Answer-Type Modification without Tears: Prompt-Passing Style Translation for Typed Delimited-Control Operators

Ikuo Kobori
University of Tsukuba, Japan
ikuo@logic.cs.tsukuba.ac.jp

Yukiyoshi Kameyama
University of Tsukuba, Japan
kameyama@acm.org

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

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 ATM in standard languages without modifying the type system. Our translation generalizes Kiselyov's direct-style implementation of typed printf, which uses two prompts to emulate the modification of answer types, and passes them during computation. We prove that our translation preserves typing. As the naive prompt-passing style translation generates and passes many prompts even for pure terms, we show an optimized translation that generate prompts only when needed, which is also type-preserving. Finally, we give an implementation in the tagless-final style which respects typing by construction.

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 [10, 8], 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.

Yet, 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 easily 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 [16] and OchaCaml [15], we cannot directly express the beautiful examples with ATM as programs in standard languages.

We address this problem, and present a solution for it. Namely, we give a translation from a calculus with ATM shift and reset into a calculus with multi-prompt shift and reset without ATM. Our translation is a generalization of Kiselyov's implementation of typed printf using multi-prompt shift and reset where he associates each answer type with a prompt (a tag) for delimited-control operators. Our translation also uses prompts to simulate answer types, and the term obtained by our translation dynamically generates

Submitted to:
WoC 2015

and passes two prompts during computation, thus we call it Prompt-Passing Style (PPS), after the well known Continuation-Passing Style.

We introduce a PPS translation from a calculus with ATM to a calculus without, and prove that it is type-preserving. We also give an implementation based on our translation in the tagless-final style, which allows us to embed a domain-specific language while preserving types by construction.

The PPS translation differs from the definitional CPS translation for shift and reset [8] in that the generated term by the our PPS translation are in direct style, while the generated terms by the CPS translation are in continuation-passing style, which makes the size of terms bigger, and may affect performance. In order to show this aspect better, we refine the naive PPS translation to obtain an optimized PPS translation, where prompts are generated and passed only when needed. It is based on the idea of one-pass CPS translation as well as an ad hoc optimization for prompts. The optimized translation is also type preserving and has been implemented in the tagless-final style. We show by examples that the optimized PPS translation generates much smaller terms than the naive PPS translation and the CPS translation.

