A Functional Abstraction of Typed Invocation Contexts

Youyou Cong
Tokyo Institute of Technology, Japan

Chiaki Ishio
Ochanomizu University, Tokyo,Japan

Kaho Honda
Ochanomizu University, Tokyo, Japan

Kenichi Asai
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 `shift` and `reset` operators from a CPS translation. In this paper, we show how this method scales to Felleisen's `control` and `prompt` operators. Compared to `shift` and `reset`, `control` and `prompt` exhibit a more dynamic behavior, in that they can manipulate a *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

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.12

Supplementary Material *Model (Agda Formalization)*: `https://github.com/YouyouCong/fscd21-artifact`; archived at `swh: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 `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 `shift0` and `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 theory 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 Fillinski'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 dynamic 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:

$$\langle (\mathcal{F}k_1.\, \texttt{is0}\ (k_1\ 5)) + (\mathcal{F}k_2.\, \texttt{b2s}\ (k_2\ 8))\rangle$$

Throughout the paper, we write $\mathcal{F}$ to mean `control` and $\langle\rangle$ to mean `prompt`. We also assume two primitive functions: `is0`, which tells us if a given integer is zero or not, and `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:

$$\begin{aligned}
&\langle (\mathcal{F}k_1.\, \texttt{is0}\ (k_1\ 5)) + (\mathcal{F}k_2.\, \texttt{b2s}\ (k_2\ 8))\rangle\\
&= \langle \texttt{is0}\ (k_1\ 5)\,[\lambda x.\, x + (\mathcal{F}k_2.\, \texttt{b2s}\ (k_2\ 8))/k_1]\rangle\\
&= \langle \texttt{is0}\ (5 + (\mathcal{F}k_2.\, \texttt{b2s}\ (k_2\ 8)))\rangle\\
&= \langle \texttt{b2s}\ (k_2\ 8)\,[\lambda x.\, \texttt{is0}\ (5 + x)/k_2]\rangle\\
&= \langle \texttt{b2s}\ (\texttt{is0}\ (5 + 8))\rangle\\
&= \langle \texttt{b2s}\ (\texttt{is0}\ 13)\rangle\\
&= \langle \texttt{b2s}\ \texttt{false}\rangle\\
&= \langle \texttt{"false"}\rangle\\
&= \texttt{"false"}
\end{aligned}$$

The first `control` operator captures the delimited context up to the enclosing `prompt`, namely $[.] + (\mathcal{F}k_2.\, \texttt{b2s}\ (k_2\ 8))$ (where $[.]$ denotes a hole). The captured context is then reified into a function $\lambda x.\, x + (\mathcal{F}k_2.\, \texttt{b2s}\ (k_2\ 8))$, and evaluation shifts to the body $\texttt{is0}\ (k_1\ 5)$, where k_1 is the reified continuation. After β-reducing the invocation of k_1, we obtain another `control` in the evaluation position. This `control` captures the context $\texttt{is0}\ (5 + [.])$, which is a composition of *two* contexts: the addition context originally surrounding the `control` construct, and the application of `is0` surrounding the invocation of k_1. The context is then reified into a function $\lambda x.\, \texttt{is0}\ (5 + x)$, and evaluation shifts to the body $\texttt{b2s}\ (k_2\ 8)$, where k_2 is the reified continuation. By β-reducing the invocation of k_2, we obtain the expression $\texttt{b2s}\ (\texttt{is0}\ (5 + 8))$, 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 `prompt` clause, and the evaluation of the whole program finishes with this value.

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

$$\begin{aligned}
&\langle (\mathcal{S}k_1.\, \texttt{is0}\ (k_1\ 5)) + (\mathcal{S}k_2.\, \texttt{b2s}\ (k_2\ 8))\rangle\\
&= \langle \texttt{is0}\ (k_1\ 5)\,[\lambda x.\, \langle x + (\mathcal{S}k_2.\, \texttt{b2s}\ (k_2\ 8))\rangle/k_1]\rangle\\
&= \langle \texttt{is0}\ \langle 5 + (\mathcal{S}k_2.\, \texttt{b2s}\ (k_2\ 8))\rangle\rangle\\
&= \langle \texttt{is0}\ \langle \texttt{b2s}\ (k_2\ 8)\,[\lambda x.\, \langle 5 + x\rangle/k_2]\rangle\rangle\\
&= \langle \texttt{is0}\ \langle \texttt{b2s}\ \langle 5 + 8\rangle\rangle\rangle\\
&= \langle \texttt{is0}\ \langle \texttt{b2s}\ 13\rangle\rangle
\end{aligned}$$

