the typed dynamic control operators can macro-express the typed *static* ones (`shift` and `reset`), while the converse direction is not possible, which exhibits a sharp contrast with the type-free case.
Keywords: Type System, Delimited Continuation, Dynamic Control Operator, CPS Translation, Polymorphism, Expressivity.

1 Introduction

Delimited continuations represent not the rest of the computation as with traditional continuations [18], but only part of the rest of the computation. As such, delimited continuations have been used to model backtracking in contrast to traditional continuations that are used to model jumps.

In direct style, traditional continuations are accessed with control operators such as call/cc. There is, however, more variety for delimited continuations:

- Felleisen [12] proposed a control delimiter to signify *part* of an evaluation context, or a *delimited* continuation. This lead to the new control operators `control` and `prompt`, which are called *dynamic* control operators.
- Danvy and Filinski discovered that delimited continuations are supported by an already existing formalism of 2CPS, the image of iterated CPS translations. They proposed new control operators `shift` and `reset` in the direct-style counterpart [9]. They are called *static* control operators.

Since these proposals, the static control operators have been intensively studied while the dynamic ones are relatively less studied. For `shift`/`reset`, there are a number of theoretical results [13, 14] as well as useful examples in partial evaluation, one-pass CPS translation, and mobile codes, while we do not find many corresponding works for `control`/`prompt` in the literature, partly due to the difficulty in reasoning about the dynamic features of `control`/`prompt`.

Recently, several authors have started to obtain better understanding for dynamic ones, and to connect dynamic and static ones. Shan [7] macro-expressed

`control` and `prompt` in terms of `shift` and `reset` using recursive types. (See also Kiselyov's work [15].) Biernacki, Danvy, and Millikin [6] derived a CPS translation for `control` and `prompt` from a definitional abstract machine, and gave another encoding of `control`/`prompt` by `shift`/`reset`. Dybvig, Peyton Jones, and Sabry [11] gave a uniform monadic framework for delimited continuations including `control`/`prompt`. However, no work has studied the direct-style type system for `control`/`prompt` in a way comparative to the type system for `shift`/`reset`. The proposed encodings were done in either type-free or recursively typed settings,[1] and the proposed CPS translations assumed a restricted direct-style type system in that the control effect of "answer-type modification" was not allowed. We think this effect is indispensable for delimited continuations, for instance, it is needed to type the `printf` function in direct style [8, 1].

In this paper, we propose a direct-style type system for `control` and `prompt`, which allows answer-type modification, does not need recursive types, and has ML-like let-polymorphism. We derive the type inference rules from the type structure for the CPS translations for `control` and `prompt` recently developed by the above mentioned works. The type system in this paper is a proper extension of that in our previous work [20], which does not allow answer-type modification. We show that our type system enjoys Subject Reduction and Progress properties, and that types are preserved by the CPS translation. The first two properties constitute type soundness, and the third property is necessary for a semantical study.

As an application of our type system, we compare the expressivity of typed `control`/`prompt` and that of typed `shift`/`reset`. In the type-free setting, they are known to be equally expressive [7, 15, 4, 6]. However, there exists a big asymmetry in the complexity of these encodings, and a question remained whether `control`/`prompt` is strictly more expressive than `shift`/`reset` under an appropriate typed setting (without recursive types). In this paper we answer this question. Namely, we show:

- typed `control`/`prompt` can macro-express typed `shift`/`reset`, while
- typed `shift`/`reset` cannot macro-express typed `control`/`prompt`,

where the type system for `shift`/`reset` is the most expressive type system by Asai and Kameyama [2]. This result contrasts with the type-free case.

This paper is organized as follows: in Section 2 we briefly explain the control operators for delimited continuations, and in Section 3 we review the functional CPS translation in the literature. The subsequent three sections are original to this paper: in Section 4 we introduce the type systems for the dynamic control operators, and in Section 5 we give several properties for the type systems. In Section 6, we compare the expressive power of typed `control`/`prompt` with typed `shift`/`reset`. Section 8 concludes.

[1] We distinguish (general) recursive types from inductive types in that the former may contain negative occurrences of the type variable being taken the fixed point, for instance, $\mu X.(X \to \texttt{int})$ where μ is for the fixed point operator.

```
fun foo xs =
 let fun visit nil = nil
       | visit (x::xs) = visit (shift (fn k => x::(k xs)))
 in reset (fn () => visit xs) end
```

```
fun bar xs =
 let fun visit nil = nil
       | visit (x::xs) = visit (control (fn k => x::(k xs)))
 in prompt (fn () => visit xs) end
```

Fig. 1. List-copying and list-reversing functions

2 Informal Explanation of `control` and `prompt`

We begin with the examples by Biernacki et al. [6] listed in Figure 1 written in Standard ML syntax.

The functions `foo` and `bar` have type `int list -> int list`, and differ in the names for control operators only: `shift` and `reset` for the former and `control` and `prompt` for the latter. Both `shift` and `control` capture evaluation contexts up to the closest delimiter (`reset` and `prompt`, resp.) Given the list `[1,2,3]`, the function `foo` evaluates as:

```
foo [1,2,3] ⇝ <visit (shift (fn k => 1::(k [2,3])))>
            ⇝ <let k = <visit •> in 1::(k [2,3]) end>
            ⇝ <1::<visit [2,3]>>
            ⇝ <1::<let k = <visit •> in 2::(k [3]) end>>
            ⇝ ...
            ⇝ <1::<2::<3::nil>>>   ⇝ [1,2,3]
```

where `< ... >` denotes the delimiter inserted by `reset`, and `<visit •>` denotes the delimited evaluation context (delimited continuation) captured by `shift` with • being the hole in it. The expression `<`v`>` evaluates to v itself when v is a value, hence the result of this computation is identical to the argument.

The evaluation of `bar` proceeds as follows (the delimiter is denoted by `#`):

```
bar [1,2,3] ⇝ #(visit (control (fn k => 1::(k [2,3]))))
            ⇝ #(let k = (visit •) in 1::(k [2,3]) end)
            ⇝ #(1::(visit [2,3]))
            ⇝ #(let k = (1::(visit •)) in 2::(k [3]) end)
            ⇝ #(2::(1::(visit [3])))
            ⇝ ...
            ⇝ #(3::(2::(1::nil)))   ⇝ [3,2,1]
```

The evaluation context captured by `control` is the whole context since there is no other delimiters. Hence, we obtain a a reversed list as the result of computation.