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, and Section 6 introduces an optimized translation with examples. Based on these theoretical development, Section 7 gives a tagless-final implementation of shift and reset with answer-type modification as well as several programming examples. Section 8 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 [2].

```
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: $[\![\langle 5 + \mathscr{S}k.k\rangle]\!]$ in which $\mathscr{S}$ 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 where $[\![e]\!]$ denotes the result of the translation of the term e. (The precise definition of the translation is given later.)

We begin with the translation of a reset expression:

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

where the primitive $\mathscr{P}p$ 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 $\mathscr{S}_q z.y$, 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', $\mathscr{S}_p$ 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 $[\![5]\!] = \lambda p.\lambda q.\ \mathscr{S}_p k.\ \langle k\ 5\rangle_q$ and the term $\langle 5\rangle$ is translated (essentially) to:

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

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

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

and when this $\mathscr{S}_q$ 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:

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

As we have explained, p is the prompt for the reset surrounding this expression, hence $\mathscr{S}_p$ in the translated term will capture a delimited continuation up to the 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 $\mathscr{S}k.e$ modifies the answer type from α to β. We use the prompts p and q, whose answer types[1] 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. [2]

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

$$\mathscr{P}p.\mathscr{P}q.\langle\text{let } y = \mathscr{P}r.((\mathscr{S}_r k.\langle k5\rangle_q) + (\mathscr{S}_p k'.\text{let } k = \lambda u.\langle(\lambda w.\Omega)(k'u)\rangle_r \text{ in } k)) \text{ in } \mathscr{S}_q z.y\rangle_p$$

where the subterm starting with $\mathscr{S}_r$ is the translation result of 5, and the one with $\mathscr{S}_p$ is that of $\mathscr{S}k.k$. 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 $\lambda x.\langle\text{let } y = (\mathscr{S}_r k.\langle k5\rangle_q) + x \text{ in } \mathscr{S}_q z.y\rangle_p$. 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:

$$\begin{aligned} &(\lambda u.\langle(\lambda w.\Omega)((\lambda x.\langle\text{let } y = (\mathscr{S}_r k.\langle k5\rangle_q) + x \text{ in } \mathscr{S}_q z.y\rangle_p)u)\rangle_r)\,9 \\ \rightsquigarrow^* &\langle(\lambda w.\Omega)\langle\text{let } y = (\mathscr{S}_r k.\langle k5\rangle_q) + 9 \text{ in } \mathscr{S}_q z.y\rangle_p\rangle_r \end{aligned}$$

$\mathscr{S}_r k.\langle k5\rangle_q$ captures the context with the dangerous shift

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

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

[1] 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.

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

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

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

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 [7, 2]. 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 [5].

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

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

where x is an ordinary variable, k is a continuation variable, and $\text{throw}(k,e)$ is application for continuations, which is syntactically different from ordinary application $e_1\,e_2$. This distinction is technical and inessential for expressivity, as we can always convert a continuation k to a value $\lambda x.\text{throw}(k,x)$. The variables x and k, resp., are bound in the terms $\lambda x.e$ and $\mathcal{S}k.e$, resp.,

Figure 1 defines call-by-value operational semantics to the language above, where $[\,]$ denotes the empty context and $E[e]$ denotes the usual hole-filling operation. Evaluation contexts are standard, and pure evaluation contexts are those evaluation contexts that have no resets enclosing the hole. We use the right-to-left evaluation order for the function application to reflect the semantics of the OCaml compiler.

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 ($\lambda y.\langle F[y]\rangle$), and substitute it for k in the body. $\{k := \lambda y.\langle F[y]\rangle\}$ denotes capture-avoiding substitution for continuation variables where we define $\text{throw}(k,e)\{k := \lambda y.\langle F[y]\rangle\}$ as $(\lambda y.\langle F[y]\rangle)\,(e\{k := \lambda y.\langle F[y]\rangle\})$, namely $\text{throw}(k,e)$ is the same as $k\,e$ in the original formulation [8].

Types, type schemes and type environments are defined as follows:

$$\begin{aligned} \tau,\sigma,\alpha,\beta &::= t \mid b \mid (\sigma/\alpha \to \tau/\beta) \\ A &::= \tau \mid \forall t.A \\ \Gamma &::= \emptyset \mid \Gamma, x : A \mid \Gamma, k : \sigma \to \tau \end{aligned}$$

Types are either type variables (t), base types (b), 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. Type scheme A represents polymorphic types as usual. Type environment Γ is a finite sequence of variable-type pairs, which possibly contain continuation variables k, that has a pure function type $\sigma \to \tau$.

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

$$\frac{\Gamma, x : \sigma \vdash e : \tau;\ \beta,\ \gamma}{\Gamma \vdash_p \lambda x.e : (\sigma/\beta \to \tau/\gamma)}\ \text{fun} \qquad \frac{\Gamma \vdash e_1 : (\sigma/\alpha \to \tau/\beta);\ \beta,\ \gamma \qquad \Gamma \vdash e_2 : \sigma;\ \gamma,\ \delta}{\Gamma \vdash e_1\, e_2 : \tau;\ \alpha,\ \delta}\ \text{app}$$

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

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

Figure 2: Typing Rules of the Source Calculus

Figure 2 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 and Filinski's [7] except that we have let-polymorphism and clear distinction of pure judgments from impure judgments following Asai and Kameyama [2].

In the var rule, $\tau < A$ means that the type τ is an instance of type scheme A, and the type $\text{Gen}\,(\sigma;\Gamma)$ 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 restriction is inessential; in the standard formulation (where the body of shift is an effectful expression), the term $\mathscr{S}x.e$ is typable if and only if $\mathscr{S}x.\langle e \rangle$ is typable, and their operational behaviors are the same. The exp rule turns pure terms into effectful terms where we have chosen an implicit coercion from a pure term to an effectful one.

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

We introduce the target calculus λ^{mpsr}, which is a polymorphic calculus with multi-prompt shift and reset without ATM. The calculus is similar, in spirit, to Gunter et al.'s calculus with the `cupto` and `set` operators [11]. 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 $\mathscr{P}$ primitive. For instance, evaluating $\mathscr{P}x.\langle 1 + \mathscr{S}_x k.e \rangle_x$ first creates a fresh prompt p and substitutes it for x, then evaluate $\langle 1 + \mathscr{S}_p k.e \rangle_p$. This choice of the formulation closely follows Kiselyov's DelimCC library for multi-prompt shift and 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 defined

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

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

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

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

Figure 3: Typing Rules of the Target Calculus

as follows:

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

where p is a prompt-constant. The control operators now receive not only prompt-constants, but values which will reduce to prompts. The term $\mathscr{P} x.e$ 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:

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

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 3. 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 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 The PPS Translation

In this section, we give a Prompt-Passing Style (PPS) translation, the syntax-directed translation from λ^{ATM} to λ^{mpsr}, which translates away the feature of answer-type modification.

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

Figure 4: Translation for Triples, Types, Type Schemes and Type Environments

$$
\begin{aligned}
(\!|x|\!) &= x \\
(\!|c|\!) &= c \\
(\!|\lambda x.e|\!) &= \lambda x.[\![e]\!] \\
(\!|\mathrm{throw}(k,e)|\!) &= k\,(\!|e|\!) \\
(\!|\langle e\rangle|\!) &= \mathscr{P}pq.\langle(\lambda y.\mathscr{S}_q\_.y)([\![e]\!]pq)\rangle_p \\
[\![e_1 e_2]\!] &= \lambda pq.\mathscr{P}rs.([\![e_1]\!]rs)([\![e_2]\!]pr)sq \\
[\![\mathrm{let}\ x = v\ \mathrm{in}\ e_2]\!] &= \lambda pq.\mathrm{let}\ x = (\!|v|\!)\ \mathrm{in}\ [\![e_2]\!]pq \\
[\![\mathscr{S}k.e]\!] &= \lambda pq.\mathscr{S}_p k'.((\lambda k.(\!|e|\!))(\lambda y.\langle(\lambda\_.\Omega)(k'y)\rangle_q) \\
[\![e]\!] &= \lambda pq.\mathscr{S}_p k.\langle k(\!|e|\!)\rangle_q \qquad \text{if } e = x,\ c,\ \lambda x.e',\ \langle e'\rangle \text{ or throw}(k,e')
\end{aligned}
$$

Figure 5: Translation for Terms

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 calculus and also proves the type preservation property. Later, we will show a tagless-final implementation based on our translation which is another evidence that our translation actually works type-safely.

Figure 4 defines the translation rules for types, type schemes, type environments and triples.

As we have explained in earlier sections, we emulate ATM from the type α to the type β in 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 $\to [\![\tau]\!]$.

