ATM without Tears: Prompt-Passing Style Transformation for Typed Delimited-Control Operators

Ikuo Kobori* Yukiyoshi Kameyama* Oleg Kiselyov†

Abstract

The salient feature of delimited-control operators is their ability to *modify* answer types during computation. The feature, answer-type modification (ATM for short), allows one to express various interesting programs such as typed printf compactly and nicely, while it makes it difficult to embed these operators in standard functional languages.

In this paper, we present a typed translation of delimited-control operators shift and reset with ATM into a familiar language with multi-prompt shift and reset without ATM, which lets us use shift and reset with ATM in standard languages without modifying the whole type system. Our translation generalizes Kiselyov's direct-style implementation of typed printf, which uses two prompts to emulate the modification of answer types. We prove that our translation preserves typing, and also give an implementation in the tagless-final style which respects typing.

*University of Tsukuba, Japan
†Tohoku University, Japan

Contents

List of Figures

1 Introduction

Delimited continuations is part of continuations, the rest of computation, and delimited-control operators provide programmers a means to access the current delimited continuations. Since the delimited-control operators control/prompt and shift/reset have been proposed around 1990 [8, 7], many researchers have been studying them intensively, to find interesting theory and application in program transformation, partial evaluation, code generation, and computational linguistics. Today, we see their implementations in many programming languages such as Scheme, Racket, SML, OCaml, Haskell, and Scala.

However, there still exists a big gap between theory and practice if we work in typed languages. Theoretically, the salient feature of delimited-control operators is their ability to modify answer types. The term `reset (3 + shift k -> k)` looks as if it has type `int`, but the result of this computation is a continuation `fun x -> reset (3 + x)` whose type is `int -> int`, which means that the initial answer type has been modified during the computation of the shift term. While this feature, called Answer-Type Modification, allows one to express surprisingly interesting programs such as typed printf, it is the source of the problem that we cannot embed the delimited-control operators in standard languages. We can hardly expect that the whole type system of a full-fledged language would be modified in such a way. With a few exceptions of Scala [12] and OchaCaml [11], we cannot directly express the beautiful examples with ATM as programs in standard languages.

This paper addresses this problem, and presents a solution for it. We will give a translation from the language with ATM shift and reset into another language with multi-prompt shift and reset without ATM. The translation is a generalization of Kiselyov's implementation of typed printf, which introduces two prompts (tags for control operators) for the answer types before and after the computation. The resulting term passes prompts of control operators during computation, and following Continuation-Passing Style, we call is Prompt-Passing Style (PPS).

The rest of this paper is organized as follows: Section 2 explains delimited-control operators and answer-type modification by a simple example. Section 3 informally states how we simulate answer-type modification using multi-prompt shift and reset, and Section 4 gives a formal account to it including formal properties. Section 5 describes the syntax-directed translation and its property. Based on the theoretical development, Section 6 gives a tagless-final implementation of shift and reset with answer-type modification as well as several programming examples. Section 7 gives related work and concluding remarks.

2 Delimited-Control Operators and Answer-type Modification

We introduce a simple example which uses delimited-control operators shift and reset where the answer types are modified through computation.

The following implementation of the `append` function is taken from Asai and Kameyama's paper [1].

```
let rec append lst = match lst with
  | [] -> shift (fun k -> k)
  | x :: xs -> x :: append xs
in let append123 =
  reset (append [1;2;3])
in
  append123 [4;5;6]
```

The function `append` takes a value of type `int list` as its input, and traverses the list. When it reaches at the end of the list, it captures the continuation (`fun ys -> reset 1 :: 2 :: 3 :: ys` in the functional form) up to the nearest reset, and returns the continuation as its result. We then apply it to the list `[4;5;6]` to obtain `[1;2;3;4;5;6]`, and it is easy to see that the function deserves its name.

Let us check the type of `append`. At the beginning, the return type of `append` (called its answer type) is `int list`, since in the second branch of the case analysis, it returns `x :: append xs`. However, the final result is a function from list to list, which is different from our initial guess. The answer type has been modified during the execution of the program.

Since its discovery, this feature has been used in many interesting examples with shift and reset, from typed printf to suspended computations, to coroutines, and even to computational linguistics. Nowadays, it is considered as one of the most attractive features of shift and reset.

Although the feature, answer-type modification, is interesting and sometimes useful, it is very hard to directly embed such control operators in conventional functional programming languages such as OCaml, as it requires a big change of the type system; a typing judgment in the form $\Gamma \vdash e : \tau$ must be changed to a more complex form $\Gamma \vdash e : \tau; \alpha, \beta$ where α and β designate the answer types before and after the execution of e. Although adjusting a type system in this way is straightforward in theory, it is rather difficult to modify existing implementations of type systems, and we therefore need a way to represent the above features in terms of standard features and/or mild extensions of existing programming languages.

This paper addresses this problem, and proposes a way to translate away the feature of ATM using multi-prompt control operators.

3 Simulating ATM with Multi-prompt shift/reset

In this section, we explain the basic ideas of our translation. Kiselyov implemented typed printf in terms of shift and reset without ATM, and we have generalized it to a translation from arbitrary terms in the source language.

Consider a simple example with answer-type modification: $[\![\langle 5 + \mathcal{S}k.k\rangle]\!]$ in which $\mathcal{S}$ is the delimited-control operator shift, and $\langle\cdots\rangle$ is reset. Its answer type changes through computation, as its initial answer type is `int` while its final answer type is `int->int`.

Let us translate the example [1] where $[\![e]\!]$ denotes the result of the translation of the term e.

