Regular Paper

A Type System for Dynamic Delimited Continuations

Takuo Yonezawa† and Yukiyoshi Kameyama†

We study the control operators "control" and "prompt" which manage part of continuations, that is, delimited continuations. They are similar to the well-known control operators "shift" and "reset", but differ in that the former is dynamic, while the latter is static. In this paper, we introduce a static type system for "control" and "prompt" which does not use recursive types. We design our type system based on the dynamic CPS transformation recently proposed by Biernacki, Danvy and Millikin. We also introduce let-polymorphism into our type system, and show that our type system satisfies several important properties such as strong type soundness.

1. Introduction

We are interested in control operators that manage (e.g. capture, discard, or reinstall) delimited continuations, part of an evaluation context. They allow one to represent various kinds of control structures such as non-determinism, logging, and a base framework to represent other effects (be they control effects or not) in direct style.

In the literature, two sets of delimited continuation operators have attracted researchers' interest. The first such operators are "control" and "prompt" proposed by Felleisen in 1988[1]. They are classified as *dynamic* operators, since, when a captured delimited continuation is used, it is merged with the current continuation with no delimiter (prompt) between them. Therefore we cannot regard each delimited continuation as an independent component: they will be dynamically merged. The second operators "shift" and "reset" proposed by Danvy and Filinski[2] in 1990 are classified as *static* operators, since a delimited continuation, when used, is composed with the current continuation, i.e., it can be regarded as an ordinary function, or a static object. Due to their static nature, shift and reset have a simple CPS transformation, which allows one to study the the representation of all monadic effects[3], abstract machine[4] and equational axiomatization[5], together with several interesting programming examples, extensively. On the other hand, control and prompt have not been active research targets until Biernacki, Danvy and Millikin[6] recently found a CPS transformation for them, together with several new applications, and there is still room for foundational studies on them.

This paper introduces a type system for control and prompt, and studies its properties. In the literature, studies on the control operators for delimited continuations were mostly done in type-free languages or with very restricted type systems. Recently, Asai and Kameyama[7] have proposed a polymorphic type system for shift and reset and shown that it satisfies several properties such as type soundness and strong normalization. However, it is still an interesting open question as to whether we can define a static polymorphic type system for control and prompt, since control/prompt is inherently dynamic. One might need recursive types which would drastically change the static nature of the source calculus. In this paper, we solve this question by introducing a static type system which does not need recursive types. Our type system is much more expressive than the one implicitly used by Biernacki, Danvy, Millikin[6].

The contribution of this paper can be summarized as follows:

- We propose a type system for control/prompt which does not need recursive types. We show that our type system displays strong type soundness.
- We introduce ML-like let-polymorphism into the type system, and show that it also satisfies properties above.
- We compare the expressibility of control/prompt and shift/reset under the typed framework.

We also emphasize that this paper is the first one which studies the type theoretic foundation of control/prompt.

This paper is organized as follows: We define

† Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba

$$
\begin{array}{lll}
M ::= & & \text{(expression)} \\
& x & \text{(variable)} \\
& \mid \lambda x.M & (\lambda \text{ abstraction}) \\
& \mid M\ M & \text{(function application)} \\
& \mid \text{let}\, x = v \,\text{in}\, M & \text{(let)} \\
& \mid \#M & \text{(prompt)} \\
& \mid \mathcal{F}c.M & \text{(control)} \\
v ::= x \mid \lambda x.M & & \text{(value)}
\end{array}
$$

Fig. 1 Syntax of the language

the language with control/prompt in Section 2. We introduce the dynamic CPS transformation in Section 3. In Section 4, we define the polymorphic type system for control/prompt. We discuss the relation of our type system with that for shift/reset in Section 5. Then we show several properties of the type system in Section 6. The conclusion appears in Section 7.

2. Definition of the Language

In this section, we define the language with control/prompt and show some examples.

Fig. 1 defines the syntax of the language where x and c are variables. Following the ML families, we restrict the let expression let $x = v$ in M by the "value restriction", where the substituted expression v must be a value.

We introduce the call-by-value reduction rules of the language. **Fig. 2** defines evaluation context, pure evaluation context (evaluation context without prompt, "pure e-context" in short), redex, and **Fig. 3** defines reduction rules. We define $M_1 \to^* M_2$ as the reflexive-transitive closure of $\to$, and the equality $M_1 = M_2$ as the least congruence relation which contains $\to$.

2.1 Example

Control/prompt can be used as follows:

$$
\begin{array}{l}
\#(\\
\quad \mathcal{F}c.((c\ 1) + 2) \\
\quad +3 \\
) + 4
\end{array}
$$

For the prompt expression $\#M$, the subexpression M is evaluated first. If it contains no control operator $\mathcal{F}$, the value of M is returned. Otherwise, the control operator $\mathcal{F}$ captures the continuation up to the most recent prompt ($[\,]+3$ in this example), which is in turn assigned to c as a function $(\lambda x.\ x + 3)$. Then the body of the control expression $((c\ 1) + 2)$ is evaluated under this assignment. When its evaluation is finished, its then-current continuation up to the prompt $([\ ] + 3)$ is discarded, so that the value of the prompt expression is $((\lambda x.x + 3)\ 1) + 2$. Finally, the entire result is $(((\lambda x.x + 3)\ 1) + 2) + 4 = 10$.

The captured continuation is called a delimited continuation since it is delimited by prompt.

Note that the captured delimited continuation is a function so that it can be used arbitrarily many times. We can simulate nondeterministic computation by using the delimited continuation many times, or an exception-like effect by never using it.

2.2 Web Application

A more practical example is web application[8].

Consider a simple console application

$$
\begin{array}{l}
x = \text{input}(); \\
y = \text{input}(); \\
\text{print}(x + y);
\end{array}
$$

If we want the application to be a web application, we must suspend the program after printing an input-form, wait for the user's submission, and resume the program when the user submits some value.

A naive solution to implement such a web

$$
\begin{array}{lll}
E ::= & & \text{(evaluation context)} \\
& [\,] & \\
& \mid E\ M & \\
& \mid v\ E & \\
& \mid \#E & \\
P ::= & & \text{(pure e-context)} \\
& [\,] & \\
& \mid P\ M & \\
& \mid v\ P & \\
R ::= & & \text{(redex)} \\
& (\lambda x.M)\ v & \\
& \mid \#v & \\
& \mid \text{let}\, x = v \,\text{in}\, M & \\
& \mid \#(P[\mathcal{F}c.M]) &
\end{array}
$$

Fig. 2 Evaluation contexts and redex

$$
\begin{array}{rl}
(\lambda x.M)v & \to M[v/x] \\
\text{let}\, x = v \,\text{in}\, M & \to M[v/x] \\
\#v & \to v \\
\#(P[\mathcal{F}c.M]) & \to \#(M[\lambda x.P[x]/c]) \\
& \quad \text{where } x \text{ is fresh.}
\end{array}
$$

where $M[v/x]$ represents the capture-avoiding substitution.