Types are translated in a natural way except the type for effectful functions $\sigma/\alpha \to \tau/\beta$; it is translated to a function type whose codomain is the translation of the triple $[\![\tau;\alpha,\beta]\!]$. Type schemes and type environments are translated naturally.

Figure 5 defines the translation for *typed* terms in λ^{ATM}, which consists of the translation $(\!|e|\!)$ for a pure term e, and the translation $[\![e]\!]$ for an effectful term e.

The first translation $(\!|e|\!)$ does little for most constructs but reset terms. A reset term $\langle e\rangle$ is translated to a term which creates two new prompts p and q, and inserts a combination of reset-p and shift-q as we explained earlier. Then it supplies p and q to its immediate subterm $[\![e]\!]$.

The second translation $[\![e]\!]$ does a lot; For application $e_1\,e_2$, it receives two prompts p and q, but also generates two new prompts r and s and distributes these prompts to its subterms etc. For a let-term let $x = v$ in e, it passes two prompts.

The translation of a shift term $\mathscr{S}k.e$ is a way more complicated than others; it receives two prompts p and q and invokes shift to capture the delimited continuation and binds k' to it. The captured delimited continuation is slightly different from the one which would have been obtained by the shift-operator in the source term, since we have inserted a combination of reset-p and shift-q at the position of reset. As we have explained in Section 3, this (bad) effect is resolved by detoxifying the continuation, which is realized by the involved term found in the translation above.

The last clause for $\llbracket e \rrbracket$ applies only when the type of the term e is derived by applying the exp rule as its last rule. In this case, the translation generates a term which receives two prompts p and q, detoxify the effect of the delimited continuation (mentioned above) by the combination of shift-p and reset-q.

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 $\llbracket \Gamma \rrbracket \vdash \llbracket e \rrbracket : \llbracket \tau;\alpha,\beta \rrbracket$ is derivable in the target calculus λ^{mpsr}.*

Similarly, if $\Gamma \vdash_p e : \tau$ is derivable in λ^{ATM}, so is $\llbracket \Gamma \rrbracket \vdash \llparenthesis e \rrparenthesis : \llbracket \tau \rrbracket$ 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:

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

By induction hypothesis, we can derive $\llbracket \Gamma \rrbracket \vdash \llbracket e_1 \rrbracket : \llbracket \sigma;\sigma,\tau \rrbracket$. Let $\Gamma' = \llbracket \Gamma \rrbracket, p : \llbracket \tau \rrbracket\ \mathrm{pr}, q : \llbracket \sigma \rrbracket\ \mathrm{pr}$ and $\Gamma'' = \Gamma', y : \llbracket \sigma \rrbracket$. We have the following derivation:

$$\dfrac{\dfrac{\Gamma' \vdash p : \llbracket \tau \rrbracket\ \mathrm{pr} \qquad \dfrac{\dfrac{\dfrac{\Gamma'' \vdash q : \llbracket \sigma \rrbracket\ \mathrm{pr} \qquad \Gamma'' \vdash y : \llbracket \sigma \rrbracket}{\Gamma'' \vdash \mathscr{S}_{q\_}.y : \llbracket \tau \rrbracket}}{\Gamma' \vdash \lambda v.\mathscr{S}_{q\_}.y : \llbracket \sigma \rrbracket \to \llbracket \tau \rrbracket} \qquad \dfrac{\Gamma' \vdash q : \llbracket \sigma \rrbracket\ \mathrm{pr} \qquad \dfrac{\Gamma' \vdash p : \llbracket \tau \rrbracket\ \mathrm{pr} \qquad \Gamma' \vdash \llbracket e_1 \rrbracket : \llbracket \sigma;\sigma,\tau \rrbracket}{\Gamma' \vdash \llbracket e_1 \rrbracket p : \llbracket \sigma \rrbracket\ \mathrm{pr} \to \llbracket \sigma \rrbracket}}{\Gamma' \vdash \llbracket e_1 \rrbracket pq : \llbracket \sigma \rrbracket}}{\Gamma' \vdash \big(\lambda y.\mathscr{S}_{q\_}.y\big)\,(\llbracket e_1 \rrbracket pq) : \llbracket \tau \rrbracket}}{\Gamma' \vdash \langle \big(\lambda y.\mathscr{S}_{q\_}.y\big)\,(\llbracket e_1 \rrbracket pq) \rangle_p : \llbracket \tau \rrbracket}}{\llbracket \Gamma \rrbracket \vdash \mathscr{P}p.\mathscr{P}q.\langle \big(\lambda v.\mathscr{S}_{q\_}.v\big)\,(\llbracket e_1 \rrbracket pq) \rangle_p : \llbracket \tau \rrbracket}$$

which derives $\llbracket \Gamma \rrbracket \vdash \llparenthesis \langle e_1 \rangle \rrparenthesis : \llbracket \tau \rrbracket$.

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

$$\dfrac{\Gamma, k : \tau \to \alpha \vdash_p e_1 : \beta}{\Gamma \vdash \mathscr{S}k.e_1 : \tau;\alpha,\beta}$$

By induction hypothesis $\llbracket \Gamma, k : \tau \to \alpha \rrbracket \vdash \llparenthesis e_1 \rrparenthesis : \llbracket \beta \rrbracket$ is derivable. Let $\Gamma' = \llbracket \Gamma \rrbracket, p : \llbracket \beta \rrbracket\ \mathrm{pr}, q : \llbracket \alpha \rrbracket\ \mathrm{pr}$, $\Gamma'' = \Gamma', k' : \llbracket \tau \rrbracket \to \llbracket \beta \rrbracket$, and $\Gamma''' = \Gamma'', y : \llbracket \tau \rrbracket$, then we have:

$$\dfrac{\dfrac{\Gamma'' \vdash p : \llbracket \beta \rrbracket\ \mathrm{pr} \qquad \dfrac{\dfrac{\Gamma'', k : \llbracket \tau \rrbracket \to \llbracket \alpha \rrbracket \vdash \llparenthesis e_1 \rrparenthesis : \llbracket \beta \rrbracket}{\Gamma'' \vdash \lambda k.\llparenthesis e_1 \rrparenthesis : (\llbracket \tau \rrbracket \to \llbracket \alpha \rrbracket) \to \llbracket \beta \rrbracket} \qquad \dfrac{\Gamma''' \vdash \langle (\lambda\_.\Omega)\,(k'y) \rangle_q : \llbracket \alpha \rrbracket}{\Gamma'' \vdash \lambda y.\langle (\lambda\_.\Omega)\,(k'y) \rangle_q : \llbracket \tau \rrbracket \to \llbracket \alpha \rrbracket}}{\Gamma'' \vdash (\lambda k.\llparenthesis e_1 \rrparenthesis)\,(\lambda y.\langle (\lambda\_.\Omega)\,(k'y) \rangle_q) : \llbracket \beta \rrbracket}}{\Gamma' \vdash \mathscr{S}_p k'.\,(\lambda k.\llparenthesis e_1 \rrparenthesis)\,(\lambda y.\langle (\lambda\_.\Omega)\,(k'y) \rangle_q) : \llbracket \tau \rrbracket}}{\llbracket \Gamma \rrbracket \vdash \lambda p.\lambda q.\mathscr{S}_p k'.\,(\lambda k.\llparenthesis e_1 \rrparenthesis)\,(\lambda y.\langle (\lambda\_.\Omega)\,(k'y) \rangle_q) : \llbracket \beta \rrbracket\ \mathrm{pr} \to \llbracket \alpha \rrbracket\ \mathrm{pr} \to \llbracket \tau \rrbracket}$$

which derives $\llbracket \Gamma \rrbracket \vdash \llbracket \mathscr{S}k.e_1 \rrbracket : \llbracket \tau;\alpha,\beta \rrbracket$. □

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

$$
\begin{aligned}
(\!| x |\!)^{1pass} &= x \\
(\!| c |\!)^{1pass} &= c \\
(\!| \lambda x.e |\!)^{1pass} &= \underline{\lambda} x.\underline{\lambda} pq.[\![e]\!]^{1pass}\overline{@}p\overline{@}q \\
(\!| \mathrm{throw}(k,e) |\!)^{1pass} &= k\underline{@}(\!| e_2 |\!)^{1pass} \\
(\!| \langle e\rangle |\!)^{1pass} &= \underline{\mathscr{P}}p.\underline{\mathscr{P}}q.\underline{\langle}(\underline{\lambda} y.\underline{\mathscr{S}_q}\_.y)\underline{@}([\![e]\!]^{1pass}\overline{@}p\overline{@}q)\underline{\rangle_p} \\
[\![e_1+e_2]\!]^{1pass} &= \overline{\lambda} pq.\underline{\mathscr{P}}r.([\![e_1]\!]^{1pass}\overline{@}r\overline{@}q)+([\![e_2]\!]^{1pass}\overline{@}p\overline{@}r) \\
[\![e_1\, e_2]\!]^{1pass} &= \overline{\lambda} pq.\underline{\mathscr{P}}r.\underline{\mathscr{P}}s.([\![e_1]\!]^{1pass}\overline{@}r\overline{@}s)\underline{@}([\![e_2]\!]^{1pass}\overline{@}p\overline{@}r)\underline{@}s\underline{@}q \\
[\![\mathrm{let}\ x=v\ \mathrm{in}\ e_2]\!]^{1pass} &= \overline{\lambda} pq.\underline{\mathrm{let}}\ x=(\!| v |\!)^{1pass}\ \underline{\mathrm{in}}\ [\![e_2]\!]^{1pass}\overline{@}p\overline{@}q \\
[\![\mathscr{S}k.e]\!]^{1pass} &= \overline{\lambda} pq.\underline{\mathscr{S}_p}k'.(\underline{\lambda} k.(\!| e |\!)^{1pass})\underline{@}\Big(\underline{\lambda} y.\underline{\langle}(\underline{\lambda}\_.\Omega)\underline{@}(k'\underline{@}y)\underline{\rangle_q}\Big) \\
[\![e]\!]^{1pass} &= \overline{\lambda} pq.\underline{\mathscr{S}_p}k.\underline{\langle}k\underline{@}(\!| e |\!)^{1pass}\underline{\rangle_q} \qquad \text{if } e=x,\ c,\ \lambda x.e',\ \langle e'\rangle \text{ or throw}(k,e')
\end{aligned}
$$

Figure 6: One-Pass PPS Translation

6 Optimization

The naive PPS translation in Section 4 works in theory, but is suboptimal for practical use, as it introduces too many prompt. To solve this problem, we will introduce an optimized PPS translation in this section.

For the purpose of comparison, we consider the term $\langle e_n\rangle$ for a natural number n, where e_n is $1+(2+\cdots+(n+\mathscr{S}k.\lambda x.\mathrm{throw}(k,x))\cdots)$.[3] Then we can derive $\vdash e_n : \mathtt{int};\ \mathtt{int},\ (\mathtt{int}/\alpha \to \mathtt{int}/\alpha)$ and $\vdash_p \langle e_n\rangle : (\mathtt{int}/\alpha \to \mathtt{int}/\alpha)$ for some type α, in the type system of λ^{ATM} augmented by the following type rule for addition:

$$\frac{\Gamma \vdash e_1 : \mathtt{int};\ \alpha,\ \gamma \qquad \Gamma \vdash e_2 : \mathtt{int};\ \gamma,\ \beta}{\Gamma \vdash e_1+e_2 : \mathtt{int};\ \alpha,\ \beta}$$

The type rule for addition in the target calculus is standard and omitted.

We define the (naive) PPS translation for addition by $[\![e_1+e_2]\!] \equiv \lambda p.\lambda q.\mathscr{P}r.\ ([\![e_1]\!]rq)+([\![e_2]\!]pr)$. It is easy to see that the naive PPS translation in the previous section translates $\langle e_n\rangle$ to a rather big term which dynamically generates $n+2$ prompts (2 for reset, and 1 for each addition) and passes them around. This is not ideal and needs improvement.

Eliminating Unnecessary Prompt Passing

We first eliminate unnecessary dynamic passing of prompts. Since the residual terms of the naive translation often contain $[\![e]\!]pq$ as subterms where $[\![e]\!]$ takes the form $\lambda p.\lambda q.\cdots$, they contain many beta redexes (administrative redexes) that can be eliminated at the translation time, by adjusting the one-pass CPS translation by Danvy and Filinski [9] to our PPS translation.