We begin with the translation of a reset expression:

$$[\![\langle e\rangle]\!] = \mathcal{P}p.\mathcal{P}q.\langle \text{let } y = [\![e]\!]pq \text{ in } \mathcal{S}_q z.\ y\rangle_p$$

where the primitive $\mathcal{P}p$ creates a new prompt and binds the variable p to it. For brevity, the variable p which stores a prompt may also be called a prompt.

The translated term, when it is executed, first creates new prompts p and q and its body e is applied to the arguments p and q. Its result is stored in y and then we execute $\mathcal{S}_q z.y$, but there is no reset with the prompt q around it. Is it an error ? Actually, no. As we will see the definition below, $[\![e]\!]$ is always in the form $\lambda p.\lambda q.e'$ and during the computation of e', $\mathcal{S}_p$ is *always* invoked. Hence e' never returns normally, and the "no-reset" error does not happen. Our invariants in the translation are that the first argument (the prompt p) corresponds to the reset surrounding the expression being translated, and the second argument (the prompt q) corresponds to the above (seemingly dangerous) shift.

From the viewpoint of typing, for each occurrence of answer-type modification from α to β, we use two prompts to simulate the behavior. The prompts p and q generated here correspond to the answer types α and β, respectively.

We translate the term 5 to $[\![5]\!] = \lambda p.\lambda q.\ \mathcal{S}_p k.\ \langle k\ 5\rangle_q$ and the term $\langle 5\rangle$ is translated (essentially) to:

$$\mathcal{P}p.\mathcal{P}q.\langle \text{let } y = \mathcal{S}_p k.\langle k\ 5\rangle_q \text{ in } \mathcal{S}_q z.\ y\rangle_p$$

When we execute the result, $\mathcal{S}_p$ captures its surrounding evaluation context $\langle \text{let } y = [\ ] \text{ in } \mathcal{S}_q z.\ y\rangle_p$, binds k to its functional form $\lambda x.\langle \text{let } y = x \text{ in } \mathcal{S}_q z.\ y\rangle_p$, and continues the evaluation of $\langle k\ 5\rangle_q$. Then we get:

$$\langle\langle\langle \text{let } y = 5 \text{ in } \mathcal{S}_q z.\ y\rangle_p\rangle_q\rangle_p$$

and when this $\mathcal{S}_q$ is invoked, it is surrounded by a reset with the prompt q, and thus it is *safe*. The final result of this computation is 5. In this case, since the execution of the term 5 does not modify the answer type, the prompts p and q passed to the term $[\![5]\!]$ correspond to the same answer type, but we will soon see an example in which they correspond to different answer types.

A shift-expression is translated to:

$$[\![\mathcal{S}k.e]\!] = \lambda p.\lambda q.\mathcal{S}_p k'.\text{let } k = (\lambda y.\langle(\lambda\_.\Omega)(k'y)\rangle_q) \text{ in } (\!|e|\!)$$

As we have explained, p is the prompt for the reset surrounding this expression, hence $\mathcal{S}_p$ in the translated term will capture a delimited continuation up to the

[1] The precise definition of the translation is given later.

reset (which, in turn, corresponds to the nearest reset in the source term). However the delimited continuation contains a dangerous shift at its top position, so we must somehow detoxify it. For this purpose, we replace the captured continuation k' by a function $\lambda y.\langle(\lambda\_.\Omega)(k'y)\rangle_q$ in which the calls to k' is enclosed by a reset with the prompt q, and the dangerous shift in k' will be surrounded by it, sanitizing the dangerous behavior.

Let us consider the types of captured continuations in this translation. Suppose the term $\mathcal{S}k.e$ modifies the answer type from α to β. We use the prompts p and q, whose answer types[2] are β and α, respectively. In the source term, the continuation captured by shift (and then bound to k) has the type $\tau \to \alpha$. In the translated term, the continuation bound to k' has the type $\tau \to \beta$, since the continuation was captured by a shift with the prompt p. After some calculation, it can be inferred that the term $\lambda y.\langle(\lambda\_.\Omega)(k'y)\rangle_q$ has the type $\tau \to \alpha$, hence we can substitute it for k. [3]

We show the mechanism for detoxifying a dangerous shift by executing $[\![\langle 5 + \mathcal{S}k.k\rangle]\!]$, which is equivalent to:

$$\mathcal{P}p.\mathcal{P}q.\langle \text{let } y = \mathcal{P}r.((\mathcal{S}_r k.\langle k\, 5\rangle_q) + (\mathcal{S}_p k'.\text{let } k = \lambda u.\langle(\lambda w.\Omega)(k'\, u)\rangle_r \text{ in } k)) \text{ in } \mathcal{S}_q z.y\rangle_p$$

where the subterm starting with $\mathcal{S}_r$ is the translation result of 5, and the one with $\mathcal{S}_p$ is that of $\mathcal{S}k.k$. In general, each subterm may modify answer types. Hence, a term $e_1 + e_2$ needs three prompts corresponding to the initial, final, and intermediate answer types. The prompt r generated here corresponds to the intermediate answer type.