Fig. 3 Reduction rules

application is to use threads, however it has a problem. After submitting a value for x, a user may come back to the input-form page for x (by using the browser's "Back" button), and may submit another value for x. The second value for x is, however, stored to y with the thread implementation.

A better solution is to use continuations. We re-define the input procedure as follows:

```
input() =
$\mathcal{F}c.$(
  generate a fresh continuation id;
  save c to a global associative table
   with the continuation id;
  print the input-form with
   the continuation id as
   a hidden parameter;
)
```

and the main routine resumes the saved continuation when the user submits a value. Since continuation can be called arbitrarily many times, the program will work correctly even if the user submits values multiple times.

The captured continuation above is a continuation up to, but excluding, the main routine or other request-specific codes. Hence it is a delimited continuation[9] rather than an unlimited continuation. We can implement the application with control/prompt: we enclose the session-specific code (i.e. two input commands for x and y and a print command) by prompt, and use the control operator $\mathcal{F}$ to capture delimited continuations.

2.3 List Reversing

The control operator "shift" (denoted by $\mathcal{S}$) is a similar operator to control but it delimits the captured continuation with prompt. Its reduction rule is defined as follows:

$$\#(P[\mathcal{S}c.M]) \rightarrow \#(M[\lambda x.\#(P[x])/c]) \quad \text{where } x \text{ is fresh.}$$

Let us compare it to one for control:

$$\#(P[\mathcal{F}c.M]) \rightarrow \#(M[\lambda x.P[x]/c])$$

Shift and control behave differently for the following example:

```
reverse $l$ =
  let
    visit = $\lambda l'$
      case $l'$ of
        Nil $\Rightarrow$ Nil
        $(x : x')$ $\Rightarrow$
          visit($\mathcal{F}k.(x :: (k\ x'))$)
  in
    #(visit $l$)
```

The example is a list reversing function. If we replace control with shift, the function becomes a list copying function.

Biernacki et al[10] have shown that we can express the depth-first (and breadth-first, respectively) traversal by using shift/reset (and control/prompt, respectively).

3. CPS Transformation

In this section, we introduce a CPS transformation for control and prompt proposed by Biernacki, Danvy and Millikin[6].

A CPS transformation is a syntax-directed program transformation from a source calculus to a target calculus in continuation passing style (CPS), i.e., all functions explicitly pass continuations as their arguments. In a typical case, the source calculus is a lambda calculus with control operators, and the target calculus is one without. Then the control operators are given precise semantics through this CPS transformation (so called CPS semantics).

For control/prompt, only non-functional style CPS transformations were previously known until Biernacki, Danvy and Millikin[6] discovered a purely functional one in 2005. They called it a *dynamic* CPS transformation, and we introduce its curried version here.

The dynamic CPS transformation is based on the 2-CPS transformation. While a standard (1-CPS) transformation maps a term to a function term which takes one continuation as its argument, a 2-CPS transformation maps a term to a function which takes two continuation parameters as its arguments, one for the first level continuation and the other for the second level continuation (or meta-continuation). 1-CPS is sufficient to give semantics for undelimited continuation operators like call/cc, while 2-CPS is the key to interpret control operators for delimited continuations like shift/reset and control/prompt.

The dynamic CPS transformation is an extension of 2-CPS transformation with the no-

$$
\begin{array}{lll}
M ::= & & \text{(expression)}\\
 & x & \text{(variable)}\\
 & \mid \lambda x.M & \text{(λ abstraction)}\\
 & \mid M\ M & \text{(function application)}\\
 & \mid \ \mathrm{let}\, x = M \,\mathrm{in}\, M & \text{(let)}\\
 & \mid L & \text{(list)}\\
 & \mid \ \mathrm{case}\, M \,\mathrm{of} & \text{(case)}\\
 & \quad \mathrm{Nil} \Rightarrow M & \\
 & \quad (x :: x) \Rightarrow M & \\
L ::= & \mathrm{Nil} \mid M :: L &
\end{array}
$$

Fig. 4 Definition of the target language

tion of "trail", a list of continuations, which will be explained later. The source calculus of the dynamic CPS transformation is the call-by-value lambda calculus with control and prompt given in the previous section, and its target calculus is the lambda calculus with lists (for representing trails). **Fig. 4** defines the syntax of the target calculus.

Fig. 5 shows the dynamic CPS transformation as a type-free transformation. Note that the results of the transformation (except those for control and prompt) is identical to Plotkin's call-by-value CPS transformation if we η-reduce the results.

A transformed expression takes (if uncurried) three arguments, a continuation, a list of continuations or *trail*, and a meta-continuation, and returns a result value (answer).

The key feature of this transformation is the introduction of trails. A trail is part of a continuation (i.e. captured by the $\mathcal{F}$ operator), but is represented as a list of delimited continuations. The reason for representing it as a list of continuations rather than a single, composed continuation comes from the dynamic nature of the $\mathcal{F}$ operator.

Let us examine how a trail is changed through a computation. Initially (without invocation of control operators) a trail is empty. But when control operators are invoked, a delimited continuation is captured and when the captured delimited continuation is invoked, it is added to a trail to form a non-empty trail, and it dynamically changes the behavior of the initial (identity) continuation so that it passes a value to the continuations in the trail.

Although we gave the transformation as a type-free one, an informal typing would help understand its meaning. Informally, if an expression M has a type τ and its answer type is α, the type of the transformed expression $[\![M]\!]$ is

$$
\begin{aligned}
\mathrm{Ans}\ \alpha &::= \mathrm{Trail}\ \alpha \rightarrow (\mathrm{Cont}_2\ \alpha\ *) \rightarrow *\\
\mathrm{Trail}\ \alpha &::= \mathrm{List}\ (\mathrm{Cont}_1\ \alpha\ \alpha)\\
\mathrm{Cont}_1\ \alpha\ \beta &::= \alpha \rightarrow \mathrm{Ans}\ \beta\\
\mathrm{Cont}_2\ \alpha\ \beta &::= \alpha \rightarrow \beta
\end{aligned}
$$

Note that Ans is defined recursively.

Fig. 6 Answer type of the dynamic CPS transformation

$\mathrm{Cont}_1\ \tau\ \alpha \rightarrow \mathrm{Ans}\ \alpha$ where Ans is informally defined as **Fig. 6**. We note that all continuations in a trail must be of the same type, continuations from α to α, since all the elements of a list (trail) should have the same type. This informal discussion will be reflected in the design of our type system in the next section. We think this is a natural restriction, but it may delimit the extent of the applicability of our type system, which will be discussed in Section 4.

4. Type System

Now we introduce the type system for the language with control/prompt based on the dynamic CPS transformation in Section 3 so that it satisfies the following property: a source expression is typable whenever its CPS transformation is typable in the target calculus.