Figure 6 gives our one-pass PPS translation where function applications are made explicit (by the infix symbol @), and the overline (e.g. $\overline{@}$) means static constructs which are reduced at the translation time, while the underline (e.g. $\underline{@}$) means dynamic constructs which remain in the residual terms.

The one-pass PPS translation eliminates unnecessary prompt passing; Applying one-pass PPS translation to a term in λ^{ATM}, and reducing all static beta-redexes $(\overline{\lambda} p.e_1)\overline{@}q$ to $e_1\{q/p\}$, one obtains a term

[3] The term $\mathscr{S}k.\lambda x.\mathrm{throw}(k,x)$ is written as $\mathscr{S}k.k$ in the standard formulation.

$$(\!|\lambda x.e_1|\!)^{opt} = \begin{cases} \underline{\lambda} x.\underline{\lambda} pq.\underline{\mathscr{S}_p} k.\underline{\langle} k \underline{@} (\!|e_1|\!)^{opt} \underline{\rangle_q} & \text{if } e_1 \text{ is q-pure} \\ \underline{\lambda} x.\underline{\lambda} pq.[\![e]\!]^{opt}\overline{@}p\overline{@}q & \text{otherwise}\end{cases}$$

$$(\!|\langle e_1\rangle|\!)^{opt} = \begin{cases} (\!|e_1|\!)^{opt} & \text{if } e_1 \text{ is q-pure} \\ \underline{\mathscr{P}}p.\underline{\mathscr{P}}q.\underline{\langle}(\underline{\lambda} y.\underline{\mathscr{S}_{q}}\text{-}.y)\underline{@}([\![e_1]\!]^{opt}\overline{@}p\overline{@}q)\underline{\rangle_p} & \text{otherwise}\end{cases}$$

$$[\![e_1\, e_2]\!]^{opt} = \begin{cases} \overline{\lambda} pq.(\!|e_1|\!)^{opt}\underline{@}(\!|e_2|\!)^{opt}\underline{@}p\underline{@}q & \text{if } e_1 \text{ and } e_2 \text{ are q-pure} \\ \overline{\lambda} pq.\underline{\mathscr{P}}r.(\!|e_1|\!)^{opt}\underline{@}([\![e_2]\!]^{opt}\overline{@}p\overline{@}r)\underline{@}r\underline{@}q & \text{if } e_1 \text{ is q-pure and } e_2 \text{ is not} \\ \overline{\lambda} pq.\underline{\mathscr{P}}r.([\![e_1]\!]^{opt}\overline{@}p\overline{@}r)\underline{@}(\!|e_2|\!)^{opt}\underline{@}r\underline{@}q & \text{if } e_2 \text{ is q-pure and } e_1 \text{ is not} \\ \overline{\lambda} pq.\underline{\mathscr{P}}r.\underline{\mathscr{P}}s.([\![e_1]\!]^{opt}\overline{@}r\overline{@}s)\underline{@}([\![e_2]\!]^{opt}\overline{@}p\overline{@}r)\underline{@}s\underline{@}q & \text{otherwise}\end{cases}$$

$$(\!|\text{let } x = v \text{ in } e_1|\!)^{opt} = \underline{\text{let}}\; x = (\!|v|\!)^{opt}\; \underline{\text{in}}\; (\!|e_1|\!)^{opt} \quad \text{if } e_1 \text{ is q-pure}$$

$$[\![\text{let } x = v \text{ in } e_1]\!]^{opt} = \overline{\lambda} pq.\underline{\text{let}}\; x = (\!|v|\!)^{opt}\; \underline{\text{in}}\; [\![e_1]\!]^{opt}\overline{@}p\overline{@}q \quad \text{if } e_1 \text{ is not q-pure}$$

Figure 7: Optimized PPS Translation (new cases only)

without static constructs (constructs with overlines). The residual terms of one-pass PPS translation pass prompts only for function applications; in Figure 6, dynamic application for prompts $e\underline{@}p\underline{@}q$ appears in the term $[\![e_1\, e_2]\!]^{1pass}$ only.

The one-pass PPS translation gives a much better result than the naive one, as it eliminates unnecessary prompt passing, but it still generates as many prompts as the naive one.

Eliminating Unnecessary Prompt Generation

We eliminate unnecessary prompt generation. Our idea is simple; if the term being translated is pure, we do not have to generate prompts. In the translation of $[\![e_1+e_2]\!] = \lambda p.\lambda q.\mathscr{P}r.\ ([\![e_1]\!]rq)+([\![e_2]\!]pr)$, the new prompt r is used to bridge $[\![e_1]\!]$ and $[\![e_2]\!]$, and if one of e_1 and e_2 is pure, we can reuse p and q to simulate ATM. For instance, if e_1 is pure and e_2 is effectful, we can define $[\![e_1+e_2]\!] = \lambda p.\lambda q.(\!|e_1|\!)+([\![e_2]\!]pq)$.

To maximize the benefit of this optimization, we extend the notion of a pure term e_p to a *quasi-pure* term (or a q-pure term) e_q by:

$$e_p ::= x \mid c \mid \lambda x.e \mid \langle e\rangle \mid \text{throw}(k,e)$$
$$e_q ::= e_p \mid \text{let } x = v \text{ in } e_q$$

Namely, we allow nested let constructs appearing around pure terms. For instance, let $x=3$ in let $y=5$ in 7 is not pure, but is q-pure.

Figure 7 defines the optimized PPS translation; $(\!|e|\!)^{opt}$ for a q-pure term e, and $[\![e]\!]^{opt}$ for a non q-pure term e. We omit the cases whose translation is the same as those for one-pass PPS translation. The optimized PPS translation dispatches if each subterm is q-pure or not, and in the former case, it gives an optimized result where prompt generation is suppressed. Translation for addition can be defined similarly, for instance, $[\![e_1+e_2]\!]^{opt} = \overline{\lambda} pq.(\!|e_1|\!)^{opt}+([\![e_2]\!]^{opt}\overline{@}p\overline{@}q)$ if e_1 is q-pure and e_2 is not.

We compute $[\![\langle e_n\rangle]\!]$ and reduce all static redexes in it, to obtain the following term:[4]

$$\mathscr{P}p.\mathscr{P}q.\ \langle(\lambda y.\mathscr{S}_{q}\text{-}.y)(1+(2+\cdots+(n+(\mathscr{S}_p k'.(\lambda k.(\!|\lambda x.\text{throw}(k,x)|\!)^{opt})@(\lambda y.\langle\ldots\rangle_q)))\cdots))\rangle_p$$

[4]We omitted the underlines in the result.