Evaluating this term in call-by-value, and right-to-left order (after generating all the prompts) leads to the term: $\langle \text{let } k = \lambda u.\langle(\lambda w.\Omega)(k'\, u)\rangle_r \text{ in } k\rangle_p$ where k' is the delimited continuation $\lambda x.\langle \text{let } y = (\mathcal{S}_r k.\langle k\, 5\rangle_q) + x \text{ in } \mathcal{S}_q z.y\rangle_p$. The result of this computation is $\lambda u.\langle(\lambda w.\Omega)(k'\, u)\rangle_r$, which is essentially equivalent to $\lambda y.\langle 5 + y\rangle$. To see this, applying it to 9 yields:

$$\begin{aligned} &(\lambda u.\langle(\lambda w.\Omega)((\lambda x.\langle \text{let } y = (\mathcal{S}_r k.\langle k\, 5\rangle_q) + x \text{ in } \mathcal{S}_q z.y\rangle_p)u)\rangle_r)\, 9 \\ \rightsquigarrow^* &\langle(\lambda w.\Omega)\langle \text{let } y = (\mathcal{S}_r k.\langle k\, 5\rangle_q) + 9 \text{ in } \mathcal{S}_q z.y\rangle_p\rangle_r \end{aligned}$$

$\mathcal{S}_r k.\langle k\, 5\rangle_q$ captures the context with the dangerous shift

$$\begin{aligned} &\rightsquigarrow^* \langle\langle(\lambda u.\langle(\lambda w.\Omega)\langle \text{let } y = u + 9 \text{ in } \mathcal{S}_q z.y\rangle_p\rangle_r)5\rangle_q\rangle_r \\ &\rightsquigarrow^* \langle\langle\langle(\lambda w.\Omega)\langle \text{let } y = 5 + 9 \text{ in } \mathcal{S}_q z.y\rangle_p\rangle_r\rangle_q\rangle_r \\ &\rightsquigarrow^* \langle\langle 14\rangle_q\rangle_r \quad \text{which reduces to 14.} \end{aligned}$$

Thus, our translation uses two prompts to make connections to two answer types, where prompts are generated dynamically.

[2] We assume that, our target language after the translation has multi-prompt shift and reset, but no answer-type modification. Hence, each prompt has a unique answer type.

[3] Here Ω is a term which has an arbitrary type. Such a term can be expressed, as, for instance, $\mathcal{P}p.\mathcal{S}_p k.\lambda x.x$. Its operational behavior does not matter, as it will be never executed.

$$
\begin{array}{rrcl}
\text{(evaluation contexts)} & E & ::= & [\,] \mid eE \mid Ev \mid \langle E \rangle \\
\text{(pure evaluation contexts)} & F & ::= & [\,] \mid eF \mid Fv
\end{array}
$$

$$
\begin{array}{rll}
E[(\lambda x.e)\, v] & \rightsquigarrow E[e\{v/x\}] & \\
E[\text{let } x = v \text{ in } e] & \rightsquigarrow E[e\{v/x\}] & \\
E[\langle v \rangle] & \rightsquigarrow E[v] & \\
E[\langle F[\mathcal{S}k.e] \rangle] & \rightsquigarrow E[\langle e\{\lambda y.\langle F[y] \rangle / k\} \rangle] & y \text{ is a fresh variable in } F
\end{array}
$$

Figure 1: Operational Semantics of Source Calculus

4 Source and Target Calculi

In this section, we formally define our source and target calculi.

The source calculus is based on Asai and Kameyama's polymorphic extension of Danvy and Filinski's calculus for shift and reset, both of which allow answer-type modification [6, 1]. We slightly modified it here; (1) we removed fixpoint and conditionals (but they can be added easily), (2) we use value restriction for let-polymorphism while they used more relaxed condition, and (3) we use Biernacki et al.'s simplification for the types of delimited continuations [4].

The syntax of values and terms of our source calculus λ^{ATM} is defined as follows:

$$
\begin{array}{rrcl}
\text{(values)} & v & ::= & x \mid c \mid \lambda x.e \\
\text{(terms)} & e & ::= & v \mid e_1 e_2 \mid \text{let } x = v \text{ in } e \mid \mathcal{S}k.e \mid \langle e \rangle
\end{array}
$$

where $\lambda x.e$ and $\mathcal{S}k.e$ bind x and k in e, resp.

Figure 1 defines call-by-value operational semantics to the language above.

The term $[\,]$ denotes the empty context. Evaluation contexts are standard, and pure evaluation contexts are those evaluation contexts that have no resets enclosing the hole. Note that we use the right-to-left evaluation order for the function applications to reflect the current OCaml compiler's semantics.

The first two evaluation rules are the standard beta and let rules, where $e\{v/x\}$ denotes capture-avoiding substitution. The next two rules are those for control operators: if the body of a reset expression is a value, the occurrence of reset is discarded. If the next redex is a shift expression, we capture the continuation up to the nearest reset and bind k to it.

Figure 2 introduces types and related notions. Types are type variables (t), base types (b), pure function types ($\sigma \to \tau$), or effectful function types ($\sigma/\alpha \to \tau/\beta$), which represent function types $\sigma \to \tau$ where the answer type changes from α to β.

Figure 3 defines the type system of λ^{ATM}. Type judgments are either $\Gamma \vdash_p e : \tau$ (pure judgments) or $\Gamma \vdash e : \tau; \alpha, \beta$ (effectful judgments), the latter of which means that evaluating e with the answer type α yields a value of type τ with the answer type being modified to β. The typing rules are based on Danvy

$$\tau, \sigma, \alpha, \beta ::= t \mid b \mid \sigma \to \tau \mid (\sigma/\alpha \to \tau/\beta)$$
$$A ::= \tau \mid \forall t.A$$
$$\Gamma ::= \emptyset \mid \Gamma, x : A$$

