A Functional Abstraction of Typed Invocation Contexts

Youyou Cong 📣-envelope
Tokyo Institute of Technology, Japan

Chiaki Ishio 📣
Ochanomizu University, Tokyo, Japan

Kaho Honda 📣-envelope
Ochanomizu University, Tokyo, Japan

Kenichi Asai 📣-envelope/home
Ochanomizu University, Tokyo, Japan

Abstract

In their paper “A Functional Abstraction of Typed Contexts”, Danvy and Filinski show how to derive a type system of the \texttt{shift} and \texttt{reset} operators from a CPS translation. In this paper, we show how this method scales to Felleisen’s \texttt{control} and \texttt{prompt} operators. Compared to \texttt{shift} and \texttt{reset}, \texttt{control} and \texttt{prompt} exhibit a more dynamic behavior, in that they can manipulate a \emph{trail} of contexts surrounding the invocation of captured continuations. Our key observation is that, by adopting a functional representation of trails in the CPS translation, we can derive a type system that allows fine-grain reasoning of programs involving manipulation of invocation contexts.

2012 ACM Subject Classification

- Theory of computation → Functional constructs
- Theory of computation → Control primitives
- Theory of computation → Type structures

Keywords and phrases delimited continuations, control operators, control and prompt, CPS translation, type system


Supplementary Material Model (Agda Formalization): https://github.com/YouyouCong/fscd21-artifact; archived at sub:1:dir:9eaf9840fc9b223e030f633c3f9b3b5ea7b47bc6

Funding Youyou Cong: supported in part by JSPS KAKENHI under Grant No. 19K24339. Kenichi Asai: supported in part by JSPS KAKENHI under Grant No. JP18H03218.

Acknowledgements We sincerely thank the reviewers for their constructive feedback.

1 Introduction

Delimited continuations have been proven useful in diverse domains. Their applications range from representation of monadic effects [19], to formalization of partial evaluation [13], and to implementation of automatic differentiation [41]. As a means to handle delimited continuations, researchers have designed a variety of control operators [18, 15, 21, 16, 32]. Among them, Danvy and Filinski’s \texttt{shift/reset} operators [15] have a solid theoretical foundation: there are a canonical CPS translation [15], a general type system [14], and a set of equational axioms [25]. Recent work by Materzok and Biernacki [32, 31] has also fostered understanding of \texttt{shift0} and \texttt{reset0}, by establishing similar artifacts for these operators. Other variants, however, are not as well-understood as the aforementioned ones, due to their complex semantics.

Understanding the subtleties of control operators is important, especially given the rapid adoption of algebraic effects and handlers [36, 6] observed in the past decade. Effect handlers can be thought of as a form of exception handlers that provide access to delimited...
continuations. As suggested by the similarity in the functionality, effect handlers have a close
connection with control operators [20, 35], and in fact, they are often implemented using
control operators provided by the host language [27, 28]. This means, a well-established
type of control operators is crucial for safer and more efficient implementation of effect
handlers.

In this paper, we formalize a typed calculus of control and prompt, a pair of control
operators proposed by Felleisen [18]. These operators bring an interesting behavior into
programs: when a captured continuation k is invoked, the subsequent computation may
capture the context surrounding the invocation of k. From a practical point of view, the
ability to manipulate invocation contexts is useful for implementing sophisticated algorithms,
such as list reversing [8] and breadth-first traversal [10]. From a theoretical perspective, on
the other hand, this ability makes it hard to type programs in a way that fully reflects their
runtime behavior.

We address the challenge with typing by rigorously following Danvy and Filinski’s [14]
recipe for building a type system of a delimited control calculus. The idea is to analyze
the CPS translation of the calculus, and identify all the constraints that are necessary for
making a translated expression well-typed. In fact, the recipe has already been applied to
the control and prompt [26] operators, but the type system obtained is not satisfactory for
two reasons. First, the type system imposes certain restrictions on the contexts in which a
captured continuation may be invoked. Second, the type system does not precisely describe
the way contexts compose and propagate during evaluation. We show that, by choosing
a right representation of invocation contexts in the CPS translation, we can build a type
system without such limitations.

Below is a summary of our specific contributions:

- We present a type system of control and prompt that allows fine-grain reasoning
  of programs involving manipulation of invocation contexts. The type system is the
  control/prompt-equivalent of Danvy and Filinski’s [14] type system for shift/reset,
  in that it incorporates all and only constraints that are imposed by the CPS translation.

- We prove three properties of our calculus: type soundness, type preservation of the CPS
  translation, and termination of well-typed programs. Among these, termination relies on
  the precise typing of invocation contexts available in our calculus; indeed, the property
does not hold for the existing type system of control and prompt [26].

We begin with an informal account of control and prompt (Section 2), highlighting the dy-
namic behavior of these operators. We next formalize an untyped calculus of control/prompt
(Section 3) and its CPS translation (Section 4), which is equivalent to the translation given
by Shan [40]. Then, from the CPS translation, we derive a type system of our calculus
(Section 5), and prove its properties (Section 6). Lastly, we discuss related work (Section 7)
and conclude with future directions (Section 8).

As an artifact, we provide a formalization of our calculus and proofs in the Agda proof
assistant [34]. The code is checked using Agda version 2.6.0.1, and is available online at:

https://github.com/YouyouCong/fscd21-artifact

Relation to Prior Work. This is an updated and extended version of our previous paper [2].
The primary contributions of this paper are a complete proof of type soundness of the
proposed calculus, and a proper formalization of the target language of the CPS translation.
We have also changed the title to clarify the kind of contexts considered in the paper.
2 Control and Prompt

As a motivating example, consider the following program:

\[ (\langle Fk_1. \text{is0} \ (k_1 \ 5) \rangle + \langle Fk_2. \text{b2s} \ (k_2 \ 8) \rangle) \]

Throughout the paper, we write \( F \) to mean \texttt{control} and \( () \) to mean \texttt{prompt}. We also assume two primitive functions: \texttt{is0}, which tells us if a given integer is zero or not, and \texttt{b2s}, which converts a boolean into a string "true" or "false".

