Shifting the Blame
A Blame Calculus with Delimited Control

Taro Sekiyama, Soichiro Ueda*, and Atsushi Igarashi

Graduate School of Informatics, Kyoto University

Abstract. We study integration of static and dynamic typing in the presence of delimited-control operators. In a program where typed and untyped parts coexist, the run-time system has to monitor the flow of values between these parts and abort program execution if invalid values are passed. However, control operators, which enable us to implement useful control effects, make such monitoring tricky; in fact, it is known that, with a standard approach, certain communications between typed and untyped parts can be overlooked.
We propose a new cast-based mechanism to monitor all communications between typed and untyped parts for a language with control operators shift and reset. We extend a blame calculus with shift/reset to give its semantics (operational semantics and CPS transformation) and prove two important correctness properties of the proposed mechanism: Blame Theorem and soundness of the CPS transformation.

1 Introduction

Many programming languages support either static or dynamic typing. Static typing makes early error detection and compilation to faster code possible while dynamic typing makes flexible and rapid software development easier. To take the best of both worlds, integration of static and dynamic typing has been investigated. Indeed, several practical programming languages—e.g., C♯, TypeScript, Typed Racket [28], Typed Clojure [4], Reticulated Python [29], Hack (an extension of PHP), etc.—allow typed and untyped parts to coexist in one program and to communicate with each other.

In languages allowing such integration, *casts* [22, 14, 30] (or *contracts* [13, 27, 26]) play an important role for monitoring the flow of values between typed and untyped parts. A source program that contains typed and untyped parts is compiled to an intermediate language such that casts are inserted in points where typed and untyped code interacts. Casts are a run-time mechanism to check that a program component satisfies a given type specification. For example, when typed code imports a certain component from untyped code as integer, a cast is inserted to check that it is actually an integer at run time. If it is detected that a component did not follow the specification, an uncatchable exception, called

* Current affiliation: Works Applications Co., Ltd.

blame, will be raised to notify that something unexpected has happened. Tobin-Hochstadt and Felleisen [27] originated a *blame calculus* to study integration of static and dynamic typing and Wadler and Findler [30] refined the theory of blame on its variant.

We study integration of static and dynamic typing in the presence of delimited-control operators. As is well known, various control effects—e.g, exception handling [25], backtracking [6], monads [12], generators [25], etc.—can be expressed by using delimited continuations as first-class values. However, control operators make it tricky to monitor the borders between typed and untyped parts; in fact, as is pointed out by Takikawa, Strickland, and Tobin-Hochstadt [26], communications between the two parts via continuations captured by control operators can be overlooked under standard cast semantics.

Our contributions. In this paper, we propose a blame calculus, based on Wadler and Findler [30], with Danvy and Filinski's delimited-control operators *shift* and *reset* [6] and give a new cast-based mechanism to monitor all communications between typed and untyped parts. The idea of the new cast comes from Danvy and Filinski's type system [5] for shift/reset, where type information about contexts is considered. Using types of contexts, our cast mechanism can monitor all communications.

As a proof of correctness of our idea, we investigate two important properties. One is Blame Theorem [27, 30], which states that values that flow from typed code never trigger run-time type errors. The other property is soundness of CPS transformation: it preserves well-typedness and, for any two source terms such that one reduces to the other, their transformation results are equivalent in the target calculus. It turns out that we need a few axioms about casts in addition to usual axioms, such as (call-by-value) β-reduction, for equality in the target calculus.

The organization of the paper. In Section 2, we review the blame calculus and the control operators shift/reset, explain why the standard cast does not work when they are naively combined, and briefly describe our solution. Section 3 formalizes our calculus with an operational semantics and a type system, and shows type soundness of the calculus. Section 4 shows a Blame Theorem in our calculus and Section 5 introduces CPS transformation and shows its soundness. Finally, discussing related work in Section 6, we conclude in Section 7. We omit proofs from the paper; interested readers are referred to a full version of the paper available at `http://www.fos.kuis.kyoto-u.ac.jp/~t-sekiym/papers/decon_blame/APLAS2015_decon_blame_full.pdf`.

2 Blame Calculus with Shift and Reset

2.1 Blame Calculus

The blame calculus of Wadler and Findler [30] is a kind of typed lambda calculus for studying integration of static and dynamic typing. It is designed as an

intermediate language for gradually typed languages [22], where a program at an early stage is written in an untyped language and parts whose specifications are stable can be gradually rewritten in a typed language, resulting in a program with both typed and untyped parts. In blame calculi, untyped parts are represented as terms of the special, *dynamic type* (denoted by $\star$), where any operation is statically allowed at the risk of causing run-time errors. Blame calculi support smooth interaction between typed and untyped parts—i.e., typed code can use an untyped component and vice versa—via a type-directed mechanism, *casts*.

A cast, taking the form $t : A \Rightarrow^p B$, checks that term t of source type A behaves as target type B at run time; p, called a *blame label*, is used to identify the cast that has failed at run time. For example, using integer type int, cast expression $1 : \mathsf{int} \Rightarrow^p \star$ injects integer 1 to the dynamic type; conversely, $t : \star \Rightarrow^p \mathsf{int}$ coerces untyped term t to int. A cast would fail if the coerced value cannot behave as the target type of the cast. For example, cast expression $(1 : \mathsf{int} \Rightarrow^{p_1} \star) : \star \Rightarrow^{p_2} \mathsf{bool}$, which coerces integer 1 to the dynamic type and then its result to Boolean type bool, causes blame $\mathsf{blame}\, p_2$ at run time since the coerced value 1 cannot behave as bool.