Figure 2: Types, Type Schemes and Type Environments

$$\frac{x : A \in \Gamma, \tau < A}{\Gamma \vdash_p \ x : \tau}\ \text{var} \qquad \frac{\Gamma \vdash e : \sigma; \sigma, \tau}{\Gamma \vdash_p \langle e \rangle : \tau}\ \text{reset}$$

$$\frac{\Gamma, x : \tau \to \alpha \vdash_p e : \beta}{\Gamma \vdash \mathcal{S}x.e : \tau; \alpha, \beta}\ \text{shift} \qquad \frac{(c \text{ is a constant of type } b)}{\Gamma \vdash_p c : b}\ \text{const}$$

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

$$\frac{\Gamma \vdash e_1 : \sigma/\alpha \to \tau/\beta; \beta, \gamma \qquad \Gamma \vdash e_2 : \sigma; \gamma, \delta}{\Gamma \vdash e_1 e_2 : \tau; \alpha, \delta}\ \text{app}$$

$$\frac{\Gamma \vdash_p e_1 : \sigma \to \tau \qquad \Gamma \vdash_p e_2 : \sigma}{\Gamma \vdash_p e_1 e_2 : \tau}\ \text{app-p}$$

$$\frac{\Gamma \vdash_p v : \sigma \qquad \Gamma, x : \mathrm{Gen}\,(\sigma; \Gamma) \vdash e : \tau; \alpha, \beta}{\Gamma \vdash \text{let } x = v \text{ in } e : \tau; \alpha, \beta}\ \text{let}$$

Figure 3: Typing Rules of the Source Calculus

and Filinski's [6] except that we have let-polymorphism and clear distinction of pure judgments from impure judgments following Asai and Kameyama [1].

In the var rule, $\tau < A$ means that the type τ is an instance of type scheme A, and the type $\mathrm{Gen}\,(\sigma; \Gamma)$ denotes $\forall t_1, \cdots \forall t_n.\sigma$ where $t_1, \cdots, t_n$ are the type variables that appear in σ but not appear in Γ freely. The delimited continuations captured by shift expressions are pure functions (they are polymorphic in answer types), and we use the pure function space $\tau \to \alpha$ for this purpose. On the contrary, the functions introduced by lambda are, in general, effectful. Accordingly, we have two rules for applications. Note that the body of a shift expression is restricted to a pure expression, in order to simplify the definition of our translation. This choice does not change the expressive power of the language, since we can always insert a reset into the topmost position of the body of a shift expression, to turn the body to a pure expression, without affecting typability and operational behavior. The exp rule turns pure terms into effectful terms.

The type system of the source calculus λ^{ATM} enjoys the subject reduction property. The proof is standard and omitted.

We then define the target calculus λ^{mpsr}, which is a polymorphic calculus with multi-prompt shift and reset (but without ATM). The calculus is similar, in spirit, to Gunter et al.'s calculus with the `cupto` and `set` operators [9]. Besides disallowing ATM, the target calculus differs from the source calculus in that the control operators are named, to allow mixing multiple effects in a single program. The names for control operators are called *prompts* for historical reasons, and denoted by $p, q, \cdots$. In our formulation, prompts are first-class values and can be bound to ordinary variables x. Prompts are given as prompt-constants, or can be generated dynamically by the $\mathcal{P}$ primitive. For instance, evaluating $\mathcal{P}x.\langle 1 + \mathcal{S}_x k.e\rangle_x$ first creates a fresh prompt p and substitutes it for x, then evaluate $\langle 1 + \mathcal{S}_p k.e\rangle_p$. This choice of the formulation closely follows Kiselyov's DelimCC library for multi-prompt shift/reset.

Types and typing environments are defined as follows:

$$\begin{aligned} \tau, \sigma &::= t \mid b \mid \sigma \to \tau \mid \tau \text{ pr} \\ A &::= \tau \mid \forall t.A \\ \Gamma &::= \emptyset \mid \Gamma, x : A \end{aligned}$$

where τ pr is the type for the prompts with the answer type τ. The syntax of values and terms are defines as follows:

$$\begin{aligned} v &::= x \mid c \mid \lambda x.e \mid p \\ e &::= v \mid e_1 e_2 \mid \mathcal{S}_v x.e \mid \langle e\rangle_v \mid \mathcal{P}x.e \mid \text{let } x = v \text{ in } e \mid \Omega \end{aligned}$$

where p is a prompt-constant. The control operators now receive not only prompt-constants, but values which will reduce to prompts. The term $\mathcal{P}x.e$ creates a fresh prompt and binds x to it. The term Ω denotes a non-terminating computation of arbitrary types. It may be defined in terms of shift, but for the sake of clarity, we added it as a primitive.

Evaluation contexts and evaluation rules are given as follows:

$$\begin{aligned} E &::= [\,] \mid Ee \mid vE \mid \langle E\rangle_p \\ E[(\lambda x.e)v] &\rightsquigarrow E[e\{v/x\}] \\ E[\text{let } x = v \text{ in } e] &\rightsquigarrow E[e\{v/x\}] \\ E[\mathcal{P}x.e] &\rightsquigarrow E[e\{p/x\}] \qquad p \text{ is a fresh prompt-constant} \\ E[\langle v\rangle_p] &\rightsquigarrow E[v] \\ E[\langle E_p[\mathcal{S}_p x.e]\rangle_p] &\rightsquigarrow E[\langle e\{\lambda y.\langle E_p[y]\rangle_p/x\}\rangle_p] \\ E[\Omega] &\rightsquigarrow E[\Omega] \end{aligned}$$

Note that we use E_p in the second last rule, which is an evaluation context that does not have a reset with the prompt p around the hole, and thus implies that we capture the continuation up to the nearest reset with the prompt p.