Under the call-by-value, left-to-right evaluation strategy, the above program evaluates in the following way:

\[
\begin{align*}
(\langle Fk_1. \text{is0} \ (k_1 \ 5) \rangle + \langle Fk_2. \text{b2s} \ (k_2 \ 8) \rangle) \\
= (\text{is0} \ (k_1 \ 5) [\lambda x. x + (\langle Fk_2. \text{b2s} \ (k_2 \ 8) \rangle)/k_1]) \\
= (\text{is0} \ (5 + (\langle Fk_2. \text{b2s} \ (k_2 \ 8) \rangle))) \\
= (\text{b2s} \ (k_2 \ 8) [\lambda x. \text{is0} \ (5 + x)/k_2]) \\
= (\text{b2s} \ (\text{is0} \ (5 + 8))) \\
= (\text{b2s} \ \text{false}) \\
= (\"false\")
\end{align*}
\]

The first \texttt{control} operator captures the delimited context up to the enclosing \texttt{prompt}, namely \([\_] + (\langle Fk_2. \text{b2s} \ (k_2 \ 8) \rangle)\) (where \([\_]\) denotes a hole). The captured context is then reified into a function \( \lambda x. x + (\langle Fk_2. \text{b2s} \ (k_2 \ 8) \rangle) \), and evaluation shifts to the body \texttt{is0} \ (k_1 \ 5), where \( k_1 \) is the reified continuation. After \( \beta \)-reducing the invocation of \( k_1 \), we obtain another \texttt{control} in the evaluation position. This \texttt{control} captures the context \texttt{is0} \ (5 + \[\_\]), which is a composition of two contexts: the addition context originally surrounding the \texttt{control} construct, and the application of \texttt{is0} surrounding the invocation of \( k_1 \). The context is then reified into a function \( \lambda x. \text{is0} \ (5 + x) \), and evaluation shifts to the body \texttt{b2s} \ (k_2 \ 8), where \( k_2 \) is the reified continuation. By \( \beta \)-reducing the invocation of \( k_2 \), we obtain the expression \texttt{b2s} \ (\text{is0} \ (5 + 8)), where the original delimited context, the invocation context of \( k_1 \), and the invocation context of \( k_2 \) are all composed together. The expression returns the value "false" to the enclosing \texttt{prompt} clause, and the evaluation of the whole program finishes with this value.

From the above example, we can make two observations. First, a \texttt{control} operator can capture the context surrounding the invocation of a previously captured continuation. More generally, \texttt{control} may capture a \textit{trail} of such invocation contexts. The ability comes from the absence of the delimiter in the body of captured continuations. Indeed, if we replace \texttt{control} with \texttt{shift} \((S)\) in the above program, the second \texttt{shift} would have no access to the context \texttt{is0} \([\_]\), since the first \texttt{shift} would insert a \texttt{reset} into the continuation \( k_1 \). As a consequence, the program gets stuck after the application of \( k_2 \).

\[
\begin{align*}
(\langle Sk_1. \text{is0} \ (k_1 \ 5) \rangle + \langle Sk_2. \text{b2s} \ (k_2 \ 8) \rangle) \\
= (\text{is0} \ (k_1 \ 5) [\lambda x. x + (\langle Sk_2. \text{b2s} \ (k_2 \ 8) \rangle)/k_1]) \\
= (\text{is0} \ (5 + (\langle Sk_2. \text{b2s} \ (k_2 \ 8) \rangle))) \\
= (\text{is0} \ (\text{b2s} \ (k_2 \ 8) [\lambda x. (5 + x)/k_2])) \\
= (\text{is0} \ (\text{b2s} \ (5 + 8))) \\
= (\text{is0} \ (\text{b2s} \ 13))
\end{align*}
\]
A Functional Abstraction of Typed Invocation Contexts

Syntax

\[ v ::= c \mid x \mid \lambda x. e \] Values

\[ e ::= v \mid e e \mid F k e \mid \langle e \rangle \] Expressions

Evaluation Contexts

\[ E ::= [\cdot] \mid E e \mid v E \mid \langle E \rangle \] General Contexts

\[ F ::= [\cdot] \mid F e \mid v F \] Pure Contexts

Reduction Rules

\[ E[(\lambda x. e) v] \rightsquigarrow E[e[v/x]] \] (\beta)

\[ E[[F[Fk e]]] \rightsquigarrow E[[e[\lambda x. F[x]/k]]] \] (F)

\[ E[v] \rightsquigarrow E[v] \] (P)

Figure 1 \( \lambda_F \): A Calculus of control and prompt.

The second observation is that a trail of invocation contexts can be heterogeneous. In our particular example, the first continuation \( k_1 \) is called in a int-to-bool context, whereas the second continuation \( k_2 \) is called in a bool-to-string context. These are apparently distinct types, and furthermore, the input and output types of each context are also different.

It turns out that our motivating example would be judged ill-typed by the existing type system for control and prompt [26]. This is because the type system imposes the following restrictions on the type of invocation contexts.

- All invocation contexts within a prompt clause must have the same type.
- For each invocation context, the input and output types must be the same.

We claim that, a fully general type system of control and prompt should be more flexible about the type of invocation contexts. Now the question is: Is it possible to allow such flexibility? Our answer is “yes”. As we will see in Section 5, we can build a type system that accommodates invocation contexts having varying types, and that accepts our motivating example as a well-typed program.

3 \( \lambda_F \): A Calculus of control and prompt

In Figure 1, we present \( \lambda_F \), a \( \lambda \)-calculus featuring the control and prompt operators. The calculus has a separate syntactic category for values, which, in addition to variables and abstractions, has a set of constants \( c \), such as integers, booleans, and string literals. Expressions consist of values, application, and delimited control constructs control and prompt.

We equip \( \lambda_F \) with a call-by-value, left-to-right evaluation strategy. As is usual with delimited control calculi, there are two groups of evaluation contexts: general contexts \( (E) \) and pure contexts \( (F) \). Their difference is that general contexts may contain prompt surrounding a hole, while pure contexts can never have such prompt. The distinction is used in the reduction rule \( (F) \) of control, which says, control always captures the context up to the nearest enclosing prompt. In the reduct, we see that the body of a captured continuation is not surrounded by prompt, as we observed in the previous section. On the other hand, the body of control is evaluated in a prompt clause. The reduction rule \( (P) \) for prompt simply removes a delimiter surrounding a value.

Note that \( \lambda_F \) is currently presented as an untyped calculus. We will introduce types in Section 5, according to the CPS translation to be defined in the next section.
Syntax

\[ v ::= e \mid x \mid \lambda x. e \mid () \]

Values

\[ e ::= v \mid e e \mid (\text{case } t \text{ of } () \Rightarrow e \mid k \Rightarrow e) \]

Expressions

Evaluation Contexts

\[ E ::= [\cdot] \mid E e \mid v E \mid (\text{case } E \text{ of } () \Rightarrow e \mid k \Rightarrow e) \]

Reduction Rules

\[
\begin{align*}
E[(\lambda x. e) v] & \rightsquigarrow E[e[v/x]] \quad (\beta) \\
E[\text{case } () \text{ of } () \Rightarrow e_1 \mid k \Rightarrow e_2] & \rightsquigarrow E[e_1] \quad \text{(case-()} \\
E[\text{case } v \text{ of } () \Rightarrow e_1 \mid k \Rightarrow e_2] & \rightsquigarrow E[e_2[v/k]] \quad \text{(case-k)}
\end{align*}
\]

Figure 2 \( \lambda_C \): Target Calculus of CPS Translation.