Using casts, in addition to fully typed and fully untyped programs, we can write a program where typed and untyped parts are mixed. For example, suppose that we first write an untyped program as follows:

$$\mathsf{let}\ \mathit{succ} = \lambda x.\, x + 1\ \mathsf{in}\ \mathit{succ}\, 1$$

where we color untyped parts gray.[1] If the successor function is statically typed, we rewrite the program so that it imports the typed successor function:

$$\mathsf{let}\ \mathit{succ} = (\lambda x.\, x + 1) : \mathsf{int} \to \mathsf{int} \Rightarrow^p \star\ \mathsf{in}\ \mathit{succ}\, 1$$

where we color typed parts white. When the source and target types in a cast are not important, as is often the case, we just surround a term by a frame to indicate the existence of some appropriate cast. So, the program above is presented as below:

$$\mathsf{let}\ \mathit{succ} = \boxed{\lambda x.\, x + 1}\ \mathsf{in}\ \mathit{succ}\, 1$$

Intuitively, a frame in programs in this style means that flows of values between the typed and untyped parts are monitored by casts. Conversely, the absence of a frame between the two parts indicates that the run-time system will overlook their communications.

What happens when a value is coerced to the dynamic type rests on the source type of the cast. If it is a first-order type such as int, the cast simply tags the value with its type. If it is a function type, by contrast, the cast generates a lambda abstraction that wraps the target function and then tags the wrapper. The wrapper, a function over values of the dynamic type, checks, by using a cast, that a given argument has the type expected by the wrapped function and coerces the return value of the wrapped function to the dynamic type, similarly

[1] Precisely speaking, even untyped programs need casts to use values of the dynamic type as functions, integers, etc., but we omit them to avoid the clutter.

to function contracts [13]. For example, cast expression $(\lambda x{:}\mathsf{int}.\, x + 1) : \mathsf{int} \to \mathsf{int} \Rightarrow^p \star$ generates lambda abstraction $\lambda y{:}\star.\, (((\lambda x{:}\mathsf{int}.\, x + 1)\,(y : \star \Rightarrow^q \mathsf{int})) : \mathsf{int} \Rightarrow^p \star)$. Here, blame label q is the negation of p, which we will discuss in detail below. Using the notation introduced above, it is easy to understand that all communications between typed and untyped parts are monitored because the program above reduces to:

$$\mathsf{let}\ \mathit{succ} = \lambda y.\, \boxed{(\lambda x.\, x + 1)\, \boxed{y}}\ \mathsf{in}\ \mathit{succ}\, 1$$

As advocated by Findler and Felleisen [13], there are two kinds of blame—positive blame and negative blame, which indicate that, when a cast fails, its responsibility lies with the term contained in the cast and the context containing the cast, respectively. Following Wadler and Findler, we introduce an involutive operation $\bar{\cdot}$ of negation on blame labels: for any blame label p, $\bar{p}$ is its negation and $\bar{\bar{p}}$ is the same as p. For a cast with blame label p in a program, $\mathsf{blame}\, p$ and $\mathsf{blame}\, \bar{p}$ denotes positive blame and negative blame, respectively. A key observation, so-called the *Blame Theorem*, in work on blame calculi is that a cast failure is never caused by values from the more precisely typed side in the cast—i.e., if the side of a term contained in a cast with p is more precisely typed, a program including the cast never evaluates to $\mathsf{blame}\, p$, while if the side of a context containing the cast is, the program never evaluates to $\mathsf{blame}\, \bar{p}$.

2.2 Delimited-Control Operators: Shift and Reset

Shift and *reset* are delimited-control operators introduced by Danvy and Filinski [6]. Shift captures the current continuation, like another control operator call/cc, and reset delimits the continuation captured by shift. The captured continuation works as if it is a composable function, namely, unlike call/cc, control is returned to a caller when the call to the captured continuation finishes.

As an example with shift and reset, let us consider the following program:

$$\langle 5 + \mathcal{S}k.\, ((k\, 1 + k\, 2) = 13)\rangle$$

Here, the shift operator is invoked by the subterm $\mathcal{S}k.\, ((k\, 1 + k\, 2) = 13)$ and the reset operator $\langle ... \rangle$ encloses the whole term. To evaluate a reset operator, we evaluate its body. Evaluation of the shift operator $\mathcal{S}k.\, ((k\, 1 + k\, 2) = 13)$ proceeds as follows. First, it captures the continuation up to the closest reset as a function. Since the delimited continuation in this program is $5 + [\,]$ (here, $[\,]$ means a hole of the context), the captured continuation takes the form $\lambda x.\, \langle 5 + x\rangle$ (note that the body of the function is enclosed by reset). Next, variable k is bound to the captured continuation. Finally, the body of the closest reset operator is replaced with the body of the shift operator. Thus, the example program reduces to:

$$\langle (((\lambda x.\, \langle 5 + x\rangle)\, 1) + ((\lambda x.\, \langle 5 + x\rangle)\, 2)) = 13\rangle.$$

Since reset returns the result of its body, it evaluates to true.

Let us consider a more interesting example of function *choice*, a user of which passes a tuple of integers and expects to return one of them. The caller tests the returned integer by some Boolean expression and surrounds it by reset. Then, the whole reset expression evaluates to the index (tagged with Some) to indicate which integer satisfied the test, or None to indicate none of them satisfied. For example, $\langle \texttt{prime?}\,(\mathit{choice}\,(141, 197))\rangle$ will evaluate to $\mathsf{Some}\,2$ because the second argument 197 is a prime number. Using shift/reset, such a (two-argument version of) *choice* function can be defined as follows:

$$\mathit{choice} = \lambda(x, y){:}\mathsf{int} \times \mathsf{int}.\, \mathcal{S}k.\, \mathsf{if}\ k\ x\ \mathsf{then}\ \mathsf{Some}\,1\ \mathsf{else\ if}\ k\ y\ \mathsf{then}\ \mathsf{Some}\,2\ \mathsf{else}\ \mathsf{None}$$

It is important to observe k is bound to the predicate (in this case, $\lambda z.\, \langle \texttt{prime?}\ z\rangle$).