The operational behavior for each set of control operators can be formalized by reduction rules. Let v and P denote a value and a call-by-value evaluation context such that no delimiter encloses the hole, resp. Then we have the reduction rules as follows:

$$\begin{array}{rr} \text{(For } \texttt{shift/reset}\text{)} & \langle P[\mathcal{S}c.e]\rangle \to \langle \texttt{let } c = \lambda x.\langle P[x]\rangle \texttt{ in } e\rangle \\ \text{(For } \texttt{control/prompt}\text{)} & \#\,(P[\mathcal{F}c.e]) \to \#\,(\texttt{let } c = \lambda x.P[x] \texttt{ in } e) \end{array}$$

Here $\mathcal{S}c.e$ corresponds to `(shift (fn c => e))`, and $\langle e\rangle$ to `(reset (fn () => e))` in the ML implementation in the previous subsection. Similarly, $\#\ e$ is `(prompt (fn () => e))`, and $\mathcal{F}c.e$ is `(control (fn c => e))`.

Besides the names for control operators, the only difference between them is whether the captured delimited continuation has an extra reset or not: $\lambda x.\langle P[x]\rangle$ for the former, and $\lambda x.P[x]$ for the latter. This small difference in syntax raises a big difference in semantics. Suppose P has other occurrences of `shift` or `control`, and the captured delimited continuation is applied to a value v in a then-current continuation E which may have a delimiter that encloses the hole.

- In the former, we evaluate $E[\langle P[v]\rangle]$ in which other occurrences of `shift` in P will be delimited by this `reset` (unless they are "escaped" in function closures). Namely, the corresponding delimiter for these `shift` is determined when the delimited continuation is captured.
- In the latter, we evaluate $E[P[x]]$ in which other occurrences of `control` ought to be delimited by `prompt` in E. (Note that P does not have any `prompt` which encloses the hole). Consequently, the corresponding delimiter for these `control` is determined not when it is captured, but when the delimited continuation is used.

Hence the former is called static, and the latter is called dynamic by analogy with static and dynamic binding in Scheme and Lisp [5]. The static/dynamic nature of control operators has an impact on their implementation. For `shift/reset`, a delimited continuation can be represented by an ordinary, composable function, which leads to a simple CPS translation [9, 10]. For `control/prompt`, we need to keep the captured delimited continuations as they are, until they are actually used, which needs an extra machinery.

3 A CPS translation for `control/prompt`

We regard a CPS translation as the fundamental analysis tool for control operators. For `control` and `prompt`, three such translations are known: Shan's [7], Dybvig, Peyton-Jones, and Sabry's [11], and Biernacki, Danvy and Millikin's [6]. In this paper we use (a variant of) the last one since it is the simplest. We could use Shan's one as well.

$$
\begin{array}{lll}
v ::= d \mid x \mid \lambda x.e & & \text{value} \\
e ::= v \mid e_1 e_2 \mid \mathcal{F}c.e \mid \# \; e \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & & \\
\quad \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & & \text{expression} \\
P ::= [\,] \mid Pe \mid vP & & \\
\quad \mid \texttt{let}\ x = P\ \texttt{in}\ e \mid \texttt{if}\ P\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & & \text{pure evaluation context} \\
E ::= [\,] \mid Ee \mid vE & & \\
\quad \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \# \; E & & \text{evaluation context}
\end{array}
$$

Fig. 2. Syntax of the language with `control`/`prompt`

$$
\begin{array}{rl}
(\lambda x.e)v & \rightsquigarrow e[v/x] \\
\texttt{if true then}\ e_1\ \texttt{else}\ e_2 & \rightsquigarrow e_1 \\
\texttt{if false then}\ e_1\ \texttt{else}\ e_2 & \rightsquigarrow e_2 \\
\texttt{let}\ x = v\ \texttt{in}\ e & \rightsquigarrow e[v/x] \\
\#\ (P[\mathcal{F}c.e]) & \rightsquigarrow \#\ (\texttt{let}\ c = \lambda x.P[x]\ \texttt{in}\ e) \\
\#\ v & \rightsquigarrow v
\end{array}
$$

Fig. 3. Reduction rules

Figure 2 gives the syntax of our source language where d is constant and x and c are variables. The expression $\mathcal{F}c.e$ is a construct for `control` in which c is a bound variable. The expression $\#\ e$ is the one for `prompt`. Variables are bound by λ or $\mathcal{F}$, the set of free variables in e is denoted by $\texttt{FV}(e)$, and we identify α-equivalent expressions. Sequencing $a;\ b$ is an abbreviation of $(\lambda x.b)a$ with $x \notin \texttt{FV}(b)$. In a pure evaluation context P, no `prompt` may enclose its hole $[\,]$, while a (general) evaluation context E allows such occurrences of `prompt`. Figure 3 gives call-by-value operational semantics to this language where $e[v/x]$ represents the result of capture-avoiding substitution.

Figure 4 defines the CPS translation for this language as a variant of the one given by Biernacki et al. The differences are: (1) they gave a 2CPS translation whch is an iterated translation (its image takes two continuations as its arguments), while we use a more traditional 1CPS translation, and (2) we extend the source language with constant, conditional, and `let`.