4 CPS Translation

As we mentioned earlier, the type system of a delimited control calculus is often derived from a translation into continuation-passing style (CPS) [14]. When the source calculus has control and prompt, a CPS translation exposes both continuations and trails of invocation contexts. Trails can be represented either as a list of functions [8, 9] or as a composition of functions [40]. While previous work [26] on typing control and prompt adopts the list representation, we adopt the functional representation, as it fits better for the purpose of building a general type system (see Section 5 for details).

4.1 \( \lambda_C \): Target Calculus of CPS Translation

In Figure 2, we define the target calculus of the CPS translation, which we call \( \lambda_C \). The calculus is a pure, call-by-value \( \lambda \)-calculus featuring the unit value (\( () \)), which represents an empty trail, and a case analysis construct, which allows inspection of trails. Note that a non-empty trail is represented as a regular function.

As in \( \lambda_F \), we evaluate \( \lambda_C \) programs under a call-by-value, left-to-right strategy. The particular choice of evaluation strategy is not necessary in our setting, but it is mandatory if the source and target calculi of the CPS translation have non-control effects (such as non-termination and I/O), because the result of the translation may have non-tail calls.

4.2 The CPS Translation

In Figure 3, we present the CPS translation \([_\_\_]\) from \( \lambda_F \) to \( \lambda_C \), which is equivalent to the translation given by Shan [40]. The translation converts an expression into a function that takes in a continuation \( k \) and a trail \( t \). The trail is the composition of the invocation contexts encountered so far, and is used together with a continuation to produce an answer (hence a continuation now receives a trail). Below, we detail the translation of three representative constructs: variables, prompt, and control.
Variables. The translation of a variable is an \( \eta \)-expanded version of the standard, call-by-value translation. The trivial use of the current trail \( t \) communicates the fact that a variable can never change the trail during evaluation. In general, the CPS translation of a pure expression uniformly calls the continuation with an unmodified trail.

Prompt. The translation of \texttt{prompt} has the same structure as the translation of variables, because \texttt{prompt} forms a pure expression. The translated body \([v] \) is run with the identity continuation \( k_{id} \) and an empty trail \( () \), describing the behavior of \texttt{prompt} as a control delimiter. Note that, in this CPS translation, the identity continuation is not the identity function. It receives a value \( v \) and a trail \( t \), and behaves differently depending on whether \( t \) is empty or not. When \( t \) is empty, the identity continuation simply returns \( v \). When \( t \) is non-empty, \( t \) must be a function composed of one or more invocation contexts, which looks like \( \lambda x. E_n[... E_1[x] ...] \). In this case, the identity continuation builds an expression \( E_n[... E_1[v] ...] \) by calling the trail with \( v \) and \( () \).

Control. The translation of \texttt{control} shares the same pattern with the translation of \texttt{prompt}, because its body is evaluated in a \texttt{prompt} clause (as defined by the \( (F) \) rule in Figure 1). The translated body \([c] \) is applied a substitution that replaces the variable \( c \) with the trail \( t \circ (k' :: t') \), describing how the trail is extended when a captured continuation is invoked\(^1\). Recall that, in this CPS translation, trails are represented as functions. The \( @ \) and \( :: \) operators are thus defined as a function producing a function\(^2\). More specifically, these operators compose contexts in a first-captured, first-called manner (as we can see from the second clause of \( :: \) ). Notice that \( :: \) is defined as a recursive function\(^4\). The reason is that, when extending a trail \( t \) with a continuation \( k \), we need to produce a function that takes in a trail \( t' \), which in turn must be composed with a continuation \( k' \).

The CPS translation is correct with respect to the definitional abstract machine given by Biernacka et al.\(^3\). The statement is proved by Shan\(^4\), using the functional correspondence\(^5\) between evaluators and abstract machines.

As a last note, let us mention here that the alternative CPS translation of \texttt{control} and \texttt{prompt}, where trails are represented as lists, can be obtained by replacing \( () \) with the empty list, and the two operations \( @ \) and \( :: \) with ones that work on lists.

5 Type System

Having defined a CPS translation, we now derive a type system of \( \lambda x. \). We proceed in three steps. First, we specify the syntax of trail types (Section 5.1). Next, we identify an appropriate form of typing judgment (Section 5.2). Lastly, we define the typing rules of individual syntactic constructs (Section 5.3). In each step, we contrast our outcome with its counterpart in Kameyama and Yonezawa\'s\(^6\) type system, showing how different representations of trails in the CPS translation lead to different typing principles.

---

1 The identity continuation \( k_{id} \) and the empty trail \( () \) correspond to the \texttt{send} function and the \texttt{#f} value of Shan\(^4\), respectively.

2 There is in fact a superficial difference between our CPS translation and Shan’s original translation\(^4\). In the rule for \texttt{control}, we replace the continuation variable \( c \) with the function \( \lambda x. \lambda k'. \lambda t'. x (t \circ (k' :: t')) \), while Shan replaces \( c \) with \( \lambda x. \lambda k'. \lambda t'. (k :: t) x (k' :: t') \). However, by expanding the definition of \( @ \) and \( :: \), we can easily see that the two functions are equivalent. We prefer the one that uses \( @ \) because it is closer to the abstract machine given by Biernacka et al.\(^3\), as well as the list-based CPS translation derived from it.

3 The \( :: \) function is equivalent to Shan’s \texttt{compose} function.

4 While recursive, the \( :: \) function is guaranteed to terminate, as the types of the two arguments become smaller in every three successive recursive calls (or they reach the base case in fewer steps).