Syntax

$v ::= c \mid x \mid \lambda x.\, e$ Values $\qquad e ::= v \mid e\ e \mid \mathcal{F}k.\, e \mid \langle e \rangle$ Expressions

Evaluation Contexts

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

$F ::= [.] \mid F\ e \mid v\ F$ Pure Contexts

Reduction Rules

$$E[(\lambda x.\, e)\ v] \rightsquigarrow E[e\,[v/x]] \qquad (\beta)$$
$$E[\langle F[\mathcal{F}k.\, e]\rangle] \rightsquigarrow E[\langle e\,[\lambda x.\, F[x]/k]\rangle] \qquad (\mathcal{F})$$
$$E[\langle v \rangle] \rightsquigarrow E[v] \qquad (\mathcal{P})$$

Figure 1 $\lambda_{\mathcal{F}}$: 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 $\lambda_{\mathcal{F}}$: A Calculus of `control` and `prompt`

In Figure 1, we present $\lambda_{\mathcal{F}}$, a λ-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 $\lambda_{\mathcal{F}}$ 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 ($\mathcal{F}$) 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 ($\mathcal{P}$) for `prompt` simply removes a delimiter surrounding a value.

Note that $\lambda_{\mathcal{F}}$ 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

$$v ::= c \mid x \mid \lambda x.\, e \mid () \qquad \text{Values}$$
$$e ::= v \mid e\ e \mid (\texttt{case}\ t\ \texttt{of}\ () \Rightarrow e \mid k \Rightarrow e) \qquad \text{Expressions}$$

Evaluation Contexts

$$E ::= [.] \mid E\ e \mid v\ E \mid (\texttt{case}\ E\ \texttt{of}\ () \Rightarrow e \mid k \Rightarrow e)$$

Reduction Rules

$$E[(\lambda x.\, e)\ v] \rightsquigarrow E[e\,[v/x]] \qquad (\beta)$$
$$E[\texttt{case}\ ()\ \texttt{of}\ () \Rightarrow e_1 \mid k \Rightarrow e_2] \rightsquigarrow E[e_1] \qquad (\textsc{case-}())$$
$$E[\texttt{case}\ v\ \texttt{of}\ () \Rightarrow e_1 \mid k \Rightarrow e_2] \rightsquigarrow E[e_2\,[v/k]] \qquad (\textsc{case-}k)$$

Figure 2 λ_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 λ_C: Target Calculus of CPS Translation

In Figure 2, we define the target calculus of the CPS translation, which we call λ_C. The calculus is a pure, call-by-value λ-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 $\lambda_{\mathcal{F}}$, we evaluate λ_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 $\lambda_{\mathcal{F}}$ to λ_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 η-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 `prompt` has the same structure as the translation of variables, because `prompt` forms a pure expression. The translated body $[\![e]\!]$ is run with the identity continuation k_{id} and an empty trail $()$[1], describing the behavior of `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 $\lambda x.\, E_n[\dots E_1[x] \dots]$. In this case, the identity continuation builds an expression $E_n[\dots E_1[v] \dots]$ by calling the trail with v and $()$.

Control. The translation of `control` shares the same pattern with the translation of `prompt`, because its body is evaluated in a `prompt` clause (as defined by the $(\mathcal{F})$ rule in Figure 1). The translated body $[\![e]\!]$ is applied a substitution that replaces the variable c with the trail $t\,@\,(k' :: t')$, describing how the trail is extended when a captured continuation is invoked[2]. Recall that, in this CPS translation, trails are represented as functions. The @ and :: operators are thus defined as a function producing a function[3]. 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. [7]. The statement is proved by Shan [40], using the functional correspondence [1] between evaluators and abstract machines.

As a last note, let us mention here that the alternative CPS translation of `control` and `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 $\lambda_{\mathcal{F}}$. 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 [26] 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 `send` function and the `#f` value of Shan [40], respectively.

2 There is in fact a superficial difference between our CPS translation and Shan's original translation [40]. In the rule for `control`, we replace the continuation variable c with the function $\lambda x.\, \lambda k'.\, \lambda t'.\, k\ x\ (t\,@\,(k' :: t'))$, 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 Biernacki et al. [9], as well as the list-based CPS translation derived from it.