Since the transformed expressions mention the type of "answers" (final results of computations), our type system for the source language has to also mention the answer type. Therefore, we extend the notation of type judgments and function types: a type judgment

$$\Gamma \vdash M : \tau/\alpha$$

is read as: under a type environment Γ, the expression M has a type τ and its answer type is α. A function type $\sigma \rightarrow \tau/\alpha$ similarly represents the type of a function which takes a value of σ as the argument, returns a value of τ, and its answer type is α. For pure expressions (i.e. values and prompt), we use a pure judgment

$$\Gamma \vdash_p M : \tau$$

to represent that the answer type of pure expressions can be arbitrary.

Fig. 7 defines the syntax of types. Greek alphabets $(\alpha, \beta, \gamma, \dots)$ denote types in this paper.

Fig. 8 defines the typing rules. The typing rules are defined to reflect the CPS transformation. For example, the expression $[\![\mathcal{F}x.M]\!]$ in the target language is typable as follows:

$$
\begin{aligned}
[\![x]\!] =& \lambda k_1\ t_1\ k_2.k_1\ x\ t_1\ k_2 \\
[\![\lambda x.M]\!] =& \lambda k_1\ t_1\ k_2.k_1\ (\lambda x.[\![M]\!])\ t_1\ k_2 \\
[\![M_0 M_1]\!] =& \lambda k_1\ t_1\ k_2.[\![M_0]\!]\ (\lambda v_0\ t_1'\ k_2'.[\![M_1]\!]\ (\lambda v_1\ t_1''\ k_2''.v_0\ v_1\ k_1\ t_1''\ k_2'')t_1'\ k_2')\ t_1\ k_2 \\
[\![\mathrm{let}\, x = v \,\mathrm{in}\, M]\!] =& \lambda\ k_1\ t_1\ k_2. \\
& \mathrm{let}\, x = v^* \\
& \mathrm{in}\ [\![M]\!]\ k_1\ t_1\ k_2 \\
[\![\#M]\!] =& \lambda k_1\ t_1\ k_2.[\![M]\!]\ \theta_1\ \mathrm{Nil}\ (\lambda v.k_1\ v\ t_1\ k_2) \\
[\![\mathcal{F}x.M]\!] =& \lambda k_1\ t_1\ k_2. \\
& \mathrm{let}\, x = \lambda v\ k_1'\ t_1'\ k_2'.k_1\ v\ (t_1 @ (k_1' :: t_1'))\ k_2' \\
& \mathrm{in}\ [\![M]\!]\ \theta_1\ \mathrm{Nil}\ k_2 \\
\theta_1 =& \lambda v\ t_1\ k_2.\,\mathrm{case}\, t_1\, \mathrm{of} \\
& \mathrm{Nil} \Rightarrow k_2\ v \\
& (k_1 :: t_1') \Rightarrow k_1\ v\ t_1'\ k_2 \\
x^* =& x \\
(\lambda x.\ M)^* =& \lambda x.\ [\![M]\!]
\end{aligned}
$$

where @ is the list concatenation.

Fig. 5 Definition of dynamic CPS transformation

$\tau ::=$	(monomorphic type)
t	(type variable)
$\mid (\tau \to \tau/\tau)$	(function type)
$A ::=$	(polymorphic type)
$\tau \mid \forall t.A$	
$\Gamma ::=$	(type environment)
$\emptyset$	(empty type environment)
$\mid \Gamma[x \mapsto A]$	(type env. extension)

Fig. 7 Syntax of type and type environment

$$
\frac{\begin{array}{c}\vdots \\ \emptyset[x \mapsto (\tau \to \mathrm{Cont}_1\ \alpha\ \alpha \to \mathrm{Ans}\ \alpha)] \vdash \\ M : \mathrm{Cont}_1\ \beta\ \beta \to \mathrm{Ans}\ \alpha \\ \vdots\end{array}}{\emptyset \vdash [\![\mathcal{F}x.M]\!] : \mathrm{Cont}_1\ \tau\ \alpha \to \mathrm{Ans}\ \alpha}
$$

We can prove that none of the types of the transformed expressions match with $\mathrm{Cont}_1\ \beta\ \beta \to \mathrm{Ans}\ \alpha$ but only with $\mathrm{Cont}_1\ \alpha\ \alpha \to \mathrm{Ans}\ \alpha$, so that we can type it as:

$$
\frac{\begin{array}{c}\vdots \\ \emptyset[x \mapsto (\tau \to \mathrm{Cont}_1\ \alpha\ \alpha \to \mathrm{Ans}\ \alpha)] \vdash \\ M : \mathrm{Cont}_1\ \alpha\ \alpha \to \mathrm{Ans}\ \alpha \\ \vdots\end{array}}{\emptyset \vdash [\![\mathcal{F}x.M]\!] : \mathrm{Cont}_1\ \tau\ \alpha \to \mathrm{Ans}\ \alpha}
$$

Mapping $\Gamma \vdash [\![X]\!] : \mathrm{Cont}_1\ \tau\ \alpha \to \mathrm{Ans}\ \alpha$ to $\Gamma \vdash X : \tau/\alpha$, we obtain a typing rule

$$
\frac{\Gamma[c \mapsto (\tau \to \alpha/\alpha)] \vdash M : \alpha/\alpha}{\Gamma \vdash \mathcal{F}c.M : \tau/\alpha}
$$

Other rules can be derived similarly.

Moreover, we introduce ML-like let polymorphism with value-restriction. In Fig. 8, $\mathrm{Gen}(\sigma;\Gamma)$ is $\forall t_1. \ldots \forall t_n.\sigma$ where $\{t_1, \ldots, t_n\} = \mathrm{FTV}(\sigma) - \mathrm{FTV}(\Gamma)$, $FTV(\alpha)$ denotes the set of free type variables in α, and $\tau \le A$ means that A is $\forall t_1. \ldots \forall t_n.\rho$ and τ is $\rho[\sigma_1, \ldots \sigma_n/t_1, \ldots, t_n]$ for some monomorphic types $\rho, \sigma_1, \ldots, \sigma_n$ and type variables $t_1, \ldots, t_n$.

Discussion: Answer Type Modification Answer type modification is a control effect where the (final) answer type of an expression differs from the return type of the current continuation. For a term without control operators, these two types always agree as shown by the type of its CPS transformation $(\alpha \to X) \to X$. With control operators, however, these two types may differ, for instance, Asai[11] has shown that a direct style implementation of `printf` in ML needs answer type modification.

Our type system does not support answer type modification. This restriction comes from the restriction that the types of continuations in a trail must be the same as the type of continuations from α to α. Eliminating the restriction is left for future work. ☆

5. Relation to shift/reset

In this section, we discuss the relationship between prompt/control and shift/reset.