5 The \( (F) \) rule in the CPS translation leads to different typing principles.
\[
\begin{align*}
[c] &= \lambda k. \lambda t. k \cdot c \cdot t \\
[x] &= \lambda k. \lambda t. k \cdot x \cdot t \\
[\lambda x. e] &= \lambda k. \lambda t. k \cdot (\lambda x. \lambda k'. \lambda t'. [e] \cdot k' \cdot t') \cdot t \\
[e_1 \cdot e_2] &= \lambda k. \lambda t. [e_1] \cdot (\lambda v_1. \lambda t_1. [e_2] \cdot (\lambda v_2. \lambda t_2. v_1 \cdot v_2 \cdot k \cdot t_2) \cdot t_1) \cdot t \\
[F_c. e] &= \lambda k. \lambda t. [e] \cdot \lambda x. \lambda k'. \lambda t'. k \cdot x \cdot (t @ (k' :: t')) / [e] \cdot k_{id} ()
\end{align*}
\]

\[k_{id} = \lambda v. \lambda t. \text{case } t \text{ of } () \Rightarrow v \mid k \Rightarrow k \cdot v ()
\]
\[\_ @ @ = \lambda k. \lambda t'. \text{case } t \text{ of } () \Rightarrow t' \mid k \Rightarrow k :: t'
\]
\[\_ :: _ = \lambda k. \lambda t. \text{case } t \text{ of } () \Rightarrow k \mid k' \Rightarrow \lambda v. \lambda t'. k \cdot v (k' :: t')
\]

**Figure 3** CPS Translation of \(\lambda F\) Expressions.

### 5.1 Syntax of Trail Types

Recall from Section 4.1 that, in \(\lambda_C\), trails have two possible forms: () or a function. Correspondingly, in \(\lambda_F\), trail types \(\mu\) are defined by a two-clause grammar: \(\bullet \mid \tau \rightarrow (\mu) \tau'\).

The latter type is interpreted in the following way.

- The trail accepts a value of type \(\tau\).
- The trail is to be composed with a context of type \(\mu\).
- After the composition, the trail produces a value of type \(\tau'\).

Put differently, \(\tau\) is the input type of the innermost invocation context, \(\tau'\) is the output type of the context to be composed in the future, and \(\mu\) is the type of this future context.

To better understand non-empty trail types, let us revisit the example from Section 2.

\((\langle F k_1. is0 (k_1 5) \rangle + (\langle F k_2. b2s (k_2 8) \rangle))
\]

\[= (is0 (k_1 5) \cdot [\lambda x. x + (\langle F k_2. b2s (k_2 8) \rangle) / k_1])
\]

\[= (is0 (5 + (\langle F k_2. b2s (k_2 8) \rangle)))
\]

\[= (b2s (k_2 8) \cdot [\lambda x. is0 (5 + x) / k_2])
\]

\[= (b2s (is0 (5 + 8)))
\]

\[= "false"
\]

When the continuation \(k_1\) is invoked, the trail is extended with the context \(is0 [\cdot]\). This context will be composed with the invocation context \(b2s [\cdot]\) of \(k_2\) later in the reduction sequence. Therefore, the trail at this point is given type \(\text{int} \rightarrow (\text{bool} \rightarrow (\bullet) \text{string}) \text{string}\), consisting of the input type of \(is0\), the type of \(b2s\), and the output type of \(b2s\).

When the continuation \(k_2\) is invoked, the trail is extended with the context \(b2s [\cdot]\) (hence the whole trail looks like \(b2s (is0 [\cdot])\)). This context will not be composed with any further contexts in the subsequent steps of reduction. Therefore, the trail at this point is given type \(\text{int} \rightarrow (\bullet) \text{string}\), consisting of the input type of \(is0\), the type of an empty trail, and the output type of \(b2s\).

Observe that our trail types can be inhabited by heterogeneous trails, where the input and output types of each invocation context may be different. The flexibility is exactly what we wish a general type system of control and prompt to have, as we discussed in Section 2.
Comparison with Previous Work. In the CPS translation of Kameyama and Yonezawa [26], a trail is treated as a list of invocation contexts. Such a list is given a recursive type $\text{Trail}(\rho)$ defined as follows:

$$\text{Trail}(\rho) = \mu X. \text{list}(\rho \to X \to \rho)$$

We can easily see that the definition restricts the type of invocation contexts in two ways. First, all invocation contexts in a trail must have the same type. This is because lists are homogeneous by definition. Second, each invocation context must have equal input and output types. This is a direct consequence of the first restriction. The two restrictions prevent one from invoking a continuation in a context such as $\text{is0} \[\cdot\]$ or $\text{b2s} \[\cdot\]$. Moreover, the use of the list type makes empty and non-empty trails indistinguishable at the level of types, and extension of trails undetectable in types. On the other hand, these limitations allow one to use an ordinary expression type (such as $\text{int}$, instead of a type designed specifically for trails) to encode the information of trails in the control/prompt calculus. That is, if a trail has type $\text{Trail}(\rho)$ in the target, it has type $\rho$ in the source.

5.2 Typing Judgment

We next turn our attention to the typing of a CPS-translated expression. Suppose $e$ is a $\lambda_f$ expression of type $\tau$. In the general case, the CPS counterpart of $e$ is typed in the following way:

$$[e] = \lambda k_{\tau \to \mu_\alpha \to \alpha}. \lambda t_{\mu_\beta}. e^{/\beta}$$

Here, $\alpha$ and $\beta$ are answer types, representing the return type of the enclosing prompt before and after evaluation of $e$. It is well-known that delimited control can make the two answer types distinct [14], and since they are needed for deciding the typability of programs, they must be integrated into the typing judgment. The other pair of types, $\mu_\beta$ and $\mu_\alpha$, are trail types, representing the composition of invocation contexts encountered before and after evaluation of $e$. As control can extend a given trail by invoking a captured continuation, the two trail types may be different, and have to be integrated into the typing judgment.

Summing up the above discussion, we conclude that a fully general typing judgment for control and prompt must carry five types, as follows:

$$\Gamma \vdash e : \tau \langle \mu_\alpha \rangle \langle \alpha \rangle \langle \mu_\beta \rangle \beta$$

We place the types in the same order as their appearance in the annotated CPS expression. That is, the first three types $\tau$, $\mu_\alpha$, and $\alpha$ correspond to the continuation of $e$, the next one $\mu_\beta$ represents the trail required by $e$, and the last one $\beta$ stands for the eventual value returned by $e$. We will hereafter call $\alpha$ and $\beta$ initial and final answer types, and $\mu_\beta$ and $\mu_\alpha$ initial and final trail types – be careful of the direction in which answer types and trail types change.

With the typing judgment specified, we can define the syntax of expression types in $\lambda_f$ (Figure 4). Expression types are formed with base types $\iota$ (such as $\text{int}$ and $\text{bool}$) and arrow types $\tau_1 \to \tau_2 \langle \mu_\alpha \rangle \langle \alpha \rangle \langle \mu_\beta \rangle \beta$. Notice that the codomain of arrow types carries five components. These types represent the control effect of a function’s body, and correspond exactly to the five types that appear in a typing judgment.

Comparison with Previous Work. In the type system developed by Kameyama and Yonezawa [26], a CPS-translated expression is typed in the following way:

$$\lambda k_{\tau \to \text{Trail}(\rho) \to \alpha}. \lambda t_{\text{Trail}(\rho)} e^{/\beta}$$
It is obvious that the typing is not as general as ours, since the two trail types are equal. This constraint is imposed by the list representation of trails: since a list type is insensitive to extension, we can always use a trail of the same type for the evaluation of $e$ and the rest of the computation. Thus, Kameyama and Yonezawa arrive at a typing judgment carrying four types, with the last one ($\rho$) representing the information of trails:

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

Correspondingly, they assign source functions an arrow type of the form $\tau_1 \rightarrow \tau_2, \alpha, \beta/\rho$.