3 The :: function is equivalent to Shan's `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).

$$
\begin{aligned}
[\![c]\!] &= \lambda k.\, \lambda t.\, k\ c\ t \\
[\![x]\!] &= \lambda k.\, \lambda t.\, k\ x\ t \\
[\![\lambda x.\, e]\!] &= \lambda k.\, \lambda t.\, k\ (\lambda x.\, \lambda k'.\, \lambda t'.\, [\![e]\!]\ k'\ t'\ )\ t \\
[\![e_1\ e_2]\!] &= \lambda k.\, \lambda t.\, [\![e_1]\!]\ (\lambda v_1.\, \lambda t_1.\, [\![e_2]\!]\ (\lambda v_2.\, \lambda t_2.\, v_1\ v_2\ k\ t_2)\ t_1)\ t \\
[\![\mathcal{F}c.\, e]\!] &= \lambda k.\, \lambda t.\, [\![e]\!]\, [\lambda x.\, \lambda k'.\, \lambda t'.\, k\ x\ (t \mathbin{@} (k' :: t'))/c]\ k_{id}\ () \\
[\![\langle e \rangle]\!] &= \lambda k.\, \lambda t.\, k\ ([\![e]\!]\ k_{id}\ ())\ t \\
\\
k_{id} &= \lambda v.\, \lambda t.\, \mathtt{case}\ t\ \mathtt{of}\ () \Rightarrow v \mid k \Rightarrow k\ v\ () \\
\_@\_ &= \lambda t.\, \lambda t'.\, \mathtt{case}\ t\ \mathtt{of}\ () \Rightarrow t' \mid k \Rightarrow k :: t' \\
\_::\_ &= \lambda k.\, \lambda t.\, \mathtt{case}\ t\ \mathtt{of}\ () \Rightarrow k \mid k' \Rightarrow \lambda v.\, \lambda t'.\, k\ v\ (k' :: t')
\end{aligned}
$$

Figure 3 CPS Translation of $\lambda_{\mathcal{F}}$ Expressions.

5.1 Syntax of Trail Types

Recall from Section 4.1 that, in λ_C, trails have two possible forms: $()$ or a function. Correspondingly, in $\lambda_{\mathcal{F}}$, trail types μ are defined by a two-clause grammar: $\bullet \mid \tau \to \langle \mu \rangle\ \tau'$. The latter type is interpreted in the following way.

- The trail accepts a value of type τ.
- The trail is to be composed with a context of type μ.
- After the composition, the trail produces a value of type τ'.

Put differently, τ is the input type of the innermost invocation context, τ' is the output type of the context to be composed in the future, and μ is the type of this future context.

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

$$
\begin{aligned}
&\langle (\mathcal{F}k_1.\, \mathtt{is0}\ (k_1\ 5)) + (\mathcal{F}k_2.\, \mathtt{b2s}\ (k_2\ 8)) \rangle \\
&= \langle \mathtt{is0}\ (k_1\ 5)\, [\lambda x.\, x + (\mathcal{F}k_2.\, \mathtt{b2s}\ (k_2\ 8))/k_1] \rangle \\
&= \langle \mathtt{is0}\ (5 + (\mathcal{F}k_2.\, \mathtt{b2s}\ (k_2\ 8))) \rangle \\
&= \langle \mathtt{b2s}\ (k_2\ 8)\, [\lambda x.\, \mathtt{is0}\ (5 + x)/k_2] \rangle \\
&= \langle \mathtt{b2s}\ (\mathtt{is0}\ (5 + 8)) \rangle \\
&= \mathtt{"false"}
\end{aligned}
$$

When the continuation k_1 is invoked, the trail is extended with the context `is0` $[.]$. This context will be composed with the invocation context `b2s` $[.]$ of k_2 later in the reduction sequence. Therefore, the trail at this point is given type $\mathtt{int} \to \langle \mathtt{bool} \to \langle \bullet \rangle\ \mathtt{string} \rangle\ \mathtt{string}$, 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` $[.]$ (hence the whole trail looks like `b2s` (`is0` $[.]$)). 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 $\mathtt{int} \to \langle \bullet \rangle\ \mathtt{string}$, 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 $\mathtt{Trail}(\rho)$ defined as follows:

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

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 `is0` [.] or `b2s` [.]. 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 `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 $\mathtt{Trail}(\rho)$ in the target, it has type ρ in the source.