Since blame calculi support type-directed casts, it is crucial to consider type discipline in the presence of shift/reset. This work adopts the type system proposed by Danvy and Filinski [5]. Their type system introduces types, called *answer types*, of contexts up to the closest reset to track modification of the body of a reset operator—we have seen above that the body of a reset operator can be modified to the body of a shift operator at run time. In the type system, using metavariables α and β for types, function types take the form $A/\alpha \to B/\beta$, which means that a function of this type is one from A to B and, when applied, it modifies the answer type α to β. For example, using a function of type $(\mathsf{int} \times \mathsf{int})/\mathsf{bool} \to \mathsf{int}/(\mathsf{int\,option})$ ($\mathsf{int\,option}$ means integers tagged with Some and None), its user, when passing a pair of integers, expects to return an integer value and to modify the answer type bool to $\mathsf{int\,option}$. Conversely, to see how functions are given such a function type, let us consider *choice*, which is typed at $(\mathsf{int} \times \mathsf{int})/\mathsf{bool} \to \mathsf{int}/(\mathsf{int\,option})$. It can be found from the type annotation that it takes pairs of integers. The body captures a continuation and calls it with the first and second components of the argument pair. Since a caller of *choice* obtains a value passed to the continuation k, the return type is int. *choice* demands the answer type of a context be bool because the captured continuation is required to return a Boolean value in conditional expressions; and the shift operator modifies the answer type to $\mathsf{int\,option}$ because the if-expression returns an $\mathsf{int\,option}$ value.

2.3 Blame Calculus with Shift and Reset

We extend the blame calculus with shift/reset so that all value flows between typed and untyped parts are monitored, following the type discipline discussed above. The main question here is how we should give the semantics of casts for function types, which now include answer type information. The standard semantics discussed above does not suffice because it is ignorant of answer types. In fact, it would fail to monitor value flows that occur due to manipulation of delimited continuations, as we see below. For example, let us consider the situation that untyped code imports typed function *choice* via a cast (represented by a frame):

$$\mathsf{let}\ f = \boxed{\mathit{choice}}\ \mathsf{in}\ 5 + \langle \mathit{succ}\,(f\,(141, 197))\rangle$$

This program contains two errors: first, subterm $succ\,(f\,(141, 197))$ within reset returns an integer, though shift in *choice* expects it to return a Boolean value since the continuation captured by the shift operator is used in conditional expressions; second, as found in subterm $5 + \langle \ldots \rangle$, the computation result of reset is expected to be an integer, though it should be an $\mathsf{int\ option}$ value coming from the body of shift in *choice*. However, if the cast on *choice* behaved as a standard function cast we discussed in Section 2.1, these errors would not be detected at run time on borders between typed and untyped parts. To see the reason, let us reduce the program. First, since the choice is coerced to the dynamic type, a wrapper that checks an argument and the return value is generated and then is applied to $(141, 197)$:

$$\mathsf{let}\ f = \boxed{choice}\ \mathsf{in}\ 5 + \langle succ\,(f\,(141, 197))\rangle \longmapsto^* 5 + \langle succ\,(\boxed{choice\ \boxed{(141, 197)}})\rangle$$

The check for $(141, 197)$ succeeds and so *choice* is applied to $(141, 197)$, and then the shift operator in *choice* is invoked.

$$\begin{aligned}\cdots &\longmapsto^* 5 + \langle succ\,\boxed{\mathcal{S}k.\ \mathsf{if}\ k\ 141\ \mathsf{then\ Some}\ 1\ \mathsf{else\ if}\ k\ 197\ \mathsf{then\ Some}\ 2\ \mathsf{else\ None}}\rangle \\ &\longmapsto^* 5 + \langle \mathsf{if}\ (\lambda x.\ \langle succ\ \boxed{x}\rangle)\ 141\ \mathsf{then\ Some}\ 1\ \mathsf{else\ if}\ \ldots\ \mathsf{then\ Some}\ 2\ \mathsf{else\ None}\rangle\end{aligned}$$

Here, there are one gray area and one white area, both without surrounding frames. The former means that the value flow from the captured continuation $\lambda x.\ \langle succ\ \boxed{x}\rangle$ to typed code will not be monitored, when it should be by the cast from the dynamic type to bool. Similarly, the latter means that the value flow from the result of the (typed) if-expression to untyped code will not be monitored, either, when it should be by the cast from $\mathsf{int\ option}$ to the dynamic type. The problem is that the standard function casts can monitor calls of functions but does not capture and calls of delimited continuations.

Our cast mechanism can monitor such capture and calls of delimited continuations. A wrapper generated by a cast from $A/\alpha \to B/\beta$ to the dynamic type, when applied, ensures that the reset expression enclosing the application returns a value of the dynamic type by inserting injection from β and that the continuation captured during the call to the wrapped function returns a value of α by the cast to α. In the above example of *choice*, our cast mechanism reduces the original program to a term like:

$$5 + \langle \boxed{\mathsf{if}\ (\lambda x.\ \boxed{\langle succ\ \boxed{x}\rangle})\ 141\ \mathsf{then\ Some}\ 1\ \mathsf{else\ if}\ \ldots\ \mathsf{then\ Some}\ 2\ \mathsf{else\ None}}\rangle$$

where two casts are added: one to check that the return value of the continuation has bool and the other to inject the result of the if-expression to the dynamic type.