### 5.3 Typing Rules

Now we are ready to define the typing rules of $\lambda_\mathcal{F}$ (Figure 4). As in the previous section, we elaborate the typing rules of variables, **prompt**, and **control**.

#### Variables.

Recall that the CPS translation of variables is an $\eta$-expanded version of the standard translation. If we annotate the types of each subexpression, a translated variable would look like:

$$\lambda k^{\tau \rightarrow \mu_\alpha \rightarrow \alpha} \cdot \lambda t^{\mu_\alpha}. (k \ x \ t)^\alpha$$

We see duplicate occurrences of the answer type $\alpha$ and the trail type $\mu_\alpha$. The duplication arises from the application $k \ x \ t$, and reflects the fact that a variable cannot change the answer type or the trail type. By a straightforward conversion from the annotated expression into a typing judgment, we obtain rule (Var) in Figure 4. In general, when the subject of a typing judgment is a pure construct, the answer types and trail types both coincide.

#### Prompt.

We next analyze the CPS translation of **prompt**, again with type annotations.

$$\lambda k^{\tau \rightarrow \mu_\alpha \rightarrow \alpha} \cdot \lambda t^{\mu_\alpha}. (k \ (\mathcal{E} \eta \mathcal{K} k \ \mathcal{I}d \ (\beta \rightarrow \mu_\alpha \rightarrow \beta') \rightarrow \bullet \rightarrow \tau \ k \ \mathcal{I}d \ ()) \ t)^\alpha$$

As $\langle e \rangle$ is a pure expression, we again have equal answer types $\alpha$ and trail types $\mu_\alpha$ for the whole expression. The initial trail type $\bullet$ and final answer type $\tau$ of $e$ are determined by the application $[e] \ k \ \mathcal{I}d \ ()$ and $k \ (\mathcal{E} \eta \mathcal{K} k \ \mathcal{I}d \ ())$, respectively. What is left is to ensure that the application of $[e]$ to the identity continuation $k \ \mathcal{I}d \ ()$ is type-safe. In our type system, we use a relation $\text{is-id-trail}(\tau, \mu, \tau')$ to ensure this type safety. The relation holds when the type $\tau \rightarrow \mu \rightarrow \tau'$ can be assigned to the identity continuation. The valid combination of $\tau$, $\mu$, and $\tau'$ is derived from the definition of the identity continuation, repeated below:

$$\lambda v^{\tau}. \lambda t^{\mu}. \text{case } t \text{ of } () \Rightarrow v^{\tau'} \mid k \Rightarrow (k \ v ())^{\tau'}$$

When $t$ is an empty trail $()$ of type $\bullet$, the return value of $k \ \mathcal{I}d \ ()$ is $v$, which has type $\tau$. Since the expected return type of $k \ \mathcal{I}d$ is $\tau'$, we need the equality $\tau \equiv \tau'$.

When $t$ is a non-empty trail $k$ of type $\tau_1 \rightarrow \mu \rightarrow \tau_1'$, the return value of $k \ \mathcal{I}d \ ()$ is the result of the application $k \ v ()$, which has type $\tau_1'$. Since the expected return type of $k \ \mathcal{I}d$ is $\tau'$, we need the equality $\tau' \equiv \tau_1'$. Furthermore, since $k$ must accept $v$ and $()$ as arguments, we need the equalities $\tau \equiv \tau_1$ and $\mu \equiv \bullet$.

We define $\text{is-id-trail}$ as an encoding of these constraints, and in the rule (Prompt), we use $\text{is-id-trail}(\beta, \mu_\alpha, \beta')$ to constrain the type of the continuation of $e$. Now, it is statically guaranteed that $e$ can be safely evaluated in an empty context.
12:10 A Functional Abstraction of Typed Invocation Contexts

Syntax of Types

- $\tau, \alpha, \beta ::= \ell | \tau \to \tau \langle \mu_\alpha \rangle \alpha \langle \mu_\beta \rangle \beta$
- $\mu, \mu_\alpha, \mu_\beta ::= \bullet | \tau \to \langle \mu \rangle \tau$

Expression Types

Trail Types

Typing Rules

- $\Gamma \vdash e : \tau \langle \mu_\alpha \rangle \alpha \langle \mu_\beta \rangle \beta$ (CONST)
- $\Gamma \vdash x : \tau \in \Gamma$ (VAR)
- $\Gamma, x : \tau_1 \vdash e : \tau_2 \langle \mu_\alpha \rangle \alpha \langle \mu_\beta \rangle \beta$ (ABS)
- $\Gamma \vdash e_1 : (\tau_1 \to \tau_2 \langle \mu_\alpha \rangle \alpha \langle \mu_\beta \rangle \beta) \langle \mu_\gamma \rangle \gamma$
- $\Gamma \vdash e_2 : (\tau_1 \to \tau_2 (\mu_\alpha) \alpha (\mu_\beta) \beta) \langle \mu_\gamma \rangle \gamma$
- $\Gamma \vdash e_1, e_2 : (\tau_1 \to \tau_2 (\mu_\alpha) \alpha (\mu_\beta) \beta)$ (APP)
- $\Gamma, k : \tau \to \tau_1 (\mu_1) \tau_1' (\mu_2) \alpha \vdash e : \gamma (\mu_{id}) \gamma' \langle \bullet \rangle \beta$
- is-id-trail($\gamma, \mu_{id}, \gamma'$)
- compatible($\langle \tau_1 \to (\mu_1) \tau_1' \rangle, \mu_2, \mu_0$)
- $\Gamma \vdash Fk.e : \tau \langle \mu_\alpha \rangle \alpha \langle \mu_\beta \rangle \beta$ (CONTROL)
- $\Gamma \vdash Fk.e : \tau \langle \mu_\alpha \rangle \alpha \langle \mu_\beta \rangle \beta$ (PROMPT)

Auxiliary Relations

- is-id-trail($\tau, \bullet, \tau'$) = $\tau \equiv \tau'$
  (first branch of $k_{id}$ in Figure 3)
- is-id-trail($\tau, (\tau_1 \to \langle \mu \rangle \tau_1')$, $\tau'$) = $(\tau \equiv \tau_1) \land (\tau' \equiv \tau_1') \land (\mu \equiv \bullet)$
  (second branch of $k_{id}$ in Figure 3)