5.2 Typing Judgment

We next turn our attention to the typing of a CPS-translated expression. Suppose e is a $\lambda_{\mathcal{F}}$ expression of type τ. In the general case, the CPS counterpart of e is typed in the following way:

$$[\![e^\tau]\!] = \lambda k^{\tau \to \mu_\alpha \to \alpha}.\, \lambda t^{\mu_\beta}.\, e'^{\beta}$$

Here, α and β 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, μ_β and μ_α, 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:

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

We place the types in the same order as their appearance in the annotated CPS expression. That is, the first three types τ, μ_α, and α correspond to the continuation of e, the next one μ_β represents the trail required by e, and the last one β stands for the eventual value returned by e. We will hereafter call α and β initial and final answer types, and μ_β and μ_α 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 $\lambda_{\mathcal{F}}$ (Figure 4). Expression types are formed with base types ι (such as `int` and `bool`) and arrow types $\tau_1 \to \tau_2 \,\langle \mu_\alpha \rangle\, \alpha \,\langle \mu_\beta \rangle\, \beta$. 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:

$$\lambda k^{\tau \to \mathtt{Trail}(\rho) \to \alpha}.\, \lambda t^{\mathtt{Trail}(\rho)}.\, e'^{\beta}$$

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 (ρ) representing the information of trails:

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

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

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 η-expanded version of the standard translation. If we annotate the types of each subexpression, a translated variable would look like:

$$\lambda k^{\tau \to \mu_\alpha \to \alpha}.\, \lambda t^{\mu_\alpha}.\, (k\ x\ t)^\alpha$$

We see duplicate occurrences of the answer type α and the trail type μ_α. 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.

$$\lambda k^{\tau \to \mu_\alpha \to \alpha}.\, \lambda t^{\mu_\alpha}.\, (k\ (\llbracket e \rrbracket^{(\beta \to \mu_{id} \to \beta') \to \bullet \to \tau}\ k_{id}\ ())\ t)^\alpha$$

As $\langle e \rangle$ is a pure expression, we again have equal answer types α and trail types μ_α for the whole expression. The initial trail type $\bullet$ and final answer type τ of e are determined by the application $\llbracket e \rrbracket\ k_{id}\ ()$ and $k\ (\llbracket e \rrbracket\ k_{id}\ ())$, respectively. What is left is to ensure that the application of $\llbracket e \rrbracket$ to the identity continuation k_{id} is type-safe. In our type system, we use a relation $\textsf{is-id-trail}(\tau, \mu, \tau')$ to ensure this type safety. The relation holds when the type $\tau \to \mu \to \tau'$ can be assigned to the identity continuation. The valid combination of τ, μ, and τ' is derived from the definition of the identity continuation, repeated below:

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

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

When t is a non-empty trail k of type $\tau_1 \to \mu \to \tau_1{}'$, the return value of k_{id} is the result of the application $k\ v\ ()$, which has type $\tau_1{}'$. Since the expected return type of k_{id} is τ', 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 is-id-trail as an encoding of these constraints, and in the rule (Prompt), we use $\textsf{is-id-trail}(\beta, \mu_{id}, \beta')$ to constrain the type of the continuation of e. Now, it is statically guaranteed that e can be safely evaluated in an empty context.

Syntax of Types

$$\tau, \alpha, \beta ::= \iota \mid \tau \to \tau \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\beta \rangle \, \beta \qquad \text{Expression Types}$$

$$\mu, \mu_\alpha, \mu_\beta ::= \bullet \mid \tau \to \langle \mu \rangle \, \tau \qquad \text{Trail Types}$$

Typing Rules $\boxed{\Gamma \vdash e : \tau \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\beta \rangle \, \beta}$

$$\frac{c : \iota \in \Sigma}{\Gamma \vdash c : \iota \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\alpha \rangle \, \alpha} \ (\textsc{Const}) \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\alpha \rangle \, \alpha} \ (\textsc{Var})$$

$$\frac{\Gamma, \, x : \tau_1 \vdash e : \tau_2 \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\beta \rangle \, \beta}{\Gamma \vdash \lambda x. \, e : (\tau_1 \to \tau_2 \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\beta \rangle \, \beta) \, \langle \mu_\gamma \rangle \, \gamma \, \langle \mu_\gamma \rangle \, \gamma} \ (\textsc{Abs})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (\tau_1 \to \tau_2 \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\beta \rangle \, \beta) \, \langle \mu_\gamma \rangle \, \gamma \, \langle \mu_\delta \rangle \, \delta \\ \Gamma \vdash e_2 : \tau_1 \, \langle \mu_\beta \rangle \, \beta \, \langle \mu_\gamma \rangle \, \gamma\end{array}}{\Gamma \vdash e_1 \ e_2 : \tau_2 \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\delta \rangle \, \delta} \ (\textsc{App})$$