The target language of this translation is a call-by-value lambda calculus with constants, conditional, and `let` as well as list-manipulating constructs such as `Nil`, cons (denoted by ::), append (@) and a destructor (`case`). Note that the translation for `let` expression is only meaningful for the typed source language (given in the next section), and we have included it in Figure 4 only to save space. For a type-free source language, we can define it as $[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!] = [\![(\lambda x.e_2)e_1]\!]$.

$$
\begin{aligned}
[\![\_]\!] &: \texttt{SourceTerm} \to \texttt{Cont} \to \texttt{Trail} \to \texttt{TargetValue} \\
[\![V]\!] &= \lambda k\ t.\ k\ (V)^*\ t \\
[\![e_1 e_2]\!] &= \lambda k\ t.\ [\![e_1]\!](\lambda m_1\ t_1.\ [\![e_2]\!](\lambda m_2\ t_2.\ m_1\ m_2\ k\ t_2)\ t_1)\ t \\
[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] &= \lambda k\ t.\ [\![e_1]\!](\lambda m_1\ t_1.\ \texttt{if}\ m_1\ \texttt{then}\ [\![e_2]\!]\ k\ t_1\ \texttt{else}\ [\![e_3]\!]\ k\ t_1)\ t \\
[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!] &= \lambda k\ t.\ \texttt{let}\ x = ([\![e_1]\!]\ \theta_1\ \texttt{Nil})\ \texttt{in}\ [\![e_2]\!]\ k\ t \\
[\![\#\ e]\!] &= \lambda k\ t.\ k([\![e]\!]\ \theta_1\ \texttt{Nil})\ t \\
[\![\mathcal{F}c.e]\!] &= \lambda k\ t.\ \texttt{let}\ c = \lambda x\ k'\ t'.\ k\ x\ (t\ @\ (k' :: t'))\ \texttt{in}\ [\![e]\!]\ \theta_1\ \texttt{Nil}
\end{aligned}
$$

$$
\begin{aligned}
(\_)^* &: \texttt{SourceValue} \to \texttt{TargetValue} \\
(d)^* &= d \quad \text{for constant} \\
(x)^* &= x \quad \text{for a variable} \\
(\lambda x.e)^* &= \lambda x\ k\ t.\ [\![e]\!]\ k\ t
\end{aligned}
$$

$$
\begin{aligned}
\theta_1 = \lambda x\ t.\ &\texttt{case}\ t\ \texttt{of} \\
&\mid\ \texttt{Nil} \Rightarrow x \\
&\mid\ (k_1 :: t_1) \Rightarrow k_1\ x\ t_1
\end{aligned}
$$

Fig. 4. 1CPS translation for the language with `control` and `prompt`

Let us see the types in Figure 4, although these types should be considered informal, and only for explanation. The types `SourceTerm`, `SourceValue`, `Cont`, `TargetValue`, are those for terms and values in the source language, and those for continuations and values in the target language, resp.

The type `Trail` is new to their CPS translation. Recall that, in order to represent the dynamic behavior of `control`/`prompt`, we need to keep the delimited continuations until they are used. A trail is a list of delimited continuations to store these continuations. Thus, we may informally define `Cont` and `Trail` as follows:

$$
\begin{aligned}
\texttt{Cont} &= \texttt{TargetValue} \to \texttt{Trail} \to \texttt{TargetValue} \\
\texttt{Trail} &= \texttt{List}(\texttt{Cont})
\end{aligned}
$$

This "definition" needs recursive types, which will be examined in the next section.

Next, we look at the term-level translation in Figure 4. For the constructs other than control operators, the translation is the same as the standard (e.g. Plotkin's) translation except that it passes trails without changing them.

The prompt-term $\#\ e$ initializes its continuation and trail: in its translation $[\![\#\ e]\!]$, $[\![e]\!]$ is applied to θ_1 and the empty trail `Nil`. The continuation θ_1 acts as the identity continuation (the empty evaluation context).

The control-term $\mathcal{F}c.e$ captures a delimited continuation (and initializes the continuation and the trail): it captures the current delimited continuation k for

a future use of c. When c is applied to some value, the captured continuation k is "composed" with the then-current continuation k'. Rather than simply composing two delimited continuations, we store (in the trail) the list of all the captured delimited continuations except the current one, namely, we extend the then-current trail t' to a new trail $t \mathbin{@} (k' :: t')$ and use k as the current continuation. Continuations stored in the trail will be used when the current continuation becomes empty (i.e. θ_1).

Note that we can easily extend our source language and the CPS translation with practical language constructs such as the fixed point operator and primitive functions.

Finally, a CPS translation for a complete program e is defined as $[\![e]\!]\theta_1\mathtt{Nil}$.

4 Type System

We introduce a polymorphic type system for `control`/`prompt` in this section. Types are an important facility in most programming languages to classify terms and also to ensure a certain kind of safety for computation. We think that, for `control`/`prompt` to be used by ordinary programmers, a sound type system is definitely needed.

4.1 Design of Type System

Our strategy to construct a type system for `control`/`prompt` is basically the same as that for `shift`/`reset` [2]: given a term e, we infer the most general type of its CPS translation $[\![e]\!]$, and use this type as the type for e. There is, however, two problems in this strategy: (1) the target terms of the CPS translation for `control`/`prompt` need recursive types if we assign types, and there is no notion of the most general type in a type system with recursive types, and (2) we need arbitrary many type variables to precisely represent types of the CPS translation.

The first problem already appears if we type the identity (delimited) continuation θ_1. Assuming the type of trails is $\mathtt{List}(\tau)$ for some τ, θ_1 must have the type $\alpha \to \mathtt{List}(X) \to \alpha$ for some α, where $X = \alpha \to \mathtt{List}(X) \to \alpha$. Hence we need some sort of recursive types.

The second problem is this: Recall that `Cont` and `Trail` are informally defined as $\mathtt{Cont} = \mathtt{TargetValue} \to \mathtt{Trail} \to \mathtt{TargetValue}$ and $\mathtt{Trail} = \mathtt{List}(\mathtt{Cont})$. When the source language is typed in some way, the type `TargetValue` should be instantiated by more specific types, and in general, a trail has the type:

$$\mathtt{List}(\alpha_1 \to \mathtt{List}(\alpha_2 \to \mathtt{List}(\ldots \mathtt{List}(\alpha_n \to \cdots \to \beta_n) \ldots) \to \beta_2) \to \beta_1)$$

for different α_i and β_i. Apparently, we cannot represent these types in a finite manner.

We give a simple solution to these problems based on the observation that most (if not all) examples with `control`/`prompt` in the literature can be typed under the following simple restriction:

– In a trail type, all α_i and β_i is the same type.

This means that any trail must have type $\mu X.\mathtt{List}(\tau \to X \to \tau)$ for some type τ where μ is for recursive types. In the sequel, we adopt this restriction, and write $\mathtt{Trail}(\tau)$ for this type. Note that the restriction does not constrain the type of continuations, that is, a continuation may have type $\alpha \to \mathtt{Trail}(\tau) \to \beta$ for different types α, β, and τ.