- compatible($\bullet, \mu_2, \mu_3$) = $\mu_2 \equiv \mu_3$
  (first branch of $\oplus$ in Figure 3)
- compatible($\mu_1, \bullet, \mu_3$) = $\mu_1 \equiv \mu_3$
  (first branch of $::$ in Figure 3)
- compatible($\langle \tau_1 \to (\mu_1) \tau_1' \rangle, \mu_2, \bullet$) = $\perp$
  (no counterpart in Figure 3)
- compatible($\langle \tau_1 \to (\mu_1) \tau_1', \mu_2, (\tau_3 \to (\mu_3) \tau_3') \rangle = (\tau_1 \equiv \tau_3) \land (\tau_1' \equiv \tau_3') \land (\text{compatible}(\mu_2, \mu_3, \mu_1))$
  (second branch of $::$ in Figure 3)

- [Figure 4] Type System of $\lambda \mathcal{F}$. We assume a global signature $\Sigma$ mapping constants to base types.
We now show that the motivating example discussed in Section 2 is judged well-typed in $\lambda^\text{exp}$-5. The well-typedness of the whole program largely relies on the well-typedness of the two control constructs, so let us look at the typing of these constructs:

\[ \lambda^\text{exp}. [e] \Rightarrow \beta = \lambda^\text{exp}. k x (t @ (k' :: t'))/e \]

As the body $e$ of control is evaluated in a prompt clause, we again have an empty initial trail type for $e$, and we know that the types $\gamma$, $\mu_1\delta$, and $\gamma'$ must satisfy the is-id-trail relation. What is left is to ensure that the composition of contexts in $t @ (k' :: t')$ is type-safe. In our type system, we use a relation $\text{compatible}(\mu_1, \mu_2, \mu_3)$ to ensure this type safety. The relation holds when composing a context of type $\mu_1$ and another context of type $\mu_2$ results in a context of type $\mu_3$. Intuitively, the relation can be thought of as an addition over trail types, and the valid combination of $\mu_1$, $\mu_2$, and $\mu_3$ is derived from the definition of the $@$ and $::$ functions.

\[ t^{\mu_1} @ t'^{\mu_2} = \text{case } f \text{ of } () \Rightarrow t^{\mu_3} \mid k \Rightarrow (k' :: t')^{\mu_3} \]

The first clause of $@$ and that of $::$ are straightforward: they tell us that the empty trail type $\bullet$ serves as the left and right identity of the addition.

The second clause of $::$ requires more careful reasoning. The return value of this case is the result of the application $k v (k' :: t')$, which has type $\tau_1'$. Since the expected return type of $::$ is $\tau_1'$, we need the equality $\tau_1' \equiv \tau_3'$. Moreover, since $k$ must accept $v$ and $k' :: t'$ as arguments, we need the equality $\tau_1 \equiv \tau_3$, as well as a recursive use of compatible, where the third type is $\mu_1$.

The definition of $@$ and $::$ further tells us that, when either of their arguments is non-empty, the result of composition cannot be an empty trail. In terms of types, this can be rephrased as: when one of $\mu_1$ and $\mu_2$ is an arrow type, $\mu_3$ cannot be the empty trail type.

We define compatible as an encoding of these constraints, and in the (CONTROL) rule, we use two instances of this relation to constrain the type of contexts appearing in $t @ (k' :: t')$. Among the two instances, the first one compatible$(\tau_1 \rightarrow \langle \mu_1 \rangle \tau_1', \mu_2, \mu_0)$ states that consing $k'$ to $t'$ is type-safe, and the result has type $\mu_0$. The second one compatible$(\mu_3, \mu_0, \mu_3)$ states that appending $t$ to $k' :: t'$ is type-safe, and the result has type $\mu_3$, which is required by the continuation $k$ of the whole control expression.

Comparison with Previous Work. In the type system of Kameyama and Yonezawa [26], the typing rules for control and prompt are defined as follows:

\[
\frac{\Gamma, k : \tau \rightarrow \rho, \rho, \alpha/\rho \vdash e : \gamma, \gamma, \beta/\gamma}{\Gamma \vdash \mathcal{F} k : e : \tau, \alpha, \beta/\rho} \quad \text{(CONTROL)}
\]

\[
\frac{\Gamma \vdash e : \rho, \rho, \tau/\rho}{\Gamma \vdash (e) : \tau, \alpha, \alpha/\sigma} \quad \text{(PROMPT)}
\]

The rules are simpler than the corresponding rules in our type system. In particular, there is no equivalent of is-id-trail or compatible, since the homogeneous nature of trails makes those relations trivial. Note that the input and output types shared among invocation contexts come from the body of prompt, namely the first occurrence of $\rho$ in the premise of (PROMPT).

5.4 Typing Motivating Example

We now show that the motivating example discussed in Section 2 is judged well-typed in $\Lambda^\text{exp}$-5. The well-typedness of the whole program largely relies on the well-typedness of the two control constructs, so let us look at the typing of these constructs:

\[ \text{Our online artifact includes an Agda implementation of this example (exp4 in lambdaf.agda).} \]
A Functional Abstraction of Typed Invocation Contexts

\[ \vdash F_{k_1}.\text{is0}\ (k_1\ 5) : \text{int}\ \langle\mu_1\rangle\ \text{string}\ (\bullet)\ \text{string}\ \]

\[ \vdash F_{k_2}.\text{b2s}\ (k_2\ 8) : \text{int}\ \langle\mu_2\rangle\ \text{string}\ \langle\mu_1\rangle\ \text{string}\ \]

For brevity, we write \(\mu_1\) to mean \(\text{int} \rightarrow \langle\text{bool} \rightarrow (\bullet)\ \text{string}\rangle\ \text{string}\), and \(\mu_2\) to mean \(\text{int} \rightarrow (\bullet)\ \text{string}\). We can see how the trail type changes from empty \((\bullet)\), to one that refers to a future context \((\mu_1)\), and to one that mentions no further context \((\mu_2)\). In particular, \(\mu_2\) is the result of “adding” \(\mu_1\) and the type of \(\text{b2s}\ [\cdot]\); that is, the invocation of \(k_2\) discharges the future context awaited by \(\text{is0}\ [\cdot]\). The trail type \(\mu_2\) serves as the final trail type of the body of the enclosing \text{prompt}, and as it allows us to establish the is-id-trail relation required by \((\text{Prompt})\), we can conclude that the whole program is well-typed.

### Properties

The type system of \(\lambda_F\) enjoys various pleasant properties. First, the type system is sound, that is, well-typed programs do not go wrong [33]. Following Wright and Felleisen [42], we prove type soundness via the preservation and progress theorems.