$$\frac{\begin{array}{c}\Gamma, \, k : \tau \to \tau_1 \, \langle \mu_1 \rangle \, {\tau_1}' \, \langle \mu_2 \rangle \, \alpha \vdash e : \gamma \, \langle \mu_{id} \rangle \, \gamma' \, \langle \bullet \rangle \, \beta \\ \mathsf{is\text{-}id\text{-}trail}(\gamma, \mu_{id}, \gamma') \\ \mathsf{compatible}((\tau_1 \to \langle \mu_1 \rangle \, {\tau_1}'), \mu_2, \mu_0) \\ \mathsf{compatible}(\mu_\beta, \mu_0, \mu_\alpha)\end{array}}{\Gamma \vdash \mathcal{F} k. \, e : \tau \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\beta \rangle \, \beta} \ (\textsc{Control}) \qquad \frac{\begin{array}{c}\Gamma \vdash e : \beta \, \langle \mu_{id} \rangle \, \beta' \, \langle \bullet \rangle \, \tau \\ \mathsf{is\text{-}id\text{-}trail}(\beta, \mu_{id}, \beta')\end{array}}{\Gamma \vdash \langle e \rangle : \tau \, \langle \mu_\alpha \rangle \, \alpha \, \langle \mu_\alpha \rangle \, \alpha} \ (\textsc{Prompt})$$

Auxiliary Relations

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

$$\begin{aligned}
\mathsf{compatible}(\bullet, \mu_2, \mu_3) &= \mu_2 \equiv \mu_3 \\
&\quad \text{(first branch of @ in Figure 3)} \\
\mathsf{compatible}(\mu_1, \bullet, \mu_3) &= \mu_1 \equiv \mu_3 \\
&\quad \text{(first branch of :: in Figure 3)} \\
\mathsf{compatible}((\tau_1 \to \langle \mu_1 \rangle \, \tau_1'), \mu_2, \bullet) &= \bot \\
&\quad \text{(no counterpart in Figure 3)} \\
\mathsf{compatible}((\tau_1 \to \langle \mu_1 \rangle \, \tau_1'), \mu_2, (\tau_3 \to \langle \mu_3 \rangle \, \tau_3')) &= (\tau_1 \equiv \tau_3) \land (\tau_1' \equiv \tau_3') \land (\mathsf{compatible}(\mu_2, \mu_3, \mu_1)) \\
&\quad \text{(second branch of :: in Figure 3)}
\end{aligned}$$

Figure 4 Type System of $\lambda_{\mathcal{F}}$. We assume a global signature Σ mapping constants to base types.

Control. Lastly, we apply the same method to `control`. Here is the annotated CPS translation:

$$\lambda k^{\tau \to \mu_\alpha \to \alpha}.\, \lambda t^{\mu_\beta}.\, [\![e]\!]^{(\gamma \to \mu_{id} \to \gamma') \to \bullet \to \beta}\, [\lambda x^{\tau}.\, \lambda k'^{\tau_1 \to \mu_1 \to \tau_1'}.\, \lambda t'^{\mu_2}.\, k\ x\ (t \,@\, (k' :: t')^{\mu_0})/c]\ k_{id}\ ()$$

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 γ, μ_{id}, and γ' 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 $\mathsf{compatible}(\mu_1, \mu_2, \mu_3)$ to ensure this type safety. The relation holds when composing a context of type μ_1 and another context of type μ_2 results in a context of type μ_3. Intuitively, the relation can be thought of as an addition over trail types, and the valid combination of μ_1, μ_2, and μ_3 is derived from the definition of the @ and :: functions.

$$\begin{aligned} t^{\mu_1} \,@\, t'^{\mu_2} &= \texttt{case}\ t\ \texttt{of}\ () \Rightarrow t'^{\mu_3} \mid k \Rightarrow (k :: t')^{\mu_3} \\ k^{\tau_1 \to \mu_1 \to \tau_1'} :: t^{\mu_2} &= \texttt{case}\ t\ \texttt{of}\ () \Rightarrow k^{\tau_3 \to \mu_3 \to \tau_3'} \mid k' \Rightarrow (\lambda v.\, \lambda t'.\, k\ v\ (k' :: t'))^{\tau_3 \to \mu_3 \to \tau_3'} \end{aligned}$$