In the type-free setting, shift/reset and control/prompt are known to be equally expressive, namely, each set of control operators can simulate the other[13]. To see this, we identify reset with prompt, and simulate shift by control and

☆ See our forthcoming paper[12].

$$\frac{\Gamma(x) = A \quad \tau \leq A}{\Gamma \vdash_p x : \tau} \text{ variable} \qquad \frac{\Gamma[x \mapsto \sigma] \vdash M : \tau/\alpha}{\Gamma \vdash_p \lambda x.M : (\sigma \to \tau/\alpha)} \ \lambda \text{ abstraction} \qquad \frac{\Gamma \vdash_p M : \tau}{\Gamma \vdash M : \tau/\alpha} \text{ from pure}$$

$$\frac{\Gamma \vdash M_1 : (\sigma \to \tau/\alpha)/\alpha \quad \Gamma \vdash M_2 : \sigma/\alpha}{\Gamma \vdash M_1\ M_2 : \tau/\alpha} \text{ function application}$$

$$\frac{\Gamma \vdash_p v : \sigma \quad \Gamma[x \mapsto \mathrm{Gen}(\sigma; \Gamma)] \vdash M : \tau/\alpha}{\Gamma \vdash \mathrm{let}\, x = v \,\mathrm{in}\, M : \tau/\alpha} \text{ let}$$

$$\frac{\Gamma \vdash M : \tau/\tau}{\Gamma \vdash_p \#M : \tau} \text{ prompt} \qquad \frac{\Gamma[c \mapsto (\tau \to \alpha/\alpha)] \vdash M : \alpha/\alpha}{\Gamma \vdash \mathcal{F}c.M : \tau/\alpha} \text{ control}$$

Fig. 8 Typing rules

prompt as follows:

$$\mathcal{S}c.M = \mathcal{F}c'.\,\mathrm{let}\, c = \lambda x.\#(c'\ x)\,\mathrm{in}\, M$$

Then it is easy to see that shift/reset can be simulated by control/prompt.

The converse direction is also possible, but is rather complicated, for instance, the encoding of control in terms of shift/reset needs recursive types if we type the encoding.

Here we show that the one-way simulation (simulating shift/reset by control/prompt) is possible even under the polymorphic type system.☆ Namely, the expression $\mathcal{F}c'.\,\mathrm{let}\, c = \lambda x.\#(c'\ x)\,\mathrm{in}\, M$ can be typed in our type system as

$$\begin{array}{c} \Gamma[c' \mapsto \dots][c \mapsto \forall\beta.(\tau \to \alpha/\beta)] \vdash M : \alpha/\alpha \\ \vdots \\ \Gamma \vdash \mathcal{F}c'.\,\mathrm{let}\, c = \lambda x.\#(c'\ x)\,\mathrm{in}\, M : \tau/\alpha \end{array}$$

This type is essentially equivalent to the type for the shift expression in Murthy's type system restricted to level 1 (with polymorphism), or the one in Asai and Kameyama's type system without answer type modification, which can be defined as follows:

$$\frac{\Gamma[c \mapsto \forall\beta.(\tau \to \alpha/\beta)] \vdash M : \alpha/\alpha}{\Gamma \vdash \mathcal{S}c.M : \tau/\alpha} \text{ shift}$$

Note that Asai and Kameyama's type system uses $\forall$ both in the rule for let and in the rule for shift (i.e. $\forall\beta$ at the type of c), while our system uses it only in the rule for let. In fact, we can derive polymorphism of the answer type of continuations captured by shift from the typing rules for let, reset, and control. Hence, we can say that the let polymorphism and the type of control effects (control and prompt) are orthogonal in our type system.

☆ At the time of submitting this paper, we did not know if the converse simulation was possible or not. After that, we came up with a counterexample of the converse simulation, which is detailed in our forthcoming paper[12].

6. Properties of the Type System

Here we show three important properties of the type system:

- Strong type soundness.
- Preservation of types with respect to CPS transformation.
- Preservation of equality with respect to CPS transformation.

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

Type soundness is the most important property for static type systems for programming languages, stated as follows:

Theorem 1 (Type Soundness). *If $\emptyset \vdash_p \#M : \tau$ is derivable, then either M is a value (a variable or lambda abstraction), or there exists some term N such that $\#M \to \#N$ and $\emptyset \vdash_p \#N : \tau$.*

The theorem states strong type soundness in the sense the type of an expression is preserved through evaluation, while weak type soundness means that a well typed program does not go wrong[14].

Theorem 1 is a combination of the following two properties:

Theorem 2 (Subject Reduction). *If $\Gamma \vdash M_1 : \tau/\alpha$ is derivable and $M_1 \to^* M_2$, then $\Gamma \vdash M_2 : \tau/\alpha$ is derivable. If $\Gamma \vdash_p M_1 : \tau$ is derivable and $M_1 \to^* M_2$, then $\Gamma \vdash_p M_2 : \tau$ is derivable.*

Theorem 3 (Progress). *If $\emptyset \vdash_p \#M : \tau$ is derivable, then either M is a value, or $\#M$ can be uniquely decomposed into the form $E[R]$ where E is an evaluation context and R is a redex.*

The progress property above takes a slightly unusual form in that the expression being considered is not an arbitrary one M, but an expression in the form of $\#M$, i.e., we only consider $\#M$ with no free variables as a *program.* In fact, Felleisen assumes that the operator $\#$ is always supplied from the top level, and thus called this operator *prompt.*

Next, we show the CPS transformation given above is well behaved, namely, it preserves types and equality. Before stating the property, we need to define the CPS transformation for types. For this purpose, let us recall the informal definition of types given in Fig. 6. By expanding the informal definition of Trail, we get:

$$\begin{aligned} \text{Trail } \alpha &= \text{List } (\text{Cont}_1\ \alpha\ \alpha) \\ &= \text{List } (\alpha \to \text{Ans } \alpha) \\ &= \text{List } (\alpha \to \text{Trail } \alpha \\ &\qquad \to (\alpha \to *) \to *) \end{aligned}$$

Based on this intuition, we formally define the type constructor Trail as:

$\text{Trail } \alpha ::= \mu X.\text{List } (\alpha \to X \to (\alpha \to *) \to *)$

where $\mu X.\dots$ is a recursive type, and $*$ is the answer type. We also regard the informal definitions for the other type constructors Ans, Cont_1, and Cont_2 as formal. Then it is easy to see the formal definition coincides with the informal one (modulo the folding/unfolding of recursive types).

Note on Answer Type Polymorphism: Since we use so called 2-CPS transformation (it uses two continuation variables k_1 and k_2), the final answer type $*$ used to type the target terms of the CPS transformation can be made polymorphic. Thus, if we were to work in the second order lambda calculus, we could bind all $*$'s by universal type quantifier as $\forall *.\text{Trail } \alpha$. Alternatively we can *locally* quantify the type variable $*$ such as $\cdots \to \forall *.((\alpha \to *) \to *))$.

For more details about answer type polymorphism, see Thielecke[15].

Now we define the CPS transformation for types:

$$\begin{aligned} t^* &= t \quad \text{for type variable} \\ (\alpha \to \beta/\gamma)^* &= \alpha^* \to (\text{Cont}_1\ \beta^*\ \gamma^*) \\ &\qquad \to (\text{Ans } \gamma^*) \\ (\forall t.\alpha)^* &= \forall t.\alpha^* \end{aligned}$$

The CPS translation can be naturally extended to type context Γ.