We think that this restriction is not too strong, since, most (if not all) examples with `control`/`prompt` in the literature follow this restriction (list-reversing function and several variations of search programs [6]). Also we will see later that the typed `control`/`prompt` calculus under this restriction can simulate typed `shift`/`reset` calculus, which has many interesting examples.

Under this restriction, the target terms of the CPS translation can be typed in the ordinary type system (with let-polymorphism, but without recursive types), in which the most general type always exists (if typable), and, therefore, the first problem is also solved.

Discussion. An anonymous reviewer has suggested us that, we could exclude the circularity in the trail types by introducing an inductive (not recursive) structure. One way to achieve it is: if we change the definition of the trail type from $\mathtt{Trail} = \mathtt{List}(\mathtt{Cont})$ to

$$\mathtt{Trail}(n) = \mathtt{Trail}(0) \mid (\mathtt{Cont} \times \mathtt{Trail}(n-1))$$

where n denotes the length of the list, then we can avoid the circularity (recursiveness) of the trail type. However, the use of dependent types drastically complicates the type structure of the target calculus (and possibly the corresponding source type system). Moreover, it is not our goal to obtain a strongly normalizing calculus for `control`/`prompt` by constraining the source type system artificially. Rather, we took Biernacki et al.'s CPS translation as a good starting point, and based on it, we tried to construct a natural type system which is harmonious with their CPS translation.

4.2 Definition of Type System

Now we introduce a type system for `control`/`prompt`.

Types and type contexts are defined by:

$$\begin{aligned} \alpha, \beta \cdots ::= b \mid t \mid \alpha \to (\beta, \gamma, \delta/\tau) && \text{monomorphic types} \\ A ::= \alpha \mid \forall t.A && \text{polymorphic types} \\ \Gamma ::= [\,] \mid \Gamma, x : A && \text{type contexts} \end{aligned}$$

where b is a basic type (including `bool`), t is a type variable, and $\alpha \to (\beta, \gamma, \delta/\tau)$ is a function type whose meaning will be made clear later. $\mathtt{FTV}(A)$ denotes the set of free type variables in A.

The type system has two forms of judgements, and the first form is:

$$\Gamma \vdash e : \alpha, \beta, \gamma/\tau$$

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

$$\frac{\Gamma, x : \alpha \vdash e : \beta, \gamma, \delta/\tau}{\Gamma \vdash_p \lambda x.e : \alpha \to (\beta, \gamma, \delta/\tau)}\ \text{fun} \qquad \frac{\Gamma \vdash_p e : \alpha}{\Gamma \vdash e : \alpha, \beta, \beta/\tau}\ \text{exp}$$

$$\frac{\Gamma \vdash e_1 : \alpha \to (\beta, \gamma, \delta/\tau), \epsilon, \rho/\tau \quad \Gamma \vdash e_2 : \alpha, \delta, \epsilon/\tau}{\Gamma \vdash e_1 e_2 : \beta, \gamma, \rho/\tau}\ \text{app}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool}, \delta, \gamma/\tau \quad \Gamma \vdash e_2 : \alpha, \beta, \delta/\tau \quad \Gamma \vdash e_3 : \alpha, \beta, \delta/\tau}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \alpha, \beta, \gamma/\tau}\ \text{if}$$

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

$$\frac{\Gamma \vdash e : \tau, \tau, \beta/\tau}{\Gamma \vdash_p \# e : \beta}\ \text{prompt} \qquad \frac{\Gamma, c : \alpha \to (\tau, \tau, \beta/\tau) \vdash e : \rho, \rho, \gamma/\rho}{\Gamma \vdash \mathcal{F}c.e : \alpha, \beta, \gamma/\tau}\ \text{control}$$

Fig. 5. Type Inference Rules

where Γ is a type context, α, β, γ, and τ are types, and e is an expression. It means that, under the type context Γ, e is an expression of type α, with the answer type modification[2] from β to γ, and the trail type is τ. The roles of the additional types in the judgement can be made clearer by CPS translating it:

$$\Gamma^* \vdash [\![e]\!] : \texttt{Cont}(\alpha^*, \beta^*/\tau^*) \to \texttt{Trail}(\tau^*) \to \gamma^*$$

with

$$\texttt{Trail}(\tau^*) = \mu X.\texttt{List}(\tau^* \to X \to \tau^*)$$
$$\texttt{Cont}(\alpha^*, \beta^*/\tau^*) = \alpha^* \to \texttt{Trail}(\tau^*) \to \beta^*$$

The second form of judgements is:

$$\Gamma \vdash_p e : \alpha$$

which means e is a pure (effect-free) expression of type α. This form is used to introduce let-polymorphism with control effects in a sound manner[2].

Figure 5 gives the type inference rules where $\alpha \leq A$ in the rule (var) means, 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 $\sigma_1, \cdots, \sigma_n$. We assume that, a basic type b is associated with each constant d. The type $\texttt{Gen}(\rho; \Gamma)$ in the rule (let) is defined by $\forall t_1. \cdots \forall t_n.\rho$ where $\{t_1, \cdots, t_n\} = \texttt{FTV}(\rho) - \texttt{FTV}(\Gamma)$.

The rule (exp) allows one to switch from the second form to the first form. Since pure expressions are insensitive to continuations and trails, we can introduce arbitrary types for their answer type (β) and the trail type (τ).

[2] For the standard CPS translation, we say, the answer type is modified in a computation of e from β to γ, if the CPS transform of e has type $(\alpha \to \beta) \to \gamma$. Without control operators for delimited continuations, the types β and γ are equal. With them, they may be different. See Asai and Kameyama [2] for details.

$$\frac{\Gamma \vdash e : \alpha, \beta, \gamma/t}{\Gamma \vdash e : \alpha, \beta, \gamma/*} \text{ star-intro, if } t \notin \mathtt{FTV}(\Gamma, \alpha, \beta, \gamma) \qquad \frac{\Gamma \vdash e : \alpha, \beta, \gamma/*}{\Gamma \vdash e : \alpha, \beta, \gamma/\tau} \text{ star-elim}$$

$$\frac{\Gamma \vdash e : \alpha \to (\beta, \gamma, \delta/*), \epsilon, \rho/\sigma}{\Gamma \vdash e : \alpha \to (\beta, \gamma, \delta/\tau), \epsilon, \rho/\sigma} \text{ star-fun}$$

Fig. 6. Type Inference Rules for $\lambda_{let}^{c/p+}$

Other rules are naturally derived from the type for the CPS translation. The rule (var) is standard. In the rule (fun), the function type $\alpha \to (\beta, \gamma, \delta/\tau)$ extends the ordinary function type $\alpha \to \beta$ to encapsulate the effect of answer type modification from γ to δ with the trail type τ. The rule (app) reflects this intuition. The rule (let) is for ML-like let-polymorphism. As is well known, we must restrict e_1 in the expression $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$ to have a sound type system, and we follow Asai and Kameyama's type system [2] which restricts e_1 to be a pure term, i.e., either a value or $\#\ e$.