Finally we give typing rules for the target calculus in Figure 4. The type system of the target calculus is mostly standard except the use of prompts. In the shift rule, the prompt expression v must be of type σ pr where σ is the

$$\frac{x : A \in \Gamma, \tau < A}{\Gamma \vdash x : \tau}\ \text{var} \qquad \frac{(c \text{ is a constant of } b)}{\Gamma \vdash c : b}\ \text{const}$$

$$\frac{\Gamma \vdash v : \tau\ \text{pr} \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \langle e \rangle_v : \tau}\ \text{reset} \qquad \frac{\Gamma \vdash v : \sigma\ \text{pr} \qquad \Gamma, x : \tau \to \sigma \vdash e : \sigma}{\Gamma \vdash \mathcal{S}_v\ x.e : \tau}\ \text{shift}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \sigma \to \tau}\ \text{fun} \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau}\ \text{app}$$

$$\frac{\Gamma \vdash v : \sigma \qquad \Gamma, x : \text{Gen}\,(\sigma; \Gamma) \vdash e : \tau}{\Gamma \vdash \text{let } x = v \text{ in } e : \tau}\ \text{let} \qquad \frac{\Gamma, x : \sigma\ \text{pr} \vdash e : \tau}{\Gamma \vdash \mathcal{P}x.e : \tau}\ \text{prompt}$$

$$\frac{}{\Gamma \vdash \Omega : \tau}\ \text{omega}$$

Figure 4: Typing Rules of the Target Calculus

$$\begin{aligned}
[\![\tau; \alpha, \beta]\!] &= [\![\beta]\!]\ \text{pr} \to [\![\alpha]\!]\ \text{pr} \to [\![\tau]\!] \\
[\![b]\!] &= b \\
[\![t]\!] &= t \\
[\![\sigma \to \tau]\!] &= [\![\sigma]\!] \to [\![\tau]\!] \\
[\![\sigma/\alpha \to \tau/\beta]\!] &= [\![\sigma]\!] \to [\![\tau; \alpha, \beta]\!] \\
[\![\tau]\!] &= [\![\tau]\!] \\
[\![\forall t.A]\!] &= \forall t.[\![A]\!] \\
[\![\Gamma, x : A]\!] &= [\![\Gamma]\!], x : [\![A]\!]
\end{aligned}$$

Figure 5: Translation for Types, Type Schemes and Type Environments

type of the body of the shift expression. A similar restriction is applied to the reset rule. In the prompt rule, we can create an arbitrary prompt and binds a variable x to it.

The type system enjoys the subject reduction property modulo the set of dynamically created prompts which have infinite extents.

5 Translation

In this section, we give the syntax-directed translation from λ^{ATM} to λ^{mpsr}, which translates away the feature of answer-type modification. The translation borrows the idea of Kiselyov's implementation of typed printf in terms of multi-prompt shift and reset, but this paper gives a translation for the whole source calculus and also a proof of the type preservation property. Later, we will show a tagless-final implementation of our translation which is another evidence that our translation actually works type-safely.

Figure 6 presents the translation rules from λ^{ATM} to λ^{mpsr}. As we have explained in earlier sections, we emulate ATM from the type α to the type β in

$$
\begin{aligned}
(\!|x|\!) &= x \\
(\!|c|\!) &= c \\
(\!|\lambda x.e|\!) &= \lambda x.[\![e]\!] \\
(\!|e_1 e_2|\!) &= (\!|e_1|\!)(\!|e_2|\!) \\
(\!|\langle e \rangle|\!) &= \mathcal{P}pq.\langle(\lambda y.\mathcal{S}_{q\_}.y)([\![e]\!]pq)\rangle_p \\
[\![e_1 e_2]\!] &= \lambda pq.\mathcal{P}rs.([\![e_1]\!]rs)([\![e_2]\!]pr)sq \\
[\![\text{let } x = v \text{ in } e_2]\!] &= \lambda pq.\text{let } x = (\!|v|\!) \text{ in } [\![e_2]\!]pq \\
[\![\mathcal{S}k.e]\!] &= \lambda pq.\mathcal{S}_p k'.((\lambda k.(\!|e|\!))(\lambda y.\langle(\lambda\_.\Omega)(k'y)\rangle_q) \\
[\![e]\!] &= \lambda pq.\mathcal{S}_p k.\langle k(\!|e|\!)\rangle_q \qquad e \text{ is a pure term}
\end{aligned}
$$

Figure 6: Translation for Terms

terms of two prompts whose answer types are α pr and β pr. Hence the triple $\tau; \alpha, \beta$ in the typing judgment is translated to the type $[\![\beta]\!]$ pr $\to [\![\alpha]\!]$ pr $\to [\![\tau]\!]$.

Figure 5 presents the translation rules for types, type schemes and type environments. They are translated in a natural way except that the type for effectful functions $\sigma/\alpha \to \tau/\beta$, which are translated to standard function types but their codomains are the translation of the triples above.

We can show that our translation preserves typing.

Theorem 1 (Type preservation)**.** *If $\Gamma \vdash e : \tau; \alpha, \beta$ is derivable in the source calculus λ^{ATM}, then $[\![\Gamma]\!] \vdash [\![e]\!] : [\![\tau; \alpha, \beta]\!]$ is derivable in the target calculus λ^{mpsr}.*

Similarly, if $\Gamma \vdash_p e : \tau$ is derivable in λ^{ATM}, so is $[\![\Gamma]\!] \vdash (\!|e|\!) : [\![\tau]\!]$ in λ^{mpsr}.

Proof. We will prove the two statements by simultaneous induction on the derivations. Here we only show a few interesting cases.