Then we can state the type preservation property with respect to the CPS transformation. The formal definition of the type system of the target language is given by **Fig. 9** and **Fig. 10** in the appendix.

Theorem 4 (Preservation of Types). *If* $\Gamma \vdash M : \tau/\alpha$ *is derivable, then we can derive* $\Gamma^* \vdash [\![M]\!] : (\text{Cont}_1\ \tau^*\ \alpha^*) \to (\text{Ans } \alpha^*)$ *in the target calculus.*

Similarly, if $\Gamma \vdash_p M : \tau$ *is derivable, then we can derive* $\Gamma^* \vdash [\![M]\!] : (\text{Cont}_1\ \tau^*\ \alpha) \to (\text{Ans } \alpha)$ *for any type* α *in the target calculus.*

Note that, the types in the target calculus implicitly contains the type variable $*$.

We can also show that equality in the source calculus is preserved with respect to the CPS transformation.

We define the equality of expressions of the target language as the least congruence relation which includes β-equality and the following additional equality:

$$[\![M]\!]\ k_1\ t_1\ k_2 = [\![M]\!]\ \theta_1\ (k_1 :: t_1)\ k_2$$

for a source expression M, a continuation k_1 of type $\text{Cont}_1\ \alpha\ \beta$, a trail t_1 of type Trail α, a meta-continuation k_2 of type $\text{Cont}_2\ \alpha\ \beta$.

The additional equality can be justified by the following argument: since the continuation k_1 is used linearly at the tail position with trail t_1 appended to some trail at the end, or captured by control with a trail t_1 appended to some trail at the end. In the former case, both sides of the additional equality are clearly equal. In the later case, since trails are destroyed (used) by θ_1 only, and $(k_1\ v\ t_1\ k_2)$ is equal to $(\theta_1\ v\ (k_1 :: t_1)\ k_2)$, both sides of the equality are equal. In this paper, we regard the equation as an axiom for simplicity.

Theorem 5 (Preservation of Equality). *If* $\Gamma \vdash M_i : \tau/\alpha$ *is derivable for* $i = 1, 2$ *and* $M_1 = M_2$*, then we have* $[\![M_1]\!] = [\![M_2]\!]$ *in the target calculus.*

7. Conclusion

We have proposed a type system for a language with control/prompt based on the dynamic CPS transformation, and showed several important properties of the type system such as type soundness, relation to the CPS transformation, and relation to shift/reset under the typed framework. We emphasize that our type system does not need recursive types, but does have let polymorphism. We have also shown that answer type polymorphism of shift can be derived from let polymorphism.

Future work includes studies on the following topics:

- Type inference algorithm for our type system.
- Answer type modification: the type system in this paper does not allow modification of

answer types. We are working on a type system which does not have this restriction[12].

- Multi-level control operators.
- More general polymorphism.

Acknowledgments We would like to thank the anonymous referees and the attendees of the SIG-Programming meeting for useful comments. This work was partly supported by JSPS Grant-in-Aid for Scientific Research (C) 16500004.

References

1) Felleisen, M.: The Theory and Practice of First-Class Prompts, *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, ACM, pp. 180–190 (1988).
2) Danvy, O. and Filinski, A.: Abstracting Control, *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, New York, NY, USA, ACM, pp. 151–160 (1990).
3) Filinski, A.: Representing Monads, *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, ACM, pp. 446–457 (1994).
4) Biernacka, M., Biernacki, D. and Danvy, O.: An Operational Foundation for Delimited Continuations, *Proceedings of the 4th ACM SIGPLAN Workshop on Continuations, Technical Report CSR-04-1, School of Computer Science, University of Birmingham*, pp. 25–34 (2004).
5) Kameyama, Y. and Hasegawa, M.: A Sound and Complete Axiomatization of Delimited Continuations, *ICFP '03: Proceedings of the 8th ACM SIGPLAN international conference on Functional programming*, New York, NY, USA, ACM, pp. 177–188 (2003).
6) Biernacki, D., Danvy, O. and Millikin, K.: A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations, Technical Report BRICS-RS-05-16, BRICS, University of Aarhus, Denmark (2005).
7) Asai, K. and Kameyama, Y.: Polymorphic Delimited Continuations, *APLAS '07: Proceedings of 5th Asian Symposium on Programming Languages and Systems, LNCS 4807*, pp. 239–254 (2007).
8) Queinnec, C.: The Influence of Browsers on Evaluators or, Continuations to Program Web Servers, *ICFP '00: Proceedings of the 5th ACM SIGPLAN international conference on Functional programming*, New York, NY, USA, ACM, pp. 23–33 (2000).
9) Kiselyov, O., chieh Shan, C. and Sabry, A.: Delimited Dynamic Binding, *ICFP '06: Proceedings of the 11th ACM SIGPLAN international conference on Functional programming*, New York, NY, USA, ACM, pp. 26–37 (2006).
10) Biernacki, D., Danvy, O. and Shan, C.: On the Static and Dynamic Extents of Delimited Continuations, Technical Report BRICS-RS-05-36, BRICS, University of Aarhus, Denmark (2005).
11) Asai, K.: On Typing Delimited Continuations: Three New Solutions to the Printf Problem (2007). Submitted. See `http://pllab.is.ocha.ac.jp/~asai/papers/`.
12) Kameyama, Y. and Yonezawa, T.: Typed Dynamic Control Operators for Delimited Continuations (2008). FLOPS '08: Proceedings of 9th International Symposium on Functional and Logic Programming.
13) Biernacki, D. and Danvy, O.: A Simple Proof of a Folklore Theorem about Delimited Control, Technical Report BRICS-RS-05-25, BRICS, University of Aarhus, Denmark (2005).
14) Wright, A. K. and Felleisen, M.: A Syntactic Approach to Type Soundness, *Information and Computation*, Vol. 115, No. 1, pp. 38–94 (1994).
15) Thielecke, H.: From Control Effects to Typed Continuation Passing, *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, ACM, pp. 139–149 (2003).

Appendix

A.1 Proof

We first prove several lemmas which are necessary to prove subject reduction.

Lemma 1 (Weakening of Type Environment). *If $\Gamma \vdash M : \tau/\alpha$ is derivable and a variable x does not freely occur in M, $\Gamma[x \mapsto \sigma] \vdash M : \tau/\alpha$ is derivable.*

If $\Gamma \vdash_p M : \tau$ is derivable and a variable x does not freely occur in M, $\Gamma[x \mapsto \sigma] \vdash_p M : \tau$ is derivable.

Proof. By structural induction on the derivation. □

Lemma 2 (Substitution for Monomorphic Variable). *If $\Gamma_1 \vdash_p v : \beta$ and $\Gamma_2[x \mapsto \beta] \vdash M : \tau/\alpha$ is derivable and $\Gamma_1 \subseteq \Gamma_2$, then $\Gamma_2 \vdash M[v/x] : \tau/\alpha$ is derivable.*

If $\Gamma_1 \vdash_p v : \beta$ and $\Gamma_2[x \mapsto \beta] \vdash_p M : \tau$ is derivable and $\Gamma_1 \subseteq \Gamma_2$, then $\Gamma_2 \vdash_p M[v/x] : \tau$ is derivable.