For the rule (prompt), look at its CPS translation in Figure 4. It is easy to check θ_1 must have type $\mathtt{Cont}(\tau, \tau/\tau)$ for some τ, and the return type of $[\![e]\!]$ is the same as that of $[\![\#\ e]\!]$. So by letting this return type β, we get the rule (prompt). The rule (control) is more complicated. In the CPS translation of $\mathcal{F}c.e$, a term $\lambda x k' t'.kx(t@(k' :: t'))$ is substituted for c, which poses constraints that t and t' are of the same list type and k' is of its member type. Since we restricted all trails to be of type $\mathtt{Trail}(\tau)$ for some τ, k' has to have the type $\mathtt{Cont}(\tau, \tau/\tau)$ and t and t' have the type $\mathtt{Trail}(\tau)$. Then we can derive the rule (control).

An example type derivation for a concrete term will be given in a later section.

We call the calculus with this type system $\lambda_{let}^{c/p}$.

4.3 Introducing Trail-Polymorphism

The type system of $\lambda_{let}^{c/p}$ can type many useful examples with `control`/`prompt`. However, it cannot express a certain kind of polymorphism in trails. We occasionally want to express that a function has type $\alpha \to (\beta, \gamma, \delta/\tau)$ for any τ, i.e., it is insensitive (or polymorphic) to the trail type. To solve this problem with a small cost, i.e., not without introducing impredicative polymorphism, we introduce a limited form of polymorphism, called trail-polymorphism, into $\lambda_{let}^{c/p}$ as follows.

We add a special type constant "$*$" to the definition of types, which can appear in a function type as $\alpha \to (\beta, \gamma, \delta/*)$, or in a judgement as $\Gamma \vdash e : \alpha, \beta, \gamma/*$. Intuitively, $*$ represents a universally quantified type variable.

We add to $\lambda_{let}^{c/p}$ three new type inference rules listed in Figure 6, which reflect the intuitive meaning of $*$. We call the extended calculus $\lambda_{let}^{c/p+}$. Note that the reductions rules are the same as those for $\lambda_{let}^{c/p}$.

We can also introduce impredicative polymorphism in the same manner as that system for `shift`/`reset`[2]. Since it is orthogonal to the present type system, we omit the details in this paper.

5 Properties

In this section, we state basic properties of $\lambda_{let}^{c/p}$ and $\lambda_{let}^{c/p+}$. Due to lack of space, we only state proof sketches in this paper.

The first property, subject reduction, is by far the most important property in a typed calculus.

Theorem 1 (Subject Reduction). *If $\Gamma \vdash e_1 : \alpha, \beta, \gamma/\tau$ is derived and $e_1 \rightsquigarrow e_2$, then $\Gamma \vdash e_2 : \alpha, \beta, \gamma/\tau$ can be derived.*

The theorem is proved by the standard induction on the derivation. For the case of the reduction $\#\ (P[\mathcal{F}c.e]) \rightsquigarrow \#\ (\texttt{let}\ c = \lambda x.P[x]\ \texttt{in}\ e)$, we decompose it into several smaller reductions as in [3], then the theorem is easy to prove.

The next theorem is the progress property which states the computation of a well typed, closed expression does not get stuck. Here, the word "close" means that the term does not have any free variables and any occurrences of `control` which are not enclosed by `prompt`. To ensure the last property, we restrict our attention to an expression in the form $\# e$. A redex is one of the expressions in the lefthand sides of the reduction rules in Figure 3.

Theorem 2 (Progress). *If $\vdash \# e : \alpha, \beta, \gamma/\tau$ is derivable, then $\# e$ is in the form $E[r]$ where E is an evaluation context and r is a redex.*

Note that, if e is a value v, $\# e$ itself is a redex which reduces to v. The progress property is proved by the standard case analysis.

These two theorems together constitute the strong type soundness property.

The next theorem states that our type system is compatible with the CPS translation. For this purpose, we define the type structure of the target calculus depending on the source calculus by:

- for $\lambda_{let}^{c/p}$, the target calculus is (predicatively) polymorphic lambda calculus with conditionals, the trail type $\texttt{Trail}(\tau)$, and the `list` type.
- for $\lambda_{let}^{c/p+}$, the target calculus is impredicatively polymorphic lambda calculus (second order lambda calculus) with conditionals, the trail type $\texttt{Trail}(\tau)$, and the `list` type.

We define the CPS translation for types and type contexts by:

$$
\begin{aligned}
\alpha^* &= \alpha \quad \text{for basic type and type variable} \\
(\alpha \to (\beta, \gamma, \delta/\tau))^* &= \alpha^* \to (\beta^* \to \texttt{Trail}(\tau^*) \to \gamma^*) \to \texttt{Trail}(\tau^*) \to \delta^* \\
(\alpha \to (\beta, \gamma, \delta/*))^* &= \alpha^* \to \forall X.((\beta^* \to \texttt{Trail}(X) \to \gamma^*) \to \texttt{Trail}(X) \to \delta^*) \\
[\,]^* &= [\,] \\
(\Gamma, x : \forall t_1 \ldots t_n.\alpha)^* &= \Gamma^*,\ x : \forall t_1 \ldots t_n.\alpha^*
\end{aligned}
$$

The third line is for $\lambda_{let}^{c/p+}$ only. We then state type preservation property as a theorem.

Theorem 3 (Type Preservation for CPS Transformation).