3 Language

In this section, we formally define a call-by-value blame calculus with delimited-control operators shift and reset and show its type soundness. Our calculus is a variant of the blame calculus by Ahmed et al. [1].

$$
\begin{array}{llll}
\textbf{variables} & x, y, k & \textbf{blame labels} \quad p, q & \textbf{constants} \quad c \qquad \textbf{base types} \quad \iota \\
\textbf{ground types} & G, H & ::= \iota \mid \star/\star \to \star/\star & \\
\textbf{types} & A, B, \alpha, \beta, \gamma, \delta & ::= \iota \mid \star \mid A/\alpha \to B/\beta & \\
\textbf{values} & v & ::= x \mid c \mid \lambda x.\, t \mid v \,:\, G \Rightarrow \star & \\
\textbf{terms} & s, t, u & ::= x \mid c \mid op(\overline{t_i}^{\,i}) \mid \lambda x.\, t \mid s\ t \mid & \\
 & & \quad s \,:\, A \Rightarrow^{p}\ B \mid s \,:\, G \Rightarrow \star \mid \mathsf{blame}\, p \mid \langle s \rangle \mid \mathcal{S}k.\, s &
\end{array}
$$

Fig. 1. Syntax.

3.1 Syntax

Figure 1 presents the syntax, which is parameterized over base types, denoted by ι, constants, denoted by c, and primitive operations, denoted by op, over constants.

Types consist of base types, the dynamic type, and function types with answer types. Unlike the blame calculus of Wadler and Findler, our calculus does not include refinement types (a.k.a., subset types) for simplicity; we believe that it is not hard to add refinement types if refinements are restricted to be pure [2]. Ground types, denoted by G and H, classify kinds of values. If the ground type is a base type, the values are constants of the base type, and if it is a function type (constituted only of the dynamic type), the values are lambda abstractions.

Values, denoted by v, consist of variables, constants, lambda abstractions, and ground values. A lambda abstraction $\lambda x.\, t$ is standard; variable x is bound in the body t. A ground value $v \,:\, G \Rightarrow \star$ is a value of the dynamic type; the kind of v follows ground type G.

Terms, denoted by s and t, extend those in the simply typed blame calculus with two forms, reset expressions and shift expressions. Using the notation $\overline{t_i}^{\,i}$ to denote a sequence $t_1, ..., t_n$ of terms, we allow primitive operators to take tuples of terms. A reset expression is written as $\langle s \rangle$ and a shift expression is as $\mathcal{S}k.\, s$ where k is bound in the body s. The syntax includes blame as a primitive construct despite the fact that exceptions can be implemented by shift and reset because blame is an *uncatchable* exception in a blame calculus. Note that ground values, ground terms ($s \,:\, G \Rightarrow \star$), and blame are supposed to be "run-time" citizens that appear only during reduction and not in a source program.

In what follows, as usual, we write $s\,[x := v]$ for capture-avoiding substitution of v for variable x in s. As shorthand, we write $s \,:\, G \Rightarrow \star \Rightarrow^{p}\ A$ and $s \,:\, A \Rightarrow^{p}\ G \Rightarrow \star$ for $(s \,:\, G \Rightarrow \star) \,:\, \star \Rightarrow^{p}\ A$ and $(s \,:\, A \Rightarrow^{p}\ G) \,:\, G \Rightarrow \star$, respectively.

3.2 Semantics

The semantics of our calculus is given in a small-step style by using two relations over terms: reduction relation $\longrightarrow$, which represents basic computation such as β-reduction, and evaluation relation $\longmapsto$, in which subterms are reduced.

$\boxed{s \longrightarrow t}$ **Reduction rules**

$$
\begin{array}{lll}
op(\overline{v_i}^{\,i}) & \longrightarrow \zeta\,(op, \overline{v_i}^{\,i}) & \text{R\_Op} \\
(\lambda x.\, s)\, v & \longrightarrow s\,[x := v] & \text{R\_Beta} \\
\langle v \rangle & \longrightarrow v & \text{R\_Reset} \\
\langle F[\mathcal{S}k.\, s] \rangle & \longrightarrow \langle s\,[k := \lambda x.\, \langle F[x] \rangle] \rangle \text{ where } x \notin fv\,(F) & \text{R\_Shift} \\
v : \iota \Rightarrow^p \iota & \longrightarrow v & \text{R\_Base} \\
v : \star \Rightarrow^p \star & \longrightarrow v & \text{R\_Dyn} \\
v : A/\alpha \to B/\beta \Rightarrow^p A'/\alpha' \to B'/\beta' & \longrightarrow & \\
\quad \lambda x.\, \mathcal{S}k.\, (\langle (k\,((v\,(x : A' \Rightarrow^{\bar{p}} A)) : B \Rightarrow^p B')) : \alpha' \Rightarrow^{\bar{p}} \alpha \rangle : \beta \Rightarrow^p \beta') & & \text{R\_Wrap} \\
v : A \Rightarrow^p \star & \longrightarrow v : A \Rightarrow^p G \Rightarrow \star \quad \text{if } A \sim G \text{ and } A \neq \star & \text{R\_Ground} \\
v : G \Rightarrow \star \Rightarrow^p A & \longrightarrow v : G \Rightarrow^p A \quad \text{if } G \sim A \text{ and } A \neq \star & \text{R\_Collapse} \\
v : G \Rightarrow \star \Rightarrow^p A & \longrightarrow \mathsf{blame}\, p \quad \text{if } G \not\sim A & \text{R\_Conflict}
\end{array}
$$

$\boxed{s \longmapsto t}$ **Evaluation rules**

$$\frac{s \longrightarrow t}{E[s] \longmapsto E[t]}\ \text{E\_Step} \qquad \frac{E \neq [\,]}{E[\mathsf{blame}\, p] \longmapsto \mathsf{blame}\, p}\ \text{E\_Abort}$$

Fig. 2. Reduction and evaluation.