Proof. By mutual structural induction on the derivation of M and Lemma 1.

We omit the proof since it is almost the same as (and simpler than) the proof of Lemma 3. □

Lemma 3 (Substitution for Polymorphic Variable). *If $\Gamma_1 \vdash_p v : \sigma$ and $\Gamma_2[x \mapsto \mathrm{Gen}(\sigma;\Gamma_1)] \vdash M : \tau/\alpha$ is derivable and $\Gamma_1 \subseteq \Gamma_2$, then $\Gamma_2 \vdash M[v/x] : \tau/\alpha$ is derivable.*

If $\Gamma_1 \vdash_p v : \sigma$ and $\Gamma_2[x \mapsto \mathrm{Gen}(\sigma;\Gamma_1)] \vdash_p M : \tau$ is derivable and $\Gamma_1 \subseteq \Gamma_2$, then $\Gamma_2 \vdash_p M[v/x] : \tau$ is derivable.

Proof. From $\Gamma_1 \vdash_p v : \sigma$, we can derive $\Gamma_2 \vdash v : \beta/\alpha$ by Lemma 1. We use structural induction on the type derivation of M.

Case 1. If the last derivation is a derivation for a variable and $M = x$, then we can derive $\Gamma_2 \vdash_p M[v/x] : \tau$ since $\tau \leq \mathrm{Gen}(\sigma;\Gamma_1)$ and we can systematically replace type variables $\mathrm{FTV}(\tau) - \mathrm{FTV}(\Gamma)$ in the derivation of $\Gamma_2 \vdash v : \beta$ to arbitrary type variables. If M is a variable with $M \neq x$, it is obvious.

Case 2. The other cases are proved by simple induction, since in all typing rules, type environment used in the conclusion is smaller than or equal to one in the premises. □

Lemma 4. *If $\Gamma \vdash P[M] : \tau/\alpha$ is derivable, then $\Gamma \vdash M : \sigma/\alpha$ is derivable for some σ.*

Proof. By structural induction on P. □

Lemma 5. *If $\Gamma \vdash P[M] : \tau/\alpha$ is derivable and $\Gamma \vdash M : \sigma/\alpha$ is derivable, $\Gamma \vdash_p \lambda x.P[x] : (\sigma \rightarrow \tau/\alpha)$ is derivable for a fresh variable x.*

Proof. Case analysis on P.

Case 1. If $P = [\,]$, it is obvious.

Case 2. If $P = P'\ M'$, the last part of the derivation of $\Gamma \vdash P[M] : \tau/\alpha$ must be:

$$\frac{\Gamma \vdash P'[M] : (\rho \rightarrow \tau/\alpha)/\alpha \quad \Gamma \vdash M' : \rho/\alpha}{\Gamma \vdash P'[M]\ M' : \tau/\alpha}$$

By the assumption of induction, we can derive $\Gamma \vdash \lambda x.P'[x] : (\sigma \rightarrow (\rho \rightarrow \tau/\alpha)/\alpha$ and $\Gamma[x \mapsto \sigma] \vdash P'[x] : (\rho \rightarrow \tau/\alpha)/\alpha$.

Thereby, we can derive $\Gamma \vdash_p \lambda x.P'[x]\ M' : (\sigma \rightarrow \tau/\alpha)$, that is, $\Gamma \vdash_p \lambda x.P[x] : (\sigma \rightarrow \tau/\alpha)$ as follows:

$$\dfrac{\dfrac{\begin{array}{r}\Gamma[x \mapsto \sigma] \vdash P'[x] : (\rho \rightarrow \tau/\alpha)/\alpha \\ \Gamma[x \mapsto \sigma] \vdash M' : \rho/\alpha\end{array}}{\Gamma[x \mapsto \sigma] \vdash P'[x]\ M' : \tau/\alpha}}{\Gamma \vdash_p \lambda x.P'[x]\ M' : (\sigma \rightarrow \tau/\alpha)}$$

Case 3. If $P = v\ P'$, we can prove it similarly to the case for $P = P'\ M$. □

Theorem 2 (Subject Reduction). *If $\Gamma \vdash M_1 : \tau/\alpha$ is derivable and $M_1 \rightarrow^* M_2$, then $\Gamma \vdash M_2 : \tau/\alpha$ is derivable.*

Proof. We show the case of one-step reduction. The other case can be derived straightforwardly.

Now we will perform case analysis on the reduction rule used in the one-step reduction.

Case 1 *(β reduction and elimination of let).* If the reduction is the β reduction $((\lambda x.M)v \rightarrow M[v/x])$, the statement holds by Lemma 2. If the reduction is the elimination of let $(\mathrm{let}\, x = v \,\mathrm{in}\, M \rightarrow M[v/x])$, the statement holds by Lemma 3.

Case 2 *(elimination of prompt).* If the reduction is the elimination of prompt $(\#v \rightarrow v)$, then, by assumption, we have the following derivation:

$$\dfrac{\dfrac{\begin{array}{c}\vdots \\ \Gamma \vdash_p v : \tau\end{array}}{\Gamma \vdash v : \tau/\tau}}{\Gamma \vdash_p \#v : \tau}$$

Therefore, the statement holds in this case.

Case 3 *(elimination of control).* If the reduction is the elimination of control $(\#P[\mathcal{F}c.M] \rightarrow \#M[\lambda x.P[x]/c])$, then we have the following derivation by the assumption and Lemma 4.

$$\dfrac{\begin{array}{c}\dfrac{\Gamma[c \mapsto (\pi \rightarrow \tau/\tau)] \vdash M : \tau/\tau}{\Gamma \vdash \mathcal{F}c.M : \pi/\tau} \\ \vdots \\ \Gamma \vdash P[\mathcal{F}c.M] : \tau/\tau\end{array}}{\Gamma \vdash_p \#P[\mathcal{F}c.M] : \tau}$$

Applying Lemma 5, we can derive

$$\Gamma \vdash_p \lambda x.P[x] : (\pi \rightarrow \tau/\tau)$$

and applying Lemma 2, we can derive

$$\Gamma \vdash M[\lambda x.P[x]/c] : \tau/\tau$$

Hence we can derive the following judgment by applying the typing rule for prompt.

$$\Gamma \vdash_p \#M[\lambda x.P[x]/c] : \tau$$

Consequently, Subject Reduction holds. □

Theorem 3 (Progress). *If $\emptyset \vdash_p \#M : \tau$ is derivable, then either M is a value, or $\#M$ can be uniquely decomposed into the form $E[R]$ where E is an evaluation context and R is a*

$$\begin{array}{lr}
\tau ::= & \text{(monomorphic type)} \\
\quad t & \text{(type variable)} \\
\quad \mid (\tau \to \tau/\tau) & \text{(function type)} \\
\quad \mid \text{List } \tau & \text{(list)} \\
\quad \mid \mu t.\tau & \text{(recursive type)} \\
A ::= & \text{(polymorphic type)} \\
\quad \tau \mid \forall t.A & \\
\Gamma ::= & \text{(type environment)} \\
\quad \emptyset & \text{(empty type environment)} \\
\quad \mid \Gamma[x \mapsto A] & \text{(type environment)}
\end{array}$$

Fig. 9 Syntax of type and type environment of the target language

redex.