- *(For $\lambda_{let}^{c/p}$ and $\lambda_{let}^{c/p+}$) If $\Gamma \vdash e : \alpha, \beta, \gamma/\tau$ is derivable for $\tau \neq *$, then $\Gamma^* \vdash [\![e]\!] : (\alpha^* \to \mathtt{Trail}(\tau^*) \to \beta^*) \to \mathtt{Trail}(\tau^*) \to \gamma^*$ is derivable in the target calculus.*
- *(For $\lambda_{let}^{c/p+}$) If $\Gamma \vdash e : \alpha, \beta, \gamma/*$ is derivable, then $\Gamma^* \vdash [\![e]\!] : (\alpha^* \to \mathtt{Trail}(\tau) \to \beta^*) \to \mathtt{Trail}(\tau) \to \gamma^*$ is derivable for any τ in the target calculus.*

In this paper, we do not state the property that equality is preserved by CPS translation, since it is independent to our type system, and in order to state the property as a theorem, we need to develop a sophisticated equality theory in the target language. For the latter, we need the following property for an arbitrary pure evaluation context P:

$$\lambda x\ k_1\ t_1.\ [\![P[x]]\!]\ \theta_1\ (k_1 :: t_1) = \lambda x\ k_1\ t_1.\ [\![P[x]]\!]\ k_1\ t_1$$

which is about the inductive nature of trails (lists). Since t_1 in this equation is a bound variable, we need to elaborate an inductive theory.

6 Encoding Shift/Reset by Control/Prompt

In type-free setting, `shift`/`reset` can be macro-defined by `control`/`prompt` [4], and we show in this section that it also holds for the typed setting here. We adopt the calculus $\lambda_{let}^{s/r}$ in [2] for `shift`/`reset`, since it is the most liberal type system which allows answer-type modification and let-polymorphism. Its type inference rules are listed in the appendix.

We define a translation from $\lambda_{let}^{s/r}$ to $\lambda_{let}^{c/p+}$ as follows:

$$\begin{array}{rr}
\text{Types} & \overline{(\alpha/\gamma \to \beta/\delta)} = \overline{\alpha} \to (\overline{\beta}, \overline{\gamma}, \overline{\delta}/*) \\
\text{Expressions} & \overline{\mathcal{S}c.e} = \mathcal{F}c'.\mathtt{let}\ c = \lambda x.\#\ (c'x)\ \mathtt{in}\ \overline{e} \\
& \overline{\langle e \rangle} = \#\ \overline{e} \\
\text{Type Contexts} & \overline{\Gamma, x : \forall t_1 \ldots t_n.\alpha} = \overline{\Gamma},\ x : \forall t_1 \ldots t_n.\overline{\alpha}
\end{array}$$

For other constructs, the translation is homomorphic. This translation preserves types and reduction.

Theorem 4.

- *If $\Gamma; \beta \vdash e : \alpha;\ \gamma$ is derivable in $\lambda_{let}^{s/r}$, $\overline{\Gamma} \vdash \overline{e} : \overline{\alpha}, \overline{\beta}, \overline{\gamma}/*$ is derivable in $\lambda_{let}^{c/p+}$.*
- *If $\Gamma; \beta \vdash e_1 : \alpha;\ \gamma$ is derivable and $e_1 \rightsquigarrow e_2$ in $\lambda_{let}^{s/r}$, $\overline{e_1} \rightsquigarrow^* \overline{e_2}$ in $\lambda_{let}^{c/p+}$.*

Proof. We prove this theorem by induction on the derivation of $\Gamma; \beta \vdash e : \alpha;\ \gamma$. We list two key cases here.

(Reset) Suppose we have a derivation in $\lambda^{s/r}_{let}$:

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma;\ \alpha \vdash e : \alpha;\ \beta\end{array}}{\Gamma;\ \gamma \vdash \langle e \rangle : \beta;\ \gamma}\ \text{reset}$$

By induction hypothesis, we can derive $\overline{\Gamma} \vdash \overline{e} : \overline{\alpha}, \overline{\alpha}, \overline{\beta}/*$. Then we can derive:

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma} \vdash \overline{e} : \overline{\alpha}, \overline{\alpha}, \overline{\beta}/*}{\overline{\Gamma} \vdash \overline{e} : \overline{\alpha}, \overline{\alpha}, \overline{\beta}/\overline{\alpha}}}{\overline{\Gamma} \vdash_p \# \ \overline{e} : \overline{\beta}}}{\overline{\Gamma} \vdash \overline{\langle e \rangle} = \# \ \overline{e} : \overline{\beta}, \overline{\gamma}, \overline{\gamma}/*}$$

(Shift) Suppose we have a derivation in $\lambda^{s/r}_{let}$:

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma, c : \forall t.(\beta/t \to \alpha/t);\ \delta \vdash e : \delta;\ \gamma\end{array}}{\Gamma;\ \alpha \vdash \mathcal{S}c.e : \beta;\ \gamma}\ \text{shift}$$

Let $\Delta_1 = \overline{\Gamma}, c : \forall t.(\overline{\beta} \to (\overline{\alpha}, t, t/*))$. By induction hypothesis, we have a derivation for $\Delta_1 \vdash e : \overline{\delta}, \overline{\delta}, \overline{\gamma}/*$ in $\lambda^{c/p+}_{let}$. Let $\Delta_2 = \overline{\Gamma}, c' : \overline{\beta} \to (s, s, \overline{\alpha}/s)$ where s is a fresh type variable, then we can derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Delta_2, x : \overline{\beta} \vdash c'x : s, s, \overline{\alpha}/s}{\Delta_2, x : \overline{\beta} \vdash_p \# \ (c'x) : \overline{\alpha}}}{\Delta_2, x : \overline{\beta} \vdash \# \ (c'x) : \overline{\alpha}, t, t/*}}{\Delta_2 \vdash_p \lambda x.\# \ (c'x) : \overline{\beta} \to (\overline{\alpha}, t, t/*)} \quad \dfrac{\Delta_1 \vdash e : \overline{\delta}, \overline{\delta}, \overline{\gamma}/*}{\Delta_1 \vdash e : \overline{\delta}, \overline{\delta}, \overline{\gamma}/\overline{\delta}}}{\Delta_2 \vdash_p \lambda x.\# \ (c'x) : \overline{\beta} \to (\overline{\alpha}, t, t/*) \quad \Delta_1 \vdash e : \overline{\delta}, \overline{\delta}, \overline{\gamma}/\overline{\delta}}}{\Delta_2 \vdash \texttt{let}\ c = \lambda x.\# \ (c'x)\ \texttt{in}\ \overline{e} : \overline{\delta}, \overline{\delta}, \overline{\gamma}/\overline{\delta}}}{\overline{\Gamma} \vdash \mathcal{F}c'.\texttt{let}\ c = \lambda x.\# \ (c'x)\ \texttt{in}\ \overline{e} : \overline{\beta}, \overline{\alpha}, \overline{\gamma}/s}$$