The reduction rules, shown at the top of Figure 2, are standard or similar to the previous calculi except (R_Wrap), which is the key of our work. In (R_Op), to reduce a call to a primitive operator, we assume that there is a function ζ which returns an appropriate value when taking an operator name and arguments to it. The rule (R_Shift) presents that the shift operator captures the continuation up to the closest reset operator. In the rule, the captured continuation is represented by pure evaluation contexts, denoted by F, which are evaluation contexts [11] where the hole does not occur in bodies of reset operators. Pure evaluation contexts are defined as follows:

$$F ::= [\,] \mid op(\overline{v_i}^{\,i}, F, \overline{t_j}^{\,j}) \mid F\ s \mid v\ F \mid F : A \Rightarrow^p B \mid F : G \Rightarrow \star$$

As mentioned earlier, the body of the function representing the captured continuation is enclosed by reset.

There are six reduction rules for cast expressions. The rules (R_Base) and (R_Dyn) mean that casts between the same base type and between the dynamic type perform no checks. We find (R_Dyn), which does not appear in Ahmed et al. [1], matches well with CPS transformation. The rule (R_Ground), applied when the target type is the dynamic type but the source type is not, turns a cast expression to a ground term by inserting a cast to the ground type G that represents the kind of the value v. The relation $\sim$, called compatibility, over two types is defined as the least compatible relation closed under $A \sim \star$ and $\star \sim B$. It intuitively means that a cast from A to B (and vice versa) can succeed; in other words, $A \not\sim B$ means that a cast will fail. One interesting fact about

compatibility is that, for any nondynamic type A, we can find exactly one ground type that is compatible with A: If A is a base type, then G is equal to A and, if A is a function type, then G is $\star/\star \to \star/\star$. As a result, G in (R_Ground) is uniquely determined. The rules (R_Collapse) and (R_Conflict) are applied when a target value is a ground value. When the kind G of the underlying value v is not compatible with the target type of the cast, the cast is blamed with blame label p by (R_Conflict). Otherwise, the underlying value is coerced from the ground type of the ground value to the target type of the cast by (R_Collapse).

The reduction rule (R_Wrap), applied to casts between function types, is the most involved. The rule means that the cast expression reduces to a lambda abstraction that wraps the target value v. Since the wrapper function works as a value of type $A'/\alpha' \to B'/\beta'$, it takes a value of A'. Like function contracts [13], in the wrapper, the argument denoted by x is coerced to argument type A of the source type to apply v to it and the return value of v is coerced to return type B' of the target type. Furthermore, to call the target function in a context of answer type α, the wrapper captures the continuation in which the wrapper is applied by using shift, applies the captured continuation to the result of the target function, and then coerces the result of the captured continuation to α. Since the wrapper is applied in a context of answer type α', the captured continuation returns a value of α'. By enclosing the cast to α with reset, a continuation captured during the call to v returns a value of α. Finally, the wrapper coerces the result of the reset operator from β to β' because the call to the target function modifies the answer type of the context to β, and so the reset expression returns a value of β, and the wrapper is expected to modify the answer type to β'. The rule (R_Wrap) reverses blame labels for casts from A' to A and from α' to α because target values for those casts originate from the context side.

We illustrate how (R_Wrap) makes monitoring of capture and calls of continuations possible, using *choice* in Section 2.3. By (R_Ground), the cast from $(\mathsf{int} \times \mathsf{int})/\mathsf{bool} \to \mathsf{int}/(\mathsf{int\ option})$ to the dynamic type reduces to that to $\star/\star \to \star/\star$. By (R_Wrap), the cast generates a wrapper.

$$\begin{aligned} & \mathsf{let}\ f = \boxed{\mathit{choice}}\ \mathsf{in}\ 5 + \langle \mathit{succ}\,(f\,(141, 197))\rangle \\ \longmapsto\ & \mathsf{let}\ f = \lambda x.\,\mathcal{S}k'.\boxed{\langle \boxed{k'\ \boxed{(\mathit{choice}\ \boxed{x})}}\rangle}\ \mathsf{in}\ 5 + \langle \mathit{succ}\,(f\,(141, 197))\rangle \end{aligned}$$

The wrapper is applied to $(141, 197)$, so the evaluation proceeds as follows:

$$\begin{aligned} \cdots \longmapsto^*\ & 5 + \langle \mathit{succ}\,(\mathcal{S}k'.\boxed{\langle \boxed{k'\ \boxed{(\mathit{choice}\ \boxed{(141, 197)})}}\rangle})\rangle \\ \longmapsto\ & 5 + \boxed{\langle \boxed{(\lambda x.\,\langle \mathit{succ}\,x\rangle)\ \boxed{(\mathit{choice}\ \boxed{(141, 197)})}}\rangle}\rangle \\ \longmapsto^*\ & 5 + \boxed{\langle \boxed{(\lambda x.\,\langle \mathit{succ}\,x\rangle)\ \boxed{(\mathcal{S}k.\ \mathsf{if}\ k\ 141\ \mathsf{then\ Some}\ 1\ \mathsf{else}\ \ldots)}}\rangle}\rangle \\ \longmapsto\ & 5 + \boxed{\langle \boxed{\langle \mathsf{if}\ v\ 141\ \mathsf{then\ Some}\ 1\ \mathsf{else\ if}\ v\ 197\ \mathsf{then\ Some}\ 2\ \mathsf{else\ None}\rangle}\rangle} \end{aligned}$$

where $v = \lambda y.\langle \boxed{(\lambda x.\langle succ\, x\rangle)\, \boxed{y}}\rangle$. We can observe that all borders in the last term are monitored by casts.