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 τ_1'. Since the expected return type of :: is τ_3', 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 μ_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 μ_1 and μ_2 is an arrow type, μ_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 $\mathsf{compatible}((\tau_1 \to \langle\mu_1\rangle\ \tau_1'), \mu_2, \mu_0)$ states that consing k' to t' is type-safe, and the result has type μ_0. The second one $\mathsf{compatible}(\mu_\beta, \mu_0, \mu_\alpha)$ states that appending t to $k' :: t'$ is type-safe, and the result has type μ_α, 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:

$$\frac{\Gamma, k : \tau \to \rho, \rho, \alpha/\rho \vdash e : \gamma, \gamma, \beta/\gamma}{\Gamma \vdash \mathcal{F}k.\, e : \tau, \alpha, \beta/\rho}\ (\textsc{Control}) \qquad \frac{\Gamma \vdash e : \rho, \rho, \tau/\rho}{\Gamma \vdash \langle e\rangle : \tau, \alpha, \alpha/\sigma}\ (\textsc{Prompt})$$

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 ρ 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 $\lambda_{\mathcal{F}}$[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:

[5] Our online artifact includes an Agda implementation of this example (`exp4` in `lambdaf.agda`).

$$\vdash \mathcal{F}k_1.\,\texttt{is0}\ (k_1\ 5) : \texttt{int}\,\langle\mu_1\rangle\,\texttt{string}\,\langle\bullet\rangle\,\texttt{string}$$

$$\vdash \mathcal{F}k_2.\,\texttt{b2s}\ (k_2\ 8) : \texttt{int}\,\langle\mu_2\rangle\,\texttt{string}\,\langle\mu_1\rangle\,\texttt{string}$$

For brevity, we write μ_1 to mean $\texttt{int} \to \langle\texttt{bool} \to \langle\bullet\rangle\ \texttt{string}\rangle\ \texttt{string}$, and μ_2 to mean $\texttt{int} \to \langle\bullet\rangle\ \texttt{string}$. We can see how the trail type changes from empty ($\bullet$), to one that refers to a future context (μ_1), and to one that mentions no further context (μ_2). In particular, μ_2 is the result of "adding" μ_1 and the type of `b2s` [.]; that is, the invocation of k_2 *discharges* the future context awaited by `is0` [.]. The trail type μ_2 serves as the final trail type of the body of the enclosing `prompt`, and as it allows us to establish the is-id-trail relation required by (Prompt), we can conclude that the whole program is well-typed.

6 Properties

The type system of $\lambda_{\mathcal{F}}$ 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 `Reduce` relation in `lambdaf-red.agda`). Note that, to prove type preservation of the `control` reduction (rule ($\mathcal{F}$) 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 $F[\mathcal{F}k.\,e']$.*

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 `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 $\lambda_{\mathcal{F}}$ expression into a well-typed λ_C expression. To establish this theorem, we define the type system of λ_C (Figure 5) and a CPS translation * on $\lambda_{\mathcal{F}}$ types (Figure 6).

Let us elaborate on rule (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}, @, and ::. Unlike the standard typing rule for case analysis, rule (Case) type-checks the two branches using equality assumptions $\mu \equiv \bullet$ and

Syntax of Types

$$\tau = \iota \mid \tau \to \tau \mid \bullet$$

Typing Rules

$$\frac{c : \iota \in \Sigma}{\Gamma \vdash x : \iota}\ (\textsc{Const}) \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (\textsc{Var}) \qquad \frac{\Gamma,\, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2}\ (\textsc{Abs})$$

$$\frac{}{\Gamma \vdash () : \bullet}\ (\textsc{Unit}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ (\textsc{App})$$

$$\frac{\begin{array}{c}\Gamma \vdash t : \mu^* \quad \Gamma,\, \mu \equiv \bullet \vdash e_1 : \tau \\ \forall \tau_1, \mu_1, {\tau_1}'.\ \Gamma,\, k : \mu^*,\, \mu \equiv \tau_1 \to \langle \mu_1 \rangle\ {\tau_1}' \vdash e_2 : \tau\end{array}}{\Gamma \vdash \mathtt{case}\ t\ \mathtt{of}\ () \Rightarrow e_1 \mid k \Rightarrow e_2 : \tau}\ (\textsc{Case})$$

Figure 5 Type System of λ_C. We assume a global signature Σ mapping constants to base types.

$\mu \equiv \tau_1 \to \langle \mu_1 \rangle\ {\tau_1}'$[6]. 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}:

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

In the first branch, we see an inconsistency between the expected return type τ' and the actual return type τ. However, by the typing rules defined in Figure 4, we know that k_{id} is used only when the relation is-id-trail(τ, μ, τ') 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 $\mu \equiv \tau_1 \to \langle \mu_1 \rangle\ \tau_1'$ to derive $\tau \equiv \tau_1$, $\tau' \equiv \tau_1'$, 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.