▶ **Theorem 1** (Preservation). If \(\Gamma \vdash e : \tau\ \langle\mu_\alpha\rangle\ \alpha\ \langle\mu_\beta\rangle\ \beta\) and \(e \rightsquigarrow e'\), then \(\Gamma \vdash e' : \tau\ \langle\mu_\alpha\rangle\ \alpha\ \langle\mu_\beta\rangle\ \beta\).

**Proof.** The proof is by induction on the typing derivation, and is formalized in Agda (the \text{Reduce} relation in \text{lambdaf-red.agda}). Note that, to prove type preservation of the control reduction (rule \((F)\) in Figure 1), we need to define a set of typing rules for evaluation contexts.

▶ **Theorem 2** (Progress). If \(\bullet \vdash e : \tau\ \langle\mu_\alpha\rangle\ \alpha\ \langle\mu_\beta\rangle\ \beta\), then either (i) \(e\) is a value, (ii) \(e\) takes a step, or (iii) \(e\) is a stuck term of the form \(F[\cdot\ F\cdot\ e']\).

**Proof.** The proof is by induction on the typing derivation. The third alternative is commonly found in the progress property of effectful calculi [3, 43]. We can remove this alternative by refining our type system to one that can decide the purity of an expression; with this refinement, we can state the usual progress theorem for pure expressions (which include top-level programs).

▶ **Theorem 3** (Type Soundness). If \(\bullet \vdash \langle e \rangle : \tau\ \langle\mu_\alpha\rangle\ \alpha\ \langle\mu_\alpha\rangle\ \alpha\), then evaluation of \(\langle e \rangle\) does not get stuck.

**Proof.** The statement is a direct implication of preservation and progress. The need for the top-level \text{prompt} stems from the fact that a well-typed, closed expression may be a stuck term (corresponding to the third clause of the progress theorem).

Secondly, our CPS translation preserves typing, \(i.e.,\) it converts a well-typed \(\lambda_F\) expression into a well-typed \(\lambda_C\) expression. To establish this theorem, we define the type system of \(\lambda_C\) (Figure 5) and a CPS translation \(\ast\) on \(\lambda_F\) types (Figure 6).

Let us elaborate on rule \((\text{Case})\) in Figure 5, which is the only non-trivial typing rule. This rule is used to type the case analysis construct in the three auxiliary functions of the CPS translation, namely \(k_{id}, @, \text{and }::\). Unlike the standard typing rule for case analysis, rule \((\text{Case})\) type-checks the two branches using equality assumptions \(\mu \equiv \bullet\) and
Syntax of Types

\[ \tau = \iota \mid \tau \rightarrow \tau \mid \bullet \]

Typing Rules

\[
\begin{align*}
\Gamma \vdash e : \iota & \quad (\text{Const}) \\
\Gamma \vdash x : \tau & \quad (\text{Var}) \\
\Gamma, \ x : \tau_1 \vdash e : \tau_2 & \quad (\text{ABS}) \\
\Gamma \vdash () : \bullet & \quad (\text{UNIT}) \\
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1 & \quad (\text{APP}) \\
\forall \tau_1, \mu, \mu_1 : \Gamma, \ k : \mu^* \rightarrow \langle \mu_1 \rangle \tau_1 \vdash e_2 : \tau & \quad (\text{CASE})
\end{align*}
\]

\[\text{Figure 5 Type System of } \lambda_C\]. We assume a global signature \(\Sigma\) mapping constants to base types.

\(\mu \equiv \tau_1 \rightarrow \langle \mu_1 \rangle \tau_1\). These assumptions, together with the is-id-trail and compatible relations, allow us to fill in the gap between the expected and actual return types. To see how the assumptions work, consider the typing of \(k\ id\):

\[\lambda v. \lambda t. \text{case } t \text{ of } () \Rightarrow v \mid k \Rightarrow (k \ v ())^r\]

In the first branch, we see an inconsistency between the expected return type \(\tau'\) and the actual return type \(\tau\). However, by the typing rules defined in Figure 4, we know that \(k_{id}\) is used only when the relation is-id-trail(\(\tau, \mu, \tau'\)) holds, and that if \(\mu \equiv \bullet\), we have \(\tau \equiv \tau'\). The equality assumption \(\mu \equiv \bullet\) made available by rule (CASE) allows us to derive \(\tau \equiv \tau'\) and conclude that the first branch has the correct type. Similarly, in the second branch, we use the equality assumption \(\mu \equiv \tau_1 \rightarrow \langle \mu_1 \rangle \tau_1\) to derive \(\tau \equiv \tau_1, \tau' \equiv \tau_1^r, \text{ and } \mu_1 \equiv \bullet\), which imply the well-typedness of the application \(k \ v ()\). The \@ and :: functions can be typed in an analogous way.

\[\text{Theorem 4 (Type Preservation of CPS Translation). If } \Gamma \vdash e : \tau \langle \mu_\alpha \rangle \alpha \langle \mu_\beta \rangle \beta \text{ in } \lambda_x, \text{ then } \Gamma^* \vdash [e] : (\tau^* \rightarrow \mu^*_x \rightarrow \alpha^*) \rightarrow \mu^*_x \rightarrow \beta^* \text{ in } \lambda_C\].

\[\text{Proof. The proof is by induction on the typing derivation, and is formalized in Agda (the cpse function in cps.agda). With the carefully designed rule for case analysis, we can prove the statement in a straightforward manner, as our type system is directly derived from the CPS translation.} \]

Thirdly, and most interestingly, our type system enjoys termination.

\(\) The use of equality assumptions in (CASE) is inspired by dependent pattern matching [12] available in dependently typed languages. Our case analysis is weaker than the dependent variant, in that the return type only depends on the type of the scrutinee, not on the scrutinee itself.
Translation of Expression Types

\[ t^* = t \]
\[ (\tau_1 \rightarrow \tau_2 \langle \mu_\alpha \rangle \alpha (\mu_\beta \beta)^* = \tau_1^* \rightarrow (\tau_2^* \rightarrow \mu_\alpha^* \rightarrow \alpha^*) \rightarrow \mu_\beta^* \rightarrow \beta^*) \]

Translation of Trail Types

\[ \cdot^* = \cdot \]
\[ (\tau \rightarrow \langle \mu \rangle \tau')^* = \tau^* \rightarrow \mu^* \rightarrow \tau'^* \]

Figure 6 CPS Translation of \( \lambda \) Types.

\[ \text{Theorem 5 (Termination). If } \Gamma \vdash e : \tau \langle \cdot \rangle \alpha \langle \cdot \rangle \alpha, \text{ then there exists some value } v \text{ such that } e \leadsto^* v, \text{ where } \leadsto^* \text{ is the reflexive, transitive closure of } \leadsto \text{ defined in Figure 1.} \]