Evaluation rules, presented at the bottom of Figure 2, are standard: (E_Step) reduces a subterm that is a redex in a program and (E_Abort) halts evaluation of a program at blame when cast failure happens. To determine a redex in a program, we use evaluation contexts [11], which are defined as follows.

$$E ::= [\,] \mid op(\overline{v_i}^{\,i}, E, \overline{t_j}^{\,j}) \mid E\, s \mid v\, E \mid \langle E\rangle \mid E : A \Rightarrow^p B \mid E : G \Rightarrow \star$$

This definition means that terms are evaluated from left to right. Unlike pure evaluation contexts, evaluation contexts include a context where the hole is put in the body of a reset operator.

3.3 Type System

This section presents a type system of our calculus. It is defined as a combination of that of Danvy and Filinski and that of Wadler and Findler. As usual, we use typing contexts, denoted by Γ, to denote a mapping of variables to types:

$$\Gamma ::= \emptyset \mid \Gamma, x{:}A$$

Typing judgments in our type system take the form $\Gamma; \alpha \vdash s : A; \beta$, which means that term s is typed at type A under typing context Γ and it modifies answer type α to β when evaluated. Perhaps, it may be easier to understand what the typing judgment means when its CPS transformation is considered. When we write $[\![\cdot]\!]$ for the CPS transformation, the typing judgment $\Gamma; \alpha \vdash s : A; \beta$ is translated into the form $[\![\Gamma]\!] \vdash [\![s]\!] : ([\![A]\!] \to [\![\alpha]\!]) \to [\![\beta]\!]$ in the simply typed blame calculus (without shift/reset). That is, type A of term s and type α are the argument type and the return type of a continuation, respectively, and type β is the type of the whole computation result when the continuation is passed.

Figure 3 shows typing rules for deriving typing judgments. Typing rules for shift operators, reset operators, and terms from the lambda calculus are the same as Danvy and Filinski's type system. In (T_Op), we use function ty from primitive operator names to their (first-order) types. Typing rules for terms from the blame calculus are changed to follow Danvy and Filinski's type system. In (T_Cast), following previous work on the blame calculus, we restrict casts in well typed programs to be ones between compatible types. In other words, (T_Cast) rules out casts that will always fail. The typing rule (T_Blame) seems to allow blame to modify answer types to any type though blame does not invoke shift operator; this causes no problems (and is necessary for type soundness) because blame halts a program.

3.4 Type Soundness

We show type soundness of our calculus in the standard way: Preservation and Progress [31]. In the presence of the dynamic type, we can write a divergent

$\boxed{\Gamma;\alpha \vdash t \,:\, A;\beta}$ **Typing rules**

$$\frac{}{\Gamma;\alpha \vdash c \,:\, ty\,(c);\alpha}\ \text{T\_Const} \qquad \frac{ty\,(op) \,=\, \overline{\iota_i}^{\,i} \to \iota \quad \overline{\Gamma;\alpha_i \vdash t_i \,:\, \iota_i;\alpha_{i-1}}^{\,i}}{\Gamma;\alpha_n \vdash op(\overline{t_i}^{\,i}) \,:\, \iota;\alpha_0}\ \text{T\_Op}$$

$$\frac{}{\Gamma;\alpha \vdash \mathsf{blame}\,p \,:\, A;\beta}\ \text{T\_Blame} \qquad \frac{\Gamma, x{:}A;\beta \vdash t \,:\, B;\gamma}{\Gamma;\alpha \vdash \lambda x.\, t \,:\, A/\beta \to B/\gamma;\alpha}\ \text{T\_Abs}$$

$$\frac{x{:}A \,\in\, \Gamma}{\Gamma;\alpha \vdash x \,:\, A;\alpha}\ \text{T\_Var} \qquad \frac{\Gamma;\gamma \vdash t \,:\, A/\alpha \to B/\beta;\delta \quad \Gamma;\beta \vdash s \,:\, A;\gamma}{\Gamma;\alpha \vdash t\,s \,:\, B;\delta}\ \text{T\_App}$$

$$\frac{\Gamma;\alpha \vdash s \,:\, A;\beta \quad A \,\sim\, B}{\Gamma;\alpha \vdash (s \,:\, A \Rightarrow^{p}\ B) \,:\, B;\beta}\ \text{T\_Cast} \qquad \frac{\Gamma;\alpha \vdash s \,:\, G;\beta}{\Gamma;\alpha \vdash (s \,:\, G \Rightarrow \star) \,:\, \star;\beta}\ \text{T\_Ground}$$

$$\frac{\Gamma, k{:}A/\gamma \to \alpha/\gamma;\delta \vdash s \,:\, \delta;\beta}{\Gamma;\alpha \vdash \mathcal{S}k.\, s \,:\, A;\beta}\ \text{T\_Shift} \qquad \frac{\Gamma;\beta \vdash s \,:\, \beta;A}{\Gamma;\alpha \vdash \langle s\rangle \,:\, A;\alpha}\ \text{T\_Reset}$$

Fig. 3. Typing rules.

term easily, and blame is a legitimate state of program evaluation. Thus, type soundness in this paper means that any well typed program (a closed term enclosed by reset) evaluates to a well typed value, diverges, or raises blame. In what follows, we write $\longmapsto^*$ for the reflexive and transitive closure of $\longmapsto$.