▶ **Theorem 4** (Type Preservation of CPS Translation)**.** *If* $\Gamma \vdash e : \tau\, \langle \mu_\alpha \rangle\, \alpha\, \langle \mu_\beta \rangle\, \beta$ *in* $\lambda_{\mathcal{F}}$*, then* $\Gamma^* \vdash [\![e]\!] : (\tau^* \to \mu_\alpha^* \to \alpha^*) \to \mu_\beta^* \to \beta^*$ *in* λ_C*.*

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.

[6] 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.

$\boxed{\text{Translation of Expression Types}}$

$$\iota^* = \iota$$
$$(\tau_1 \to \tau_2 \langle \mu_\alpha \rangle \alpha \langle \mu_\beta \rangle \beta)^* = \tau_1^* \to (\tau_2^* \to \mu_\alpha^* \to \alpha^*) \to \mu_\beta^* \to \beta^*$$

$\boxed{\text{Translation of Trail Types}}$

$$\bullet^* = \bullet$$
$$(\tau \to \langle \mu \rangle \tau')^* = \tau^* \to \mu^* \to \tau'^*$$

Figure 6 CPS Translation of $\lambda_{\mathcal{F}}$ Types.

▶ **Theorem 5** (Termination). *If $\Gamma \vdash e : \tau \langle \bullet \rangle \alpha \langle \bullet \rangle \alpha$, then there exists some value v such that $e \rightsquigarrow^* v$, where $\rightsquigarrow^*$ is the reflexive, transitive closure of $\rightsquigarrow$ defined in Figure 1.*

Proof. The statement is witnessed by a CPS interpreter of $\lambda_{\mathcal{F}}$ implemented in Agda (the `go` function in `lambdaf.agda`). Since every well-typed Agda program terminates, and since our interpreter is judged well-typed, we know that evaluation of $\lambda_{\mathcal{F}}$ 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:

$$\begin{aligned}
&\langle (\mathcal{F}k_1.\, k_1\ 1; k_1\ 1); (\mathcal{F}k_2.\, k_2\ 1; k_2\ 1)\rangle \\
&= \langle k_1\ 1; k_1\ 1\, [\lambda x.\, x; (\mathcal{F}k_2.\, k_2\ 1; k_2\ 1)/k_1]\rangle \\
&= \langle (\mathcal{F}k_2.\, k_2\ 1; k_2\ 1); ((\lambda x.\, x; (\mathcal{F}k_2.\, k_2\ 1; k_2\ 1))\ 1)\rangle \\
&= \langle k_2\ 1; k_2\ 1\, [\lambda y.\, y; (\lambda x.\, x; (\mathcal{F}k_2.\, k_2\ 1; k_2\ 1))\ 1/k_2]\rangle \\
&= \langle (\mathcal{F}k_2.\, k_2\ 1; k_2\ 1); (\lambda y.\, y; (\lambda x.\, x; (\mathcal{F}k_2.\, k_2\ 1; k_2\ 1))\ 1)\ 1\rangle \\
&= \ldots
\end{aligned}$$

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, `shift` and `reset` [15] are called *static*, as the extent of a captured continuation can always be determined from the lexical structure of the program. Other variants are all *dynamic*, since the control operator may capture the invocation contexts of previously captured continuations (as `control` does), or the meta-contexts outside of the original innermost delimiter (as `shift0` [32] does), or both kinds of contexts (as `control0` [16] does). Dynamic control operators all have a semantics that involves a trail-like structure, containing the contexts beyond the lexically enclosing one.

Type Systems for Control Operators. The CPS-based approach to designing type systems has been applied to several variants of delimited control operators, including `shift`/`reset` [14, 3], `control`/`prompt` [26], and `shift0`/`prompt0` [32]. While Danvy and Filinski [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 [3, 26, 32], and allows more efficient compilation via a selective CPS translation [37, 32, 4].

Algebraic Effects and Handlers. In the past decade, algebraic effects and handlers [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 [5, 27] existed before their CPS semantics [29, 24, 23]. Also, type-preserving CPS translation of effect handlers is an open problem in the community [23].

8 Conclusion and Future Work

In this paper, we show how to derive a general type system for the `control` and `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.

Implementation. Having designed a type system for `control` and `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 [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 `shift` and `reset` [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 (μ_0, $\tau_1 \to \langle\mu_1\rangle\ \tau_1'$, and μ_2) in the (Control) rule. Once we have done these, we will develop an implementation (possibly as an extension of OchaCaml [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

1 Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, PPDP '03, pages 8–19, New York, NY, USA, 2003. ACM. doi:10.1145/888251.888254.