Proof. The statement is witnessed by a CPS interpreter of \( \lambda X \) implemented in Agda (the \texttt{go} function in \texttt{lambdaf.agda}). Since every well-typed Agda program terminates, and since our interpreter is judged well-typed, we know that evaluation of \( \lambda X \) expressions must terminate. ▶

The termination property is unique to our type system. In the existing type system of Kameyama and Yonezawa [26], it is possible to write a well-typed program that does not evaluate to a value, as shown below:

\[
\langle (Fk_1.k_1 1;k_1 1); (Fk_2.k_2 1;k_2 1) \rangle
\]
\[
= \langle k_1 1;k_1 1[\lambda x.x; (Fk_2.k_2 1;k_2 1)]/k_1 \rangle
\]
\[
= \langle (Fk_2.k_2 1;k_2 1); (\lambda x.(Fk_2.k_2 1;k_2 1) 1)) 1 \rangle
\]
\[
= \langle k_2 1;k_2 1[\lambda y.y; (\lambda x.(Fk_2.k_2 1;k_2 1) 1) 1]/k_2 \rangle
\]
\[
= \langle (Fk_2.k_2 1;k_2 1); (\lambda y.y; (\lambda x.(Fk_2.k_2 1;k_2 1) 1) 1) 1 \rangle
\]
\[
= ... \]

We see that the two succeeding invocations of captured continuations result in duplication of control, leading to a looping behavior.

The well-typedness of the above program in Kameyama and Yonezawa’s type system is due to the limited expressiveness of trail types. More precisely, their trail types are mere expression types, which carry no information about the type of contexts to be composed in the future. In our type system, on the other hand, trail types explicitly mention the type of future contexts. This prevents us from duplicating expressions forever, which in turn allows us to statically reject the above looping program.

7 Related Work

Variations of Control Operators. There are four variants of delimited control operators in the style of control and prompt, differing in whether the control operator keeps the surrounding delimiter, and whether it inserts a delimiter into the captured continuation [16].
Among those variants, \texttt{shift} and \texttt{reset} \cite{15} are called \textit{static}, as the extent of a captured continuation can always be determined from the lexical structure of the program. Other variants are all \textit{dynamic}, since the control operator may capture the invocation contexts of previously captured continuations (as \texttt{control} does), or the meta-contexts outside of the original innermost delimiter (as \texttt{shift0} \cite{32} does), or both kinds of contexts (as \texttt{control0} \cite{16} does). Dynamic control operators all have a semantics that involves a trail-like structure, containing the contexts beyond the lexically enclosing one.

\textbf{Type Systems for Control Operators.} The CPS-based approach to designing type systems has been applied to several variants of delimited control operators, including \texttt{shift/reset} \cite{14,3}, \texttt{control/prompt} \cite{26}, and \texttt{shift0/prompt0} \cite{32}. While Danvy and Filinski \cite{14} consider all expressions as effectful (like we do), subsequent studies distinguish between pure and effectful expressions. This is typically done by not mentioning the answer type (and trail type) of syntactically pure expressions. Having pure expressions makes more programs typable \cite{3,26,32}, and allows more efficient compilation via a selective CPS translation \cite{37,32,4}.

\textbf{Algebraic Effects and Handlers.} In the past decade, algebraic effects and handlers \cite{6,36} have become a popular tool for handling delimited continuations. A prominent feature of effect handlers is that a captured continuation is used at the delimiter site. This makes it unnecessary to keep track of answer types in the type system, as we can decide within a handler whether the use of a continuation is consistent with the actual context. The irrelevance of answer types in turn makes the connection between the type system and CPS translation looser. Indeed, type systems of effect handlers \cite{5,27} existed before their CPS semantics \cite{29,24,23}. Also, type-preserving CPS translation of effect handlers is an open problem in the community \cite{23}.

8 \textbf{Conclusion and Future Work}

In this paper, we show how to derive a general type system for the \texttt{control} and \texttt{prompt} operators. The main idea is to identify all the typing constraints from a CPS translation, where trails are represented as a composition of functions.

The present study is part of a long-term project on formalizing delimited control facilities whose theory is not yet fully developed. In the rest of this section, we describe several directions for future work.

\textbf{Implementation.} Having designed a type system for \texttt{control} and \texttt{prompt}, a natural next step is to implement a language based on the type system. To make the language practical, we need to address the following challenges. First, we must extend our type system with a form of effect polymorphism or subtyping \cite{26,32}, in order to allow a function or continuation to be called in different contexts. We are currently attempting to adapt Kameyama and Yonezawa’s treatment of trail polymorphism to a setting where every typing judgment carries two trail types. Second, we need to design an algorithm for type inference and type checking. We conjecture that answer types can be left implicit in the user program, because it is the case in a calculus featuring \texttt{shift} and \texttt{reset} \cite{3}. On the other hand, we anticipate that some of the trail types need to be explicitly given by the user, as it does not seem always possible to synthesize the intermediate trail types \((\mu_0, \tau_1 \rightarrow \langle \mu_1 \rangle \tau_1', \text{ and } \mu_2)\) in the \texttt{(Control)} rule. Once we have done these, we will develop an implementation (possibly as an extension of OchaCaml \cite{30}) and experiment with various programs from the continuations literature.
Equational Theory. The semantics of control and prompt is currently given in the form of a CPS translation or an abstract machine [40, 9]. A more direct approach to specifying the semantics of these operators is to establish an equational theory, that is, we identify a set of equations that are sound and complete with respect to the existing semantics. Such equations are particularly useful for compilation: for instance, they enable converting an optimization in a CPS compiler into a rewrite in a direct-style (DS) program [38]. We intend to develop an equational theory for control and prompt, following previous studies on call/cc [38], shift/reset [25], and shift0/reset0 [31].

Reflection. An equational theory can be strengthened to a reflection [39] by defining a DS translation that serves as a left inverse of the CPS translation. Having a reflection means every reduction in the DS calculus has a corresponding reduction in the CPS calculus, and vice versa. We seek to establish a reflection for control and prompt, by extending Biernacki et al.’s [11] reflection for shift and reset.

Control0/Prompt0 and Shallow Effect Handlers. The control0 and prompt0 operators are a variation of control and prompt that remove the matching delimiter upon capturing of a continuation (which is a feature of shift0 and reset0). We plan to formalize a typed calculus of control0/prompt0, as well as their equational theory, by combining the insights from our work on control/prompt and previous studies on shift0/reset0 [32, 31]. As shown by Piróg et al. [35], there exists a pair of macro translations [17] between control0/prompt0 and shallow effect handlers [22]. Therefore, an equational theory for control0/prompt0 could potentially serve as a stepping stone to optimization of shallow handlers, which has not yet been explored [43].

References


12:18 A Functional Abstraction of Typed Invocation Contexts