Theorem 1 (Type Soundness). *If $\emptyset;\alpha \vdash \langle s\rangle \,:\, A;\alpha$, then one of the followings holds:*

- *there is an infinite evaluation sequence from $\langle s\rangle$;*
- *$\langle s\rangle \longmapsto^* \mathsf{blame}\,p$ for some p; or*
- *$\langle s\rangle \longmapsto^* v$ for some v such that $\emptyset;\alpha \vdash v \,:\, A;\alpha$.*

The outermost reset is assumed to exclude terms stuck at a shift operator without a surrounding reset. The statement of Progress shown after Preservation, however, has to take into account such a possibility for proof by induction to work.

Lemma 1 (Preservation). *If $\emptyset;\alpha \vdash s \,:\, A;\beta$ and $s \longmapsto t$, then $\emptyset;\alpha \vdash t \,:\, A;\beta$.*

Proof. By induction on the typing derivation with case analysis on the reduction/evaluation rule applied to s. In the case for (R_Shift), we follow the proof in the previous work on shift/reset [3].

Lemma 2 (Progress). *If $\emptyset;\alpha \vdash s \,:\, A;\beta$, then one of the followings holds:*

- *$s \longmapsto s'$ for some s';*
- *s is a value;*
- *$s \,=\, \mathsf{blame}\,p$ for some p; or*
- *$s \,=\, F[\mathcal{S}k.\, t]$ for some F, k and t.*

Proof. Straightforward by induction on the typing derivation.

Proof (Theorem 1). By Progress and Preservation. Note that the evaluation from $\langle s\rangle$ to $F[\mathcal{S}k.\, t]$ as stated in Progress does not happen since s is enclosed by reset and reset does not appear in F.

4 Blame Theorem

Blame Theorem intuitively states that values from the typed code will never be sources of cast failure at run time and, more specifically, clarifies conditions under which some blame never happens. Following the original work [30], we formalize such conditions using a few, different subtyping relations. Our proof is based on that in Ahmed et al.'s work [1], which defined a safety relation for terms and showed Blame Preservation and Blame Progress like preservation and progress for type soundness.

4.1 Subtyping

To state a Blame Theorem, we introduce naive subtyping $<:_n$, which formalizes the notion of being "more precisely typed." Roughly speaking, type A is a naive subtype of B when A is obtained by substituting some types for occurrences of the dynamic type in B. For example, $\mathsf{int} <:_n \star$ and $\mathsf{int}/\mathsf{int} \to \mathsf{int}/\mathsf{int} <:_n \star/\mathsf{int} \to \mathsf{int}/\star$. Note that argument types are covariant here. The Blame Theorem states that if type A is a naive subtype of type B, then the side of A is never blamed, that is, a cast $s : A \Rightarrow^p B$ does not cause $\mathsf{blame}\, p$ and $s : B \Rightarrow^p A$ does not $\mathsf{blame}\, \bar{p}$.

To prove the Blame Theorem, we introduce positive and negative subtyping. Intuitively, that type A is a positive (resp. negative) subtype of B expresses that positive (resp. negative) blame never happens for a cast from A to B. It turns out that naive subtyping can be expressed in terms of positive and negative subtyping, from which the Blame Theorem easily follows. In addition, a cast from an ordinary subtype—where argument types of function types are contravariant—to a supertype is shown not to raise blame.

Subtyping relations—ordinary subtyping $<:$, naive subtyping $<:_n$, positive subtyping $<:^+$, and negative subtyping $<:^-$—are reflexive relations satisfying subtyping rules presented in Figure 4. The idea shared across all subtyping rules for function types is that function type $A/\alpha \to B/\beta$ is interpreted as if it takes the CPS-transformation form $A \to (B \to \alpha) \to \beta$. In this form, A and α occur at negative positions while B and β occur at positive positions.

We write $A <: B$ to denote that A is a subtype of B. The rule (S_Dyn) means that any (nondynamic) type is a subtype of the dynamic type if it is a subtype of the (unique) ground type compatible to it. The premise is needed for cases that the subtype is higher order. Function types are covariant at positive positions and contravariant at negative positions as usual.

As mentioned before, type A is a naive subtype of B when A is obtained by putting some types in occurrences of the dynamic type in B. The rule (SN_Dyn) means that the dynamic type is least precise. In the rule (SN_Fun), function types for naive subtyping are covariant in both positive and negative positions.

The definitions of positive and negative subtyping are mutually recursive. The rule (S^+_Dyn) means that positive blame never happens when any value is coerced to the dynamic type. Similarly to ordinary subtyping, in (S^+_Fun), function types are covariant at positive positions and contravariant at negative

$\boxed{A <: B}$ **Subtype**

$$\frac{A <: G}{A <: \star}\ \text{S\_Dyn} \qquad \frac{A' <: A \quad B <: B' \quad \alpha' <: \alpha \quad \beta <: \beta'}{A/\alpha \to B/\beta <: A'/\alpha' \to B'/\beta'}\ \text{S\_Fun}$$

$\boxed{A <:_n B}$ **Naive Subtype**

$$\frac{}{A <:_n \star}\ \text{SN\_Dyn} \qquad \frac{A <:_n A' \quad B <:_n B' \quad \alpha <:_n \alpha' \quad \beta <:_n \beta'}{A/\alpha \to B/\beta <:_n A'/\alpha' \to B'/\beta'}\ \text{SN\_Fun}$$

$\boxed{A <:^+ B}$ **Positive Subtype**

$$\frac{}{A <:^+ \star}\ \text{S}^+\text{\_Dyn} \qquad \frac{A' <:^- A \quad B <:^+ B' \quad \alpha' <:^- \alpha \quad \beta <:^+ \beta'}{A/\alpha \to B/\beta <:^+ A'/\alpha' \to B'/\beta'}\ \text{S}^+\text{\_Fun}$$