```
module type Symantics = sig
  type τ pure                (* pure expression *)
  type (τ, α, β) eff         (* effectful expression *)
  type (σ, τ, α, β) efun     (* effectful function type *)
  type (σ, τ) pfun           (* pure function type *)
  val const : τ → τ pure
  val lam : (σ pure → (τ, α, β) eff) → (σ, τ, α, β) efun pure
  val app : ((σ, τ, α, β) efun, β, γ) eff
        → (σ, γ, 'd) eff → (τ, α, 'd) eff
  val throw : (σ, τ) pfun pure → σ pure → τ pure
  val shift : ((τ, α) pfun pure → β pure) → (τ, α, β) eff
  val reset : (σ, σ, τ) eff → τ pure
  val exp : τ pure → (τ, α, α) eff
  val run : τ pure → τ
end
```

Figure 8: Signature of the Embedded Language

where only two prompts are generated at the beginning of the computation. The result is quite close to the source term, and in fact, the source and target terms differ only at the control operators. Although it is possible to further optimize the results, by applying partial evaluation techniques for shift and reset (e.g. [1]), we believe that our translation is practical and efficient. In the next section, we show an implementation of our translation based on the optimized PPS translation.

We can prove that the optimized PPS translation preserves types where the type system of the target calculus λ^{mpsr} has two function types $\sigma\Rightarrow\tau$ (static) and $\sigma\underline{\to}\tau$ (dynamic). Namely, we can prove that, if $\Gamma \vdash e : \tau; \alpha, \beta$ is derivable in λ^{ATM} and e is not q-pure, so is $[\![\Gamma]\!]^{opt} \vdash [\![e]\!]^{opt} : [\![\beta]\!]^{opt}\ \text{pr}\Rightarrow[\![\alpha]\!]^{opt}\ \text{pr}\Rightarrow[\![\tau]\!]^{opt}$ in λ^{mpsr}. Similarly, if $\Gamma \vdash e : \tau; \alpha, \alpha$ is derivable and e is q-pure, or $\Gamma \vdash_p e : \tau$ is derivable, so is $[\![\Gamma]\!]^{opt} \vdash (\!|e|\!)^{opt} : [\![\tau]\!]^{opt}$. The details of this development and proofs are omitted.

7 Tagless-final embedding

We have implemented the calculus λ^{ATM} in Figure 2 and the naive and optimized PPS translations in Figures 4, 5 and 7 for a monomorphic version of λ^{ATM}.

Our implementation is based on the tagless-final style [6, 14], which allows one to embed a typed domain-specific language (DSL) in a metalanguage. In this style, the syntax as well as the typing rules of DSL are represented by a signature (an interface of modules), and its semantics is given as an interpretation of this signature. One of the important merits with this style is that type checking (or type inference) of DSL is automatically done by the type checker (or the type inferencer) of the metalanguage. Although we have already proved the subject reduction property of λ^{ATM} and the type preservation property for the PPS translations, implementing them in the tagless-final style gives us another indication for well typedness. It is particularly useful when we extend the source calculus and the translation; type errors are immediately raised by the type system of the metalanguage.

We have chosen OCaml plus the DelimCC library as the metalanguage, where DelimCC gives an efficient implementation for multi-prompt shift and reset [13]. We also give an implementation in MetaOCaml, a multi-stage extension of OCaml, to generate (and show) the translated terms, rather than immediately executing them.

Figure 8 shows the signature called Symantics for our source calculus λ^{ATM}. It represents the syntax and the typing rules of λ^{ATM}; the types τ `pure` and `(`τ`,` α`,` β`) eff`, resp., represent the relations $\Gamma \vdash_p e : \tau$ and $\Gamma \vdash e : \tau;\ \alpha,\ \beta$, resp. The type `(`$\sigma$`,` τ`,` α`,` β`) efun` represents the effectful function type $(\sigma/\alpha \to \tau/\beta)$ and `(`σ`,` τ`) pfun` the pure function type $\sigma \to \tau$ for continuations. Since all these types are kept abstract, we can arbitrarily instantiate them in different implementations. Each function but `run` encodes a typing rule in λ^{ATM}. For instance, the function `exp` encodes the exp rule in λ^{ATM}, and does not have a concrete primitive in DSL. The function `run` does not correspond to a constructor in DSL; it converts a DSL value to a value in the metalanguage, and is thus useful to test interpreters.

As an example, a DSL term $\mathscr{S}k.\lambda x.\mathrm{throw}(k,x)$ is represented by the term `shift (fun k → lam (fun x → exp (throw k x)))`, which encodes a type derivation of the above DSL term using higher-order abstract syntax. Note that, we can represent all and only typable terms in λ^{ATM} using this signatures, and the typability of embedded terms are checked by OCaml; all representable terms are typable *as they are constructed*.

In the tagless-final style, operational semantics of the embedded language is given as an interpretation of the Symantics signature, namely, a module of type Symantics. For this work, we have given two interpretations for each of two PPS translations, and thus obtained four interpreters. The two interpretations differ in the target; the first one, called the R interpreter, translates the source term and evaluates the result. The second one, called the S interpreter[5] translates the source term and generates the result as a code in MetaOCaml, which can be executed by the `run` primitive.

Due to lack of space, we cannot list the source code of these interpreters, but it should be noted that the two PPS translations (the naive one and the optimized one) have been successfully implemented in the tagless-final style. After extending the source calculus with conditional, recursion and so on, we can write programming examples such as list-append in Section 1, list-prefix and others, and running these examples gives correct answers.

Figure 9 shows a few results of the optimized translation with the S interpreter. We first define `append` and a test program `res1`. Then we translate `res1` and run it, to obtain the desired list. We then translate `append` itself (but not run it), to obtain the code `.<let rec g_56 ...>.` where the variable `g_56` corresponds to[6] the `append` function in Section 1. The result is instructive; control operators[7] are used only at the point where shift was there in the source term, in particular, no dynamic prompt generation happens during the recursive calls for append. This result clearly shows the merit of our optimized translation over the naive translation as well as the definitional CPS translation [8].

This implementation also provides a good evidence that our translation is type preserving. Thanks to the tagless-final style, our implementation is extensible, and in fact, it was easy to add primitives such as the fixpoint operator to our source language in a type-safe way.

8 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 for the naive and optimized translations, and implemented them in OCaml (and MetaOCaml) using the tagless-final style, which we think add further assurance for type safety.

[5] The S interpreter is actually a *compiler*.

[6] MetaOCaml renames all bound variables.

[7] `Delimcc.shift` is shift and `Delimcc.push_prompt` is reset.