Since $s \notin \texttt{FTV}(\overline{\Gamma}, \overline{\alpha}, \overline{\beta}, \overline{\gamma})$, we can derive $\overline{\Gamma} \vdash \overline{\mathcal{S}c.e} : \overline{\beta}, \overline{\alpha}, \overline{\gamma}/*$.

We can also show that, if $e \rightsquigarrow e'$ in $\lambda^{s/r}_{let}$, then $\overline{e} \rightsquigarrow^* \overline{e'}$ in $\lambda^{c/p+}_{let}$, whose key case is proved as follows:

$$\begin{aligned}
\overline{\langle P[\mathcal{S}c.e] \rangle} &= \# \ (\overline{P}[\mathcal{F}c'.\texttt{let}\ c = \lambda x.\# \ c'x\ \texttt{in}\ \overline{e}]) \\
&\rightsquigarrow \# \ (\texttt{let}\ c' = \lambda y.\overline{P}[y]\ \texttt{in let}\ c = \lambda x.\# \ c'x\ \texttt{in}\ \overline{e}) \\
&\rightsquigarrow \# \ (\texttt{let}\ c = \lambda y.\# \ (\lambda y.\overline{P}[x])x\ \texttt{in}\ \overline{e}) \\
&\rightsquigarrow \# \ (\texttt{let}\ c = \lambda x.\# \ \overline{P}[x]\ \texttt{in}\ \overline{e}) \\
&= \overline{\langle \texttt{let}\ c = \lambda x.\langle P[x] \rangle\ \texttt{in}\ e \rangle}
\end{aligned}$$

We remark that this proof does not work for $\lambda^{c/p}_{let}$.

7 Typed Control/Prompt is Strictly More Expressive Than Shift/Reset

In the type-free setting, we can encode `control`/`prompt` in terms of `shift`/`reset` [7, 15, 6]. In the typed setting, it is not the case, as we will prove in this section. Since $\lambda_{let}^{s/r}$ is strongly normalizing [2], it is sufficient to construct a typable expression in $\lambda_{let}^{c/p+}$ (or $\lambda_{let}^{c/p}$) whose computation is not terminating.

Let α and $\bullet$, resp. be a type and its inhabitant, resp., for instance, $\alpha = \texttt{bool}$ and $\bullet = \texttt{true}$. Let Γ be the type context $c : \alpha \to (\alpha, \alpha, \alpha/\alpha)$, and recall $e_1;\ e_2$ is an abbreviation of $(\lambda x.e_2)e_1$ for $x \notin \texttt{FV}(e_2)$. We can type the expression $\#\ (\mathcal{F}c.(c\bullet;\ c\bullet);\ \mathcal{F}c.(c\bullet;\ c\bullet))$ in $\lambda_{let}^{c/p}$ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash c : \alpha \to (\alpha, \alpha, \alpha/\alpha), \alpha, \alpha/\alpha} \quad \dfrac{}{\Gamma \vdash \bullet : \alpha}}{\Gamma \vdash c\ \bullet : \alpha, \alpha, \alpha/\alpha}}{\Gamma \vdash c\ \bullet;\ c\ \bullet : \alpha, \alpha, \alpha/\alpha}}{\vdash_p \mathcal{F}c.(c\ \bullet;\ c\ \bullet) : \alpha}}{\vdash \mathcal{F}c.(c\ \bullet;\ c\ \bullet);\ \mathcal{F}c.(c\ \bullet;\ c\ \bullet) : \alpha, \alpha, \alpha/\alpha}}{\vdash \#\ (\mathcal{F}c.(c\ \bullet;\ c\ \bullet);\ \mathcal{F}c.(c\ \bullet;\ c\ \bullet)) : \alpha, \beta, \beta/\tau}$$

The computation of this term does not terminate:

$$\begin{aligned}
&\#\ (\mathcal{F}c.(c\bullet;\ c\bullet);\ \mathcal{F}c.(c\bullet;\ c\bullet)) \\
&\rightsquigarrow \#\ (\texttt{let}\ c = \lambda u.(u;\ \mathcal{F}c.(c\bullet;\ c\bullet))\ \texttt{in}\ (c\bullet;\ c\bullet)) \\
&\rightsquigarrow^* \#\ (\mathcal{F}c.(c\bullet;\ c\bullet);\ (\lambda u.(u;\ \#\ \mathcal{F}c.(c\bullet;\ c\bullet)))\bullet) \\
&\rightsquigarrow^* \#\ (\mathcal{F}c.(c\bullet;\ c\bullet);\ (\lambda u'.(u';\ (\lambda u.(u;\ \#\ \mathcal{F}c.(c\bullet;\ c\bullet)))\bullet))\bullet) \\
&\rightsquigarrow^* \ldots
\end{aligned}$$

Since this example does not use answer-type modification or polymorphism, it can be typed in a more restricted type system such as our previous one [20].

We can go a step further. After submission of this paper, Kiselyov [16] and the second author (Yonezawa) have independently constructed fixed point combinators in call-by-value:

$$\begin{aligned}
(Kiselyov)\ \ &\texttt{Y}_1 = \lambda f.\#\ (Z_f;\ Z_f) \quad \text{where}\ \ Z_f = \mathcal{F}c.\ f\ (\lambda x.\ \#\ (c\bullet;\ c\bullet)\ x) \\
(Yonezawa)\ \ &\texttt{Y}_2 = \lambda f.\#\ (X_f;\ X_f) \quad \text{where}\ \ X_f = \mathcal{F}c.\ \lambda x.\ (f\ \#\ (c\bullet;\ c\bullet))\ x
\end{aligned}$$

Both $\texttt{Y}_1$ and $\texttt{Y}_2$ are typable in $\lambda_{let}^{c/p}$, and satisfy $\texttt{Y}_1\ f\ x = f\ (\lambda x.\ \texttt{Y}_1\ f\ x)\ x$, and $\texttt{Y}_2\ f\ x = f\ (\lambda x.\ f\ (\texttt{Y}_2\ f)\ x)\ x$. Therefore, they can serve as fixed point combinators. The former satisfies a simpler equation, while $\texttt{Y}_1\ e$ may not terminate for some term e, but $\texttt{Y}_2\ e$ always terminates.