(Case $e = \langle e_1 \rangle$) We have a derivation for:

$$\frac{\Gamma \vdash e_1 : \sigma; \sigma, \tau}{\Gamma \vdash_p \langle e_1 \rangle : \tau}$$

By induction hypothesis, we can derive $[\![\Gamma]\!] \vdash [\![e_1]\!] : [\![\sigma; \sigma, \tau]\!]$. Let $\Gamma' = [\![\Gamma]\!], p : [\![\tau]\!]$ pr, $q : [\![\sigma]\!]$ pr and $\Gamma'' = \Gamma', y : [\![\sigma]\!]$. We have the following derivation:

$$\dfrac{\Gamma' \vdash p : [\![\tau]\!] \text{ pr} \qquad \dfrac{\dfrac{\dfrac{\Gamma'' \vdash q : [\![\sigma]\!] \text{ pr} \qquad \Gamma'' \vdash y : [\![\sigma]\!]}{\Gamma'' \vdash \mathcal{S}_{q\_}.y : [\![\tau]\!]}}{\Gamma' \vdash \lambda v.\mathcal{S}_{q\_}.y : [\![\sigma]\!] \to [\![\tau]\!]} \qquad \dfrac{\Gamma' \vdash q : [\![\sigma]\!] \text{ pr} \qquad \dfrac{\Gamma' \vdash p : [\![\tau]\!] \text{ pr} \qquad \Gamma' \vdash [\![e_1]\!] : [\![\sigma; \sigma, \tau]\!]}{\Gamma' \vdash [\![e_1]\!]p : [\![\sigma]\!] \text{ pr} \to [\![\sigma]\!]}}{\Gamma' \vdash [\![e_1]\!]pq : [\![\sigma]\!]}}{\Gamma' \vdash (\lambda y.\mathcal{S}_{q\_}.y)\,([\![e_1]\!]pq) : [\![\tau]\!]}}{\dfrac{\Gamma' \vdash \langle(\lambda y.\mathcal{S}_{q\_}.y)\,([\![e_1]\!]pq)\rangle_p : [\![\tau]\!]}{[\![\Gamma]\!] \vdash \mathcal{P}p.\mathcal{P}q.\langle(\lambda v.\mathcal{S}_{q\_}.v)\,([\![e_1]\!]pq)\rangle_p : [\![\tau]\!]}}$$

which derives $[\![\Gamma]\!] \vdash (\!|\langle e_1 \rangle|\!) : [\![\tau]\!]$.

(Case $e = \mathcal{S}x.e_1$) We have a deviation for

$$\frac{\Gamma, x : \tau \to \alpha \vdash_p e_1 : \beta}{\Gamma \vdash \mathcal{S}x.e_1 : \tau; \alpha, \beta}$$

By induction hypothesis $[\![\Gamma, x : \tau \to \alpha]\!] \vdash (\!|e_1|\!) : [\![\beta]\!]$ is derivable. Let $\Gamma' = [\![\Gamma]\!], p : [\![\beta]\!]$ pr, $q : [\![\alpha]\!]$ pr, $\Gamma'' = \Gamma', k' : [\![\tau]\!] \to [\![\beta]\!]$, and $\Gamma''' = \Gamma'', y : [\![\tau]\!]$, then we have:

$$\dfrac{\Gamma'' \vdash p : [\![\beta]\!] \text{ pr} \qquad \dfrac{\dfrac{\dfrac{\Gamma'', x : [\![\tau]\!] \to [\![\alpha]\!] \vdash (\!|e_1|\!) : [\![\beta]\!]}{\Gamma'' \vdash \lambda x.(\!|e_1|\!) : ([\![\tau]\!] \to [\![\alpha]\!]) \to [\![\beta]\!]} \qquad \dfrac{\Gamma''' \vdash \langle (\lambda\_.\Omega)\,(k'y)\rangle_q : [\![\alpha]\!]}{\Gamma'' \vdash \lambda y.\langle (\lambda\_.\Omega)\,(k'y)\rangle_q : [\![\tau]\!] \to [\![\alpha]\!]}}{\Gamma'' \vdash (\lambda x.(\!|e_1|\!))\,(\lambda y.\langle (\lambda\_.\Omega)\,(k'y)\rangle_q) : [\![\beta]\!]}}{\Gamma' \vdash \mathcal{S}_p k'.\,(\lambda x.(\!|e_1|\!))\,(\lambda y.\langle (\lambda\_.\Omega)\,(k'y)\rangle_q) : [\![\tau]\!]}}{[\![\Gamma]\!] \vdash \lambda p.\lambda q.\mathcal{S}_p k'.\,(\lambda x.(\!|e_1|\!))\,(\lambda y.\langle (\lambda\_.\Omega)\,(k'y)\rangle_q) : [\![\beta]\!] \text{ pr} \to [\![\alpha]\!] \text{ pr} \to [\![\tau]\!]}$$

which derives $[\![\Gamma]\!] \vdash [\![\mathcal{S}x.e_1]\!] : [\![\tau; \alpha, \beta]\!]$. (end of proof)

Hence our translation preserves typing. We think that our translation also preserve operational semantics but its formalization is left for future work.

6 Tagless-final embedding in OCaml

We have implemented interpreters for the calculus in Figure 3 via the translation presented in the previous section. To simplify the implementation, we have eliminated let polymorphism from the source calculus.