Proof. We can prove that if M is not a value, we can decompose it to the unique forms $P[R]$, $P[\mathcal{F}c.M']$, or $P[\#M']$.

If it is decomposed to $P[R]$, the theorem holds.

If it is decomposed to $P[\mathcal{F}c.M']$, combining P with the outer $\#$, it becomes a redex $\#P[\mathcal{F}c.M']$ and the theorem holds.

If it is decomposed to $P[\#M']$, if M' is a value, it is trivial. Otherwise, we recursively decompose M' to $E'[R]$, and let $E[\,]$ be $P[E'[\,]]$, and the theorem holds. □

Proof (Theorem 4), Preservation of Types.
We prove that, if $\Gamma \vdash M : \tau/\alpha$ is derivable in the source calculus, then $\Gamma^* \vdash [\![M]\!] : (\text{Cont}_1\ \tau^*\ \alpha^*) \to (\text{Ans}\ \alpha^*)$ is derivable in the target calculus. This is proved by induction on the derivation of $\Gamma \vdash M : \tau/\alpha$, and we state several important cases here.

Case 1 *($M = \#M'$).* We have the following derivation:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash M' : \tau/\tau}}{\Gamma \vdash_p \#M' : \tau}$$

and by the induction hypothesis, we have:

$$\Gamma^* \vdash [\![M']\!] : (\text{Cont}_1\ \tau^*\ \tau^*) \to (\text{Ans}\ \tau^*)$$

in the target calculus.

Note that θ_1, defined in Fig. 5 has type $\text{Cont}_1\ \alpha\ \alpha$ for any type α, and therefore $\vdash \theta_1 : \text{Cont}_1\ \tau^*\ \tau^*$ is derivable. We also have $\vdash \text{Nil} : \text{Trail}\ \tau^*$ and

$$\begin{array}{r}
\emptyset[k_1 \mapsto \text{Cont}_1\ \tau^*\ \tau^*][t_1 \mapsto \text{Trail}\ \tau^*] \\
[k_2 \mapsto \tau^* \to *] \\
\vdash \lambda v.k_1 v t_1 k_2 : (\tau^* \to *)
\end{array}$$

Then it is easy to see that:

$$\begin{array}{l}
\Gamma^* \vdash \lambda k_1\ t_1\ k_2.[\![M']\!]\ \theta_1\ \text{Nil}\ (\lambda v.k_1\ v\ t_1\ k_2) \\
\quad : (\text{Cont}_1\ \tau^*\ \tau^*) \to (\text{Ans}\ \tau^*)
\end{array}$$

Case 2 *($M = \mathcal{F}x.M'$).* We have the following derivation:

$$\dfrac{\dfrac{\vdots}{\Gamma[c \mapsto (\tau \to \alpha/\alpha)] \vdash M' : \alpha/\alpha}}{\Gamma \vdash \mathcal{F}c.M' : \tau/\alpha}$$

By the induction hypothesis, we have:

$$\begin{array}{l}
\Gamma^*[c \mapsto \tau^* \to (\text{Cont}_1\ \alpha^*\ \alpha^*) \to (\text{Ans}\ \alpha^*)] \\
\quad \vdash [\![M']\!] : (\text{Cont}_1\ \alpha^*\ \alpha^*) \to (\text{Ans}\ \alpha^*)
\end{array}$$

in the target calculus.

Let $\Delta = \Gamma^*[k_1 \mapsto \text{Cont}_1\ \tau^*\ \alpha^*, t_1 \mapsto \text{Trail}\ \alpha^*, k_2 \mapsto \alpha^* \to *]$ and $\Sigma = \Delta[c \mapsto \tau^* \to (\text{Cont}_1\ \alpha^*\ \alpha^*) \to (\text{Ans}\ \alpha^*)]$.

Then, we can derive:

$$\begin{array}{l}
\Delta \vdash \lambda v\ k_1'\ t_1'\ k_2'.k_1\ v\ (t_1 @ (k_1' :: t_1'))\ k_2' \\
\quad : \tau^* \to (\text{Cont}_1\ \alpha^*\ \alpha^*) \to (\text{Ans}\ \alpha^*) \\
\Sigma \vdash [\![M']\!]\ \theta_1\ \text{Nil}\ k_2 : *
\end{array}$$

Then we can derive:

$$\Gamma^* \vdash [\![\mathcal{F}x.M]\!] : (\text{Cont}_1\ \tau^*\ \alpha^*) \to (\text{Ans}\ \alpha^*)$$

Case 3 *($M = \mathbf{let}\, x = v \,\mathbf{in}\, M$).* We have the following derivation:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash_p v : \sigma} \quad \dfrac{\vdots}{\Gamma[x \mapsto \text{Gen}(\sigma;\Gamma)] \vdash M : \tau/\alpha}}{\Gamma \vdash \mathbf{let}\, x = v \,\mathbf{in}\, M : \tau/\alpha}$$

By induction hypothesis, we have:

$$\Gamma^* \vdash v^* : \sigma^*$$

and

$$\begin{array}{l}
\Gamma^*[x \mapsto \text{Gen}(\sigma^*;\Gamma^*)] \\
\quad \vdash [\![M]\!] : (\text{Cont}_1\ \tau^*\ \alpha^*) \to (\text{Ans}\ \alpha^*)
\end{array}$$

in the target calculus.

Let $\Delta = \Gamma^*[k_1 \mapsto \text{Cont}_1\ \tau^*\ \alpha^*, t_1 \mapsto \text{Trail}\ \alpha^*, k_2 \mapsto \alpha^* \to *]$.

Then we can easily derive $\Delta\ [x \mapsto \text{Gen}(\sigma^*;\Gamma^*)] \vdash [\![M]\!]\ k_1\ t_1\ k_2 : *$.