8 Conclusion

We have introduced a polymorphic type system for `control`/`prompt`, which allows answer type modification and does not need recursive types. We have shown

that our calculus enjoys type soundness and is compatible with the CPS translation, and that typed `control`/`prompt` is strictly more powerful than typed `shift`/`reset` in the absence of recursive types. Although we cannot claim that our type system is *the* only type system for `control`/`prompt`, we believe that ours can be a good starting point to study the type structure of these control operators.

Based on this work, Kiselyov and the second author have successfully shown that $\lambda_{let}^{c/p}$ is Turing-complete if we extend the calculus with integers and primitive functions. The situation is similar to the exception mechanism in ML, for which Lillibridge [17] has proved that typed (unchecked) exception can simulate all type-free lambda terms, and therefore can represent all Turing-computable functions. Thielecke also compared the expressive powers of several control operators using a different technique [19].

In this paper we have been concentrating on the foundational aspect of `control`/`prompt` in this paper, and the practical aspect of our type systems is unstudied. In particular, more application programs other than those by Biernacki, Danvy, Millikin's are called for, but it is left for future work.

Acknowledgements: We thank Kenichi Asai, Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan for their insights. Special thanks go to Oleg Kiselyov and anonymous reviewers for valuable comments. This work was partly supported by JSPS Grant-in-Aid for Scientific Research (C) 16500004.

References

1. K. Asai. On Typing Delimited Continuations: Three New Solutions to the Printf Problem. Technical Report OCHA-IS 07-1, Department of Information Science, Ochanomizu University, September 2007.
2. K. Asai and Y. Kameyama. Polymorphic Delimited Continuations. In *Proc. Asian Programming Languages and Systems, LNCS 4807*, pages 239–254, Nov-Dec 2007.
3. K. Asai and Y. Kameyama. Polymorphic Delimited Continuations. Technical Report CS-TR-07-10, Dept. of Computer Science, University of Tsukuba, Sep 2007.
4. D. Biernacki and O. Danvy. A Simple Proof of a Folklore Theorem about Delimited Control. *J. Funct. Program.*, 16(3):269–280, 2006.
5. D. Biernacki, O. Danvy, and C. c. Shan. On the Static and Dynamic Extents of Delimited Continuations. *Science of Computer Programming*, 60(3):274–297, 2006.
6. D. Biernacki, O. Danvy, and K. Millikin. A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations. *TOPLAS*, to appear.
7. C. c. Shan. Shift to control. In *Proc. Workshop on Scheme and Functional Programming*, pages 99–107, 2004.
8. O. Danvy. Functional Unparsing. *J. Funct. Program.*, 8(6):621–625, 1998.
9. O. Danvy and A. Filinski. Abstracting Control. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, 1990.
10. O. Danvy and A. Filinski. Representing Control: a Study of the CPS Transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.
11. R. K. Dybvig, S. Peyton Jones, and A. Sabry. A Monadic Framework for Delimited Continuations. *J. Funct. Program.*, to appear.
12. M. Felleisen. The Theory and Practice of First-Class Prompts. In *Proc. 15th Symposium on Principles of Programming Languages*, pages 180–190, 1988.

13. A. Filinski. Representing Monads. In *POPL*, pages 446–457, 1994.
14. Y. Kameyama and M. Hasegawa. A sound and complete axiomatization for delimited continuations. In *ICFP*, pages 177–188, 2003.
15. O. Kiselyov. How to remove dynamic prompt: Static and dynamic delimited continuation operators are equally expressive. Technical Report 611, Computer Science Department, Indiana University, March 2005.
16. O. Kiselyov. Fixpoint combinator from typed prompt/control. 2007. `http://okmij.org/ftp/Computation/Continuations.html`.
17. M. Lillibridge. Unchecked Exceptions Can Be Strictly More Powerful Than Call/CC. *Higher-Order and Symbolic Computation*, 12(1):75–104, 1999.
18. C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford Univ. Comput. Lab., Oxford, England, 1974. Reprinted in Higher-Order and Symbolic Computation 13(1/2):135–152, 2000.
19. H. Thielecke. On Exceptions Versus Continuations in the Presence of State. In *ESOP*, pages 397–411, 2000.
20. T. Yonezawa and Y. Kameyama. A Type System for Dynamic Delimited Continuations. *IPSJ Transactions on Programming, Information Processing Society of Japan*, to appear.

A Polymorphic Type System for `shift/reset`

We define the type system for $\lambda_{let}^{s/r}$ in [2] except the fixed point operator. Types and type contexts are defined by:

$$\begin{array}{rr} \alpha, \beta \cdots ::= b \mid t \mid (\alpha/\beta \to \gamma/\delta) & \text{monomorphic types} \\ A ::= \alpha \mid \forall t.A & \text{polymorphic types} \\ \Gamma ::= [\,] \mid \Gamma, x : A & \text{type contexts} \end{array}$$

where the function type $(\alpha/\beta \to \gamma/\delta)$ corresponds to $\alpha \to (\gamma, \beta, \delta/*)$ in $\lambda_{let}^{c/p+}$. Judgements in $\lambda_{let}^{s/r}$ are either $\Gamma; \alpha \vdash e : \beta;\ \gamma$ or $\Gamma \vdash_p e : \beta$. The former corresponds to $\Gamma \vdash e : \beta, \alpha, \gamma/*$ in $\lambda_{let}^{c/p+}$. Finally, Figure 7 gives several important type inference rules of $\lambda_{let}^{s/r}$.

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

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

Fig. 7. Type Inference Rules of $\lambda_{let}^{s/r}$.