Our implementation is based on Carette et al.'s tagless-final approach [5], which lets one embed a domain-specific language in a metalanguage while preserving typing of the embedded language. The syntax as well as the typing rules of the embedded language are represented by a signature, and its semantics is given as an interpretation of this signature. In this approach, type checking/inference of the embedded language is reduced to that of the metalanguage, and all representable terms and interpretations are type safe, by construction.

In this work, we choose OCaml plus multi-prompt shift and reset (implemented in the DelimCC library) as the metalanguage. We embed our source calculus λ^{ATM} in this metalanguage, and give its semantics using our translation.

Figure 7 shows the module signature (called Symantics) for our source calculus λ^{ATM}, embedded in OCaml. Each function in the signature represents a term-constructor of λ^{ATM}, and its type encodes the corresponding typing rule. The type `'t pure` means that the expression is pure and has the type `'t`. The type `('t, 'a, 'b) eff` means that the expression has the type `'t` and has an effect that modifies the answer type from `'a` to `'b`.

Note that, we can represent all and only typable terms in λ^{ATM} as a combination of the functions defined in the signature, and the typability of embedded terms are checked by the type system of OCaml; in other words, all representable terms are typable *when they are constructed*.

In the tagless-final approach, operational semantics of the embedded language is given as an interpretation of the Symantics signature, namely, a module

```
module type Symantics = sig
  type 't pure
  type ('t, 'a, 'b) eff
  type ('s, 't, 'a, 'b) efun
  type ('s, 't) pfun

  val const : 't -> 't pure
  val lam : ('s pure -> ('t, 'a, 'b) eff)
       -> ('s, 't, 'a, 'b) efun pure
  val app : (('s, 't, 'a, 'b) efun, 'b, 'c) eff
       -> ('s, 'c, 'd) eff -> ('t, 'a, 'd) eff
  val appP : ('s, 't) pfun pure -> 's pure -> 't pure
  val shift : (('t, 'a) pfun pure -> 'b pure)
       -> ('t, 'a, 'b) eff
  val reset : ('s, 's, 't) eff -> 't pure
  val exp : 't pure -> ('t, 'a, 'a) eff
  val run : 't pure -> 't
end
```

Figure 7: Signature of the Embedded Language

of type Symantics. An interpreter for our source language is shown in Figure 10 in the appendix. The interpreter interprets an embedded term in the meta-language, using the translation introduced in this paper.

To write concrete examples, we extend the source language with primitive functions, conditionals, and the fixpoint operator. Its signature SymP is shown in Figure 9 in the appendix of this paper.

Using these modules, we can write programming examples such as list append and prefix, shown in Figure 8 where we use syntax sugars for list-manipulating functions defined in Figure 12. Running these examples gives correct answers, as shown in the comments of the code in Figure 8.

In summary, we have successfully implemented our source language and the translation into the target language using the tagless-final approach. We think that this implementation provides another good evidence that our translation is type preserving. Thanks to the tagless-final approach, our implementation is extensible, and in fact, we added several primitive such as the fixpoint operator in a type-safe way.

7 Related Work and Conclusion

In this paper, we have proposed type-preserving translation for embedding programs with ATM into those without. Our translation uses multi-prompt systems and dynamic creation of prompts to emulate two answer types in effectful terms. We proved type preservation and implemented in OCaml the translation using the tagless-final style, which would add the "credibility" of our work.

```
module Example (S: SymP) = struct
 open LIST  (* LIST module defines syntax sugars for lists.
    *)

 let append = fixE (fun f x ->
     ifE (null x)
       (shift (fun k -> k))
       (head (exp x) @* app (exp f) (tail @@ exp x)))
 let res1 = run @@
   appP (reset (app (exp append) (exp @@ list [1;2;3])))
        (list [4;5;6])
 (* res1 returns [1; 2; 3; 4; 5; 6] *)

 let prefix =
   fixE (fun f x ->
     ifE (null x)
       (shift (fun k -> list []))
       (head (exp x) @* shift (fun k -> reset @@
              (exp (appP k (list [])))
               @* (exp (reset @@ app (exp2 k)
                         (app (exp f) (tail @@ exp x)))))))

 let res2 = run @@
   reset @@ app (exp prefix) (exp @@ list [1;2;3])
 (* res2 returns [[1]; [1; 2]; [1; 2; 3]] *)
end
```

Figure 8: Programming Examples

Let us briefly summarize related work. Rompf et al. [12] implemented shift and reset in Scala, that allow answer-type modification. Their source language needs relatively heavy type annotations to be implemented by a selective CPS transformation, and does not allow higher-order functions. Masuko and Asai [11] designed a language OchaCaml, which is Caml light extended with shift and reset. OchaCaml fully supports ATM at the cost of redesigning the whole type system and an extension of the run-time system. Wadler [13] studied monad-like structures to express shift and reset with Danvy and Filinski's type system [6]. Inspired by his work, Atkey [2, 3] proposed *parameterised monads*, which generalize monads. Parametrised monads take two more type parameters to express inputs and outputs, or initial states and final states. Unlike standard monads, parameterised monads can express answer-type modification. He studied categorical foundation of parameterised monads. Kiselyov [10] independently studied the same notion as parameterised monads, and gave an implementation and a number of programming examples using it.

For future work, we plan to formally prove the semantics-preservation property mentioned in this paper. Investigating other delimited-control operators

such as shift0/reset0 and control/prompt with answer-type modification would be also interesting.

Acknowledgments: We are grateful to the anonymous reviewers for their constructive comments. The first and second authors are supported in part by JSPS Grant-in-Aid for Scientific Research B (No. 25280020).