```
module Example (S: SymPL) = struct
  open S
  let append = fixE (fun f x →
      ifE (null @@ exp x) (shift (fun k → k))
                  (head (exp x) @* app (exp f) (tail @@ exp x)))

  let res1 = run @@
             throw (reset (app (exp append) (exp @@ list [1;2;3])))
           (list [4;5;6])
end

# let _ = let module M = Example(SPL_opt) in M.res1;;
- : int list = [1; 2; 3; 4; 5; 6]
# let _ = let module M = Example(SPL_opt) in M.append;;
- : (int list, int list, int list, (int list, int list) SPL_opt.pfun)
    SPL_opt.efun SPL_opt.pure
= .<
let rec g_56 x_57 p_58 q_59 =
  if x_57 = []
  then
    Delimcc.shift p_58
      (fun k'_62  →
         (fun x_64  → x_64)
           (fun y_63  →
              Delimcc.push_prompt q_59
                (fun ()  →
                   (fun _  → Pervasives.failwith "Omega") @@ (k'_62
   y_63))))
  else
    (let v2_60 = g_56 (List.tl x_57) p_58 q_59 in
     let v1_61 = List.hd x_57 in v1_61 :: v2_60) in
g_56>.
```

Figure 9: Programming Examples

One may wonder if the reverse translation is possible. The answer is no, as our source calculus λ^{ATM} is strongly normalizing, while the target λ^{mpsr} is not. An open question is to identify the image of our translation which corresponds to the source calculus.

Let us briefly summarize related work. Rompf et al. [16] 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 [15] designed OchaCaml, which is an extension of Caml light 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 [17] studied monad-like structures to express shift and reset with Danvy and Filinski's type system [7]. Inspired by his work, Atkey [3, 4] proposed *parameterised monads* as a generalization of monads. They take two additional type parameters to express inputs and outputs, and therefore, can express answer-type modification. He studied categorical foundation of parameterised monads. Kiselyov [12] independently studied a similar notion, and gave an implementation and programming examples.

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 and the audience at WoC'15 for their constructive comments. The authors are supported in part by JSPS Grant-in-Aid for Scientific Research (25280020, 25540023).

References

[1] Kenichi Asai (2004): *Offline partial evaluation for shift and reset*. In: *PEPM*, pp. 3–14, doi:10.1145/1014007.1014009.

[2] Kenichi Asai & Yukiyoshi Kameyama (2007): *Polymorphic delimited continuations*. In: *APLAS*, *LNCS* 4807, Springer Berlin Heidelberg, pp. 239–254, doi:10.1007/978-3-540-76637-7_16.

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

[4] Robert Atkey (2009): *Parameterised notions of computation*. *Journal of Functional Programming* 19(3-4), p. 335, doi:10.1017/S095679680900728X.

[5] Malgorzata Biernacka & Dariusz Biernacki (2009): *Context-based proofs of termination for typed delimited-control operators*. *PPDP*, p. 289, doi:10.1145/1599410.1599446.

[6] Jacques Carette, Oleg Kiselyov & Chung-Chieh Shan (2009): *Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages*. *Journal of Functional Programming* 19(05), p. 509, doi:10.1017/S0956796809007205.

[7] Olivier Danvy & Andrzej Filinski (1989): *A Functional Abstraction of Typed Contexts*. Technical Report, Computer Science Department, University of Copenhagen.

[8] Olivier Danvy & Andrzej Filinski (1990): *Abstracting control*. In: *Proceedings of the 1990 ACM conference on LISP and functional programming*, ACM Press, New York, USA, pp. 151–160, doi:10.1145/91556.91622.

[9] Olivier Danvy & Andrzej Filinski (2009): *Representing Control: a Study of the CPS Transformation*. *Mathematical Structures in Computer Science* 2(04), p. 361, doi:10.1017/S0960129500001535.

[10] Matthias Felleisen (1988): *The theory and practice of first-class prompts*. In: *POPL*, ACM Press, New York, USA, pp. 180–190, doi:10.1145/73560.73576.

[11] Carl A. Gunter, Didier Rémy & Jon G. Riecke (1995): *A generalization of exceptions and control in ML-like languages*. In: *FPCA*, ACM Press, New York, USA, pp. 12–23, doi:10.1145/224164.224173.

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

[13] Oleg Kiselyov (2012): *Delimited control in OCaml, abstractly and concretely*. *Theoretical Computer Science* 435, pp. 56–76, doi:10.1016/j.tcs.2012.02.025.

[14] Oleg Kiselyov (2012): *Typed Tagless Final Interpreters*. In: *PSSGIP '10*, *LNCS* 7470, Springer Berlin Heidelberg, pp. 130–174, doi:10.1007/978-3-642-32202-0_3.

[15] Moe Masuko & Kenichi Asai (2011): *Caml Light + shift/reset = Caml Shift*. In: *First International Workshop on the Theory and Practice of Delimited Continuations*, pp. 33–46.

[16] Tiark Rompf, Ingo Maier & Martin Odersky (2009): *Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform*. In: *ICFP*, ACM Press, New York, USA, p. 317, doi:10.1145/1596550.1596596.

[17] Philip Wadler (1994): *Monads and composable continuations*. *LISP and Symbolic Computation* 7(1), pp. 39–55, doi:10.1007/BF01019944.