2 Kenichi Asai, Youyou Cong, and Chiaki Ishio. A functional abstraction of typed trails. Short paper presented at the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2021), 2021.

3 Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In *Proceedings of the 5th Asian Conference on Programming Languages and Systems*, APLAS'07, pages 239–254, Berlin, Heidelberg, 2007. Springer-Verlag.

4 Kenichi Asai and Chihiro Uehara. Selective CPS transformation for shift and reset. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '18, pages 40–52, New York, NY, USA, December 2017. ACM. doi:10.1145/3162069.

5 Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. In Reiko Heckel and Stefan Milius, editors, *Algebra and Coalgebra in Computer Science*, pages 1–16, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

6 Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015.

7 Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1, 2005.

8 Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style for dynamic delimited continuations. *BRICS Report Series*, 13(15), 2006.

9 Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style for dynamic delimited continuations. *ACM Trans. Program. Lang. Syst.*, 38(1), 2015. `doi:10.1145/2794078`.

10 Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60(3):274–297, 2006.

11 Dariusz Biernacki, Mateusz Pyzik, and Filip Sieczkowski. A reflection on continuation-composing style. In *Proceedings of 5th International Conference on Formal Structures for Computation and Deduction*, FSCD '20. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020.

12 Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Third Workshop on Logical Frameworks*, 1992.

13 Olivier Danvy. Type-directed partial evaluation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 242–257. ACM, 1996.

14 Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. BRICS 89/12, 1989.

15 Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160. ACM, 1990.

16 R. Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *J. Funct. Program.*, 17(6):687–730, November 2007. `doi:10.1017/S0956796807006259`.

17 Matthias Felleisen. On the expressive power of programming languages. In *Selected Papers from the Symposium on 3rd European Symposium on Programming*, ESOP '90, pages 35–75, New York, NY, USA, 1991. Elsevier North-Holland, Inc.

18 Mattias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 180–190, New York, NY, USA, 1988. ACM. `doi:10.1145/73560.73576`.

19 Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 446–457, New York, NY, USA, 1994. ACM. `doi:10.1145/174675.178047`.

20 Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.*, 1(ICFP):13:1–13:29, August 2017. `doi:10.1145/3110257`.

21 Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 12–23, New York, NY, USA, 1995. ACM. `doi:10.1145/224164.224173`.

22 Daniel Hillerström and Sam Lindley. Shallow effect handlers. In *Asian Symposium on Programming Languages and Systems*, APLAS '18, pages 415–435. Springer, 2018.

23 Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *Journal of Functional Programming*, 30, 2020. `doi:10.1017/S0956796820000040`.

24 Daniel Hillerström, Sam Lindley, Robert Atkey, and KC Sivaramakrishnan. Continuation passing style for effect handlers. In *Proceedings of 2nd International Conference on Formal Structures for Computation and Deduction*, FSCD '17, pages 18:1–18:19. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

25 Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 177–188. ACM, 2003.

26 Yukiyoshi Kameyama and Takuo Yonezawa. Typed dynamic control operators for delimited continuations. In *International Symposium on Functional and Logic Programming*, FLOPS '08, pages 239–254. Springer, 2008.

27 Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 145–158, New York, NY, USA, 2013. ACM. doi:10.1145/2500365.2500590.

28 Oleg Kiselyov and K. C. Sivaramakrishnan. Eff directly in ocaml. In *ML Workshop*, 2016.

29 Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, pages 486–499, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009872.

30 Moe Masuko and Kenichi Asai. Caml Light+ shift/reset= Caml Shift. In *Theory and Practice of Delimited Continuations*, TPDC '11, pages 33–46, 2011.

31 Marek Materzok. Axiomatizing subtyped delimited continuations. In *Computer Science Logic 2013*, CSL 2013. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

32 Marek Materzok and Dariusz Biernacki. Subtyping delimited continuations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 81–93, New York, NY, USA, 2011. ACM. doi:10.1145/2034773.2034786.

33 Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

34 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

35 Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Typed equivalence of effect handlers and delimited control. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

36 Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, ESOP '09, pages 80–94. Springer, 2009.

37 Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 317–328, New York, NY, USA, 2009. ACM. doi:10.1145/1596550.1596596.

38 Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and symbolic computation*, 6(3):289–360, 1993.

39 Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM transactions on programming languages and systems (TOPLAS)*, 19(6):916–941, 1997.

40 Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 20(4):371–401, 2007.

41 Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M Essertel, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–31, 2019.

42 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

43 Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. Effect handlers, evidently. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–29, 2020. doi:10.1145/3408981.