References

[1] Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In *Proc. Fifth Asian Symposium on Programming Languages and Systems*, volume 4807 of *Lecture Notes in Computer Science*, pages 239–254. Springer Berlin Heidelberg, 2007. 2, 5, 6

[2] Robert Atkey. Parameterised notions of computation. In *Proceedings of the Workshop on Mathematically Structured Functional Programming*, Kuressare, Estonia, 2006. eWiC. 12

[3] Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3-4):335, June 2009. 12

[4] Malgorzata Biernacka and Dariusz Biernacki. Context-based proofs of termination for typed delimited-control operators. *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, page 289, 2009. 5

[5] Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509, April 2009. 10

[6] Olivier Danvy and Andrzej Filinski. A Functional Abstraction of Typed Contexts. Technical report, Computer Science Department, University of Copenhagen, 1989. 5, 6, 12

[7] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160, New York, USA, 1990. ACM Press. 1

[8] Matthias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 180–190, New York, USA, 1988. ACM Press. 1

[9] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 12–23, New York, USA, 1995. ACM Press. 7

[10] Oleg Kiselyov. Parametrized 'monad'. `http://okmij.org/ftp/Computation/monads.html#param-monad`, 2006. 12

[11] Moe Masuko and Kenichi Asai. Caml Light + shift/reset = Caml Shift. In *First International Workshop on the Theory and Practice of Delimited Continuations*, pages 33–46, 2011. 1, 12

[12] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, page 317, New York, USA, 2009. ACM Press. 1, 12

[13] Philip Wadler. Monads and composable continuations. *LISP and Symbolic Computation*, 7(1):39–55, January 1994. 12

A Implementation of Interpreters

We show our tagless-final implementation of the source language and its interpretation, which is based on our translation in the paper.

Figure 9 shows the signature for the extended language.

```
module type SymP = sig
 include Symantics

 val p2E : ('t1 -> 't2 -> 't3) -> ('t1, 'a, 'b) eff
     -> ('t2, 'b, 'c) eff -> ('t3, 'a, 'c) eff
 val pE: ('t -> 'u) -> ('t, 'a, 'a) eff
     -> ('u, 'a, 'a) eff
 val ifE : (bool, 'b, 'c) eff -> ('t, 'a, 'b) eff
     -> ('t, 'a, 'b) eff -> ('t, 'a, 'c) eff
 val fixE : ((('s, 't, 'a, 'b) efun) pure -> 's pure
     -> ('t, 'a, 'b) eff) -> ('s, 't, 'a, 'b) efun pure
end
```

Figure 9: Signature of the Extended Language

Figure 10 is an interpreter for the embedded language, which is a Symantics module. As can be seen, our implementation reflects our translation to the target language, which is OCaml plus DelimCC.

Figure 11 shows an interpreter for the extended language, which is a SymP module.

We use syntax sugar for list-manipulating functions, which are defined in the LIST module in Figure 12.

Here pE and p2E are primitives for lifting up one or two arguments primitives, and fixE, ifE and exp2 are the fixpoint operator, conditional, and a primitive to turn pure functions to effectful one, respectively.

```
module R = struct
  type 't pure = unit -> 't
  type ('t, 'a, 'b) eff = 'b prompt -> 'a prompt -> 't
  type ('s, 't, 'a, 'b) efun = 's pure -> ('t, 'a, 'b)
   eff
  type ('s, 't) pfun = 's pure -> 't pure

  let coerce _ = failwith "unreachable"

  let wrap x : 'a pure = fun () -> x
  let unwrap : 'a pure -> 'a = fun x -> x ()

  let const x = wrap x
  let lam f = wrap f

  let app e1 (e2: ('s, 'c, 'd ) eff) : ('t, 'a, 'd) eff =
    fun p q ->
    let r, s = new_prompt (), new_prompt () in
    let v2 = wrap @@ e2 p r in
    let v1 = e1 r s in
    v1 v2 s q

  let appP e1 e2 : 't pure =
    fun () ->
    let v2 = unwrap e2 in
    let v1 = unwrap e1 in
    v1 (wrap v2) ()

  let shift f : ('t, 'a, 'b) eff =
    fun p q ->
    Delimcc.shift p (fun k' ->
      unwrap @@ f @@ wrap (fun y ->
        wrap @@ push_prompt q
                  (fun () -> coerce (k' @@ unwrap y ))))

  let reset e =
    fun () ->
      let p, q = new_prompt (), new_prompt () in
      push_prompt p (fun () -> abort q @@ e p q)

  let exp e =
    fun p q -> Delimcc.shift p
      (fun k -> push_prompt q (fun () -> k @@ unwrap e))

   let run (x: 't pure) : 't = x ()
 end
```

Figure 10: Interpretation of the Embedded Language

```
module RP = struct
 include R

 let p2E f e1 e2 = fun p q ->
   let r = new_prompt () in
   let v2 = e2 p r in
   let v1 = e1 r q in
   f v1 v2

 let pE f x = fun p q -> f (x p q)

 let ifE b e e' = fun p q ->
   let r = new_prompt () in
   if b p r then e r q else e' r q

 let rec fixE f : ('s, 't, 'a, 'b) efun pure =
   lam (fun x -> f (fun x -> fixE f x) x)
end
```

Figure 11: Interpretation of the Extended Language

```
module LIST (S: SymP) = struct
 open S

 let list x = const x
 let (@* ) x y = p2E (fun x y -> x :: y) x y
 let head x = pE List.hd x
 let tail x = pE List.tl x
 let null x = pE (fun x -> x = []) @@ exp x
 let exp2 f = exp @@ lam (fun x -> exp @@ appP f x)
end
```

Figure 12: Syntax sugar for Lists