Altogether, we can derive:

$$\dfrac{\dfrac{\begin{array}{c}\Delta \vdash v^* : \sigma^* \\ \Delta[x \mapsto \text{Gen}(\sigma^*;\Gamma^*)] \vdash [\![M]\!]\ k_1\ t_1\ k_2 : *\end{array}}{\begin{array}{l}\Delta \vdash \mathbf{let}\, x = v^* \\ \quad \mathbf{in}\, [\![M]\!]\ k_1\ t_1\ k_2 : *\end{array}}}{\begin{array}{l}\Gamma^* \vdash [\![\mathbf{let}\, x = v \,\mathbf{in}\, M]\!] \\ \quad : (\text{Cont}_1\ \tau^*\ \alpha^*) \to (\text{Ans}\ \alpha^*)\end{array}}$$

□

Proof (Theorem 5), Preservation of Equality.
We show the interesting case only, that is,

$$\begin{array}{l}
M_1 = \#P[\mathcal{F}c.\ M'] \\
M_2 = \#M'[(\lambda x.\ P[x])/c]
\end{array}$$

We let

$$[\![P[M]]\!] = \lambda k_1\ t_1\ k_2.\ [\![M]\!]\ (f\ k_1)\ t_1\ k_2$$

We then transform $[\![M_1]\!]$ as follows:

$$\frac{\Gamma(x)=A \quad \tau \le A}{\Gamma \vdash x : \tau}\ \text{variable} \qquad \frac{\Gamma[x \mapsto \sigma] \vdash M : \tau}{\Gamma \vdash \lambda x.M : (\sigma \to \tau)}\ \lambda\ \text{abstraction}$$

$$\frac{\Gamma \vdash M_1 : (\sigma \to \tau) \quad \Gamma \vdash M_2 : \sigma}{\Gamma \vdash M_1\ M_2 : \tau}\ \text{function application} \qquad \frac{\Gamma \vdash M_1 : \sigma \quad \Gamma[x \mapsto \mathrm{Gen}(\sigma;\Gamma)] \vdash M_1 : \tau}{\Gamma \vdash \mathrm{let}\, x = M_1 \,\mathrm{in}\, M_2 : \tau}\ \text{let}$$

$$\frac{}{\Gamma \vdash \mathrm{Nil} : \mathrm{List}\ \tau}\ \text{empty list} \qquad \frac{\Gamma \vdash M_1 : \tau \quad \Gamma \vdash M_2 : \mathrm{List}\ \tau}{\Gamma \vdash (M_1 :: M_2) : \mathrm{List}\ \tau}\ \text{list}$$

$$\frac{\Gamma \vdash M_0 : \mathrm{List}\ \sigma \quad \Gamma \vdash M_1 : \tau \quad \Gamma[x \mapsto \sigma][y \mapsto \mathrm{List}\ \sigma] \vdash M_2 : \tau}{\Gamma \vdash \mathrm{case}\, M_0 \,\mathrm{of}\ \ \mathrm{Nil} \Rightarrow M_1\ (x :: y) \Rightarrow M_2 : \tau}\ \text{case}$$

$$\frac{\Gamma \vdash M : \tau[\mu t.\tau/t]}{\Gamma \vdash M : \mu t.\tau}\ \text{roll} \qquad \frac{\Gamma \vdash M : \mu t.\tau}{\Gamma \vdash M : \tau[\mu t.\tau/t]}\ \text{unroll}$$

Fig. 10 Typing rules of target language

$$\begin{aligned}
[\![M_1]\!] =&\lambda k_1\ t_1\ k_2.\\
&(\lambda k_1'\ t_1'\ k_2'.\ [\![\mathcal{F}c.\ M']\!]\ (f\ k_1')\ t_1'\ k_2')\\
&\theta_1\ \ \mathrm{Nil}\ (\lambda v.\ k_1\ v\ t_1\ k_2)\\
=&\lambda k_1\ t_1\ k_2.\ [\![\mathcal{F}c.\ M']\!]\\
&(f\ \theta_1)\ \ \mathrm{Nil}\ (\lambda v.\ k_1\ v\ t_1\ k_2)\\
=&\lambda k_1\ t_1\ k_2.\\
&\mathrm{let}\, c = \lambda v\ k_1'\ t_1'\ k_2'.\ (f\ \theta_1)\ v\\
&\qquad (k_1' :: t_1')\ k_2'\\
&\mathrm{in}\, [\![M']\!]\ \theta_1\ \ \mathrm{Nil}\ (\lambda v.\ k_1\ v\ t_1\ k_2)\\
=&\lambda k_1\ t_1\ k_2.\\
&[\![M']\!][(\lambda v\ k_1'\ t_1'\ k_2'.\ (f\ \theta_1)\ v\\
&\qquad (k_1' :: t_1')\ k_2')/c]\\
&\theta_1\ \ \mathrm{Nil}\ (\lambda v.\ k_1\ v\ t_1\ k_2)
\end{aligned}$$

Moreover, we transform $[\![M_2]\!]$ as follows:

$$\begin{aligned}
[\![M_2]\!]\ =&\ \lambda k_1\ t_1\ k_2.\\
&[\![M'[(\lambda x.\ P[x])/c]]\!]\\
&\theta_1\ \ \mathrm{Nil}\ (\lambda v.\ k_1\ v\ t_1\ k_2)\\
=&\ \lambda k_1\ t_1\ k_2.\\
&[\![M']\!][(\lambda v.\ [\![P[v]]\!])/c]\\
&\theta_1\ \ \mathrm{Nil}\ (\lambda v.\ k_1\ v\ t_1\ k_2)\\
=&\ \lambda k_1\ t_1\ k_2.\\
&[\![M']\!][(\lambda v\ k_1'\ t_1'\ k_2'.\ (f\ k_1')\ v\\
&\qquad t_1'\ k_2')/c]\\
&\theta_1\ \ \mathrm{Nil}\ (\lambda v.\ k_1\ v\ t_1\ k_2)
\end{aligned}$$

Hence, we only need show the following equation.

$$\begin{aligned}
&(\lambda v\ k_1'\ t_1'\ k_2'.\ (f\ k_1')\ v\ t_1'\ k_2')\\
=&(\lambda v\ k_1'\ t_1'\ k_2'.\ (f\ \theta_1)\ v\ (k_1' :: t_1')\ k_2'
\end{aligned}$$

We can transform these expressions as follows:

$$\begin{aligned}
&(\lambda v\ k_1'\ t_1'\ k_2'.\ (f\ k_1')\ v\ t_1'\ k_2')\\
=&(\lambda v\ k_1'\ t_1'\ k_2'.\ [\![P[v]]\!]\ k_1'\ t_1'\ k_2')\\
&(\lambda v\ k_1'\ t_1'\ k_2'.\ (f\ \theta_1)\ v\ (k_1' :: t_1')\ k_2')\\
=&(\lambda v\ k_1'\ t_1'\ k_2'.\ [\![P[v]]\!]\ \theta_1\ (k_1' :: t_1')\ k_2')
\end{aligned}$$

By the definition of the equality of the target language, we have that these two expressions are equal. □

(Received ??? ??, ????)
(Accepted ??? ??, ????)

Takuo Yonezawa is a Graduate Student of the Master's Program at the Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba.
He is interested in type systems and computational effects of programming languages.
e-mail: yone@logic.cs.tsukuba.ac.jp
URL: http://logic.cs.tsukuba.ac.jp/~yone/

Yukiyoshi Kameyama is an Associate Professor of Computer Science at the Graduate School of Systems and Information Engineering, University of Tsukuba. He is interested in programming logic and software verification. He is a member of ACM, JSSST, and IEICE.
e-mail: kameyama@acm.org
URL: http://logic.cs.tsukuba.ac.jp/~kam/