$\boxed{A <:^- B}$ **Negative Subtype**

$$\frac{}{\star <:^- A}\ \text{S}^-\text{\_Dyn} \qquad \frac{A <:^- G}{A <:^- B}\ \text{S}^-\text{\_Any}$$

$$\frac{A' <:^+ A \quad B <:^- B' \quad \alpha' <:^+ \alpha \quad \beta <:^- \beta'}{A/\alpha \to B/\beta <:^- A'/\alpha' \to B'/\beta'}\ \text{S}^-\text{\_Fun}$$

Fig. 4. Subtyping rules.

positions. Negative subtyping is a reversed version of positive subtyping except for addition of (S^-_Any), which is a combination of (S^-_Dyn) and the fact that a cast from type A to the dynamic type never gives rise to negative blame when A is a negative subtype of its ground type. The rule (S^-_Any) follows from Ahmed et al.'s work [1] and represents a relaxed form of the system of Wadler and Findler [30]. Notice that polarity of subtyping is reversed at negative positions.

As mentioned above, we show that naive subtyping (and ordinary subtyping) can be expressed in terms of positive and negative subtyping.

Lemma 3. *$A <:_n B$ iff $A <:^+ B$ and $B <:^- A$.*

Lemma 4. *$A <: B$ iff $A <:^+ B$ and $A <:^- B$.*

The proofs of the direction from left to right are straightforward by induction on the derivations of $A <:_n B$ and $A <: B$. The other direction is shown by structural induction on A.

4.2 Blame Theorem

The proof of the Blame Theorem is similar to preservation and progress for type soundness. Instead of a type system, we introduce a safety relation using positive and negative subtyping and show Blame Preservation, which states safety is

$$\frac{s\ \mathsf{sf}\ p \quad A <:^{+} B}{s\ :\ A \Rightarrow^{p} B\ \mathsf{sf}\ p} \qquad \frac{s\ \mathsf{sf}\ p \quad A <:^{-} B}{s\ :\ A \Rightarrow^{\bar{p}} B\ \mathsf{sf}\ p} \qquad \frac{}{c\ \mathsf{sf}\ p} \qquad \frac{\forall i.\ t_i\ \mathsf{sf}\ p}{op(\overline{t_i}^{\,i})\ \mathsf{sf}\ p} \qquad \frac{}{x\ \mathsf{sf}\ p}$$

$$\frac{s\ \mathsf{sf}\ p}{\lambda x.\, s\ \mathsf{sf}\ p} \qquad \frac{s\ \mathsf{sf}\ p \quad t\ \mathsf{sf}\ p}{s\ t\ \mathsf{sf}\ p} \qquad \frac{q \neq p \quad q \neq \bar{p} \quad s\ \mathsf{sf}\ p}{s\ :\ A \Rightarrow^{q} B\ \mathsf{sf}\ p} \qquad \frac{s\ \mathsf{sf}\ p}{s\ :\ G \Rightarrow \star\ \mathsf{sf}\ p}$$

$$\frac{q \neq p}{\mathsf{blame}\ q\ \mathsf{sf}\ p} \qquad \frac{s\ \mathsf{sf}\ p}{\mathcal{S}k.\, s\ \mathsf{sf}\ p} \qquad \frac{s\ \mathsf{sf}\ p}{\langle s \rangle\ \mathsf{sf}\ p}$$

Fig. 5. Safety rules.

preserved by evaluation, and Blame Progress, which states that a safe term does not give rise to blame. In this section, we focus only on whether a term gives rise to blame or not and not on whether a term gets stuck or not.

A term s is *safe for blame label* p, written as $s\ \mathsf{sf}\ p$, if every cast with blame label p in s is from a type to its positive supertype and every cast with $\bar{p}$ is from a type to its negative supertype. We present inference rules for the safety relation in Figure 5. From the definition, it is observed that a term safe for p does not contain blame with p; this does not restrict a source program written by a programmer because it should not contain any blame.

Blame Preservation and Blame Progress show that, if $s\ \mathsf{sf}\ p$, term s never gives rise to blame with label p. We write $s \not\mapsto t$ and $s \not\mapsto^{*} t$ to denote that term s does not reduce to term t in a single step and in multiple steps, respectively.

Lemma 5 (Blame Preservation). *If $s\ \mathsf{sf}\ p$ and $s \longmapsto t$, then $t\ \mathsf{sf}\ p$.*

Lemma 6 (Blame Progress). *If $s\ \mathsf{sf}\ p$, then $s \not\mapsto \mathsf{blame}\ p$.*

Finally, we show the Blame Theorem—values that flow from the more precisely typed side never cause blame—and, furthermore, that casts from one type to its supertype never give rise to blame.

Theorem 2 (Blame Theorem).
Let s be a term with a subterm $t : A \Rightarrow^{p} B$ where cast is labeled by the only occurrence of p in s. Moreover, suppose that $\bar{p}$ does not appear in s.

1. *If $A <:^{+} B$, then $s \not\mapsto^{*} \mathsf{blame}\ p$.*
2. *If $A <:^{-} B$, then $s \not\mapsto^{*} \mathsf{blame}\ \bar{p}$.*
3. *If $A <:_{n} B$, then $s \not\mapsto^{*} \mathsf{blame}\ p$; if $B <:_{n} A$, then $s \not\mapsto^{*} \mathsf{blame}\ \bar{p}$.*
4. *If $A <: B$, then $s \not\mapsto^{*} \mathsf{blame}\ p$ and $s \not\mapsto^{*} \mathsf{blame}\ \bar{p}$.*

Proof. The first and second cases are shown by Blame Preservation and Blame Progress because $s\ \mathsf{sf}\ p$ in the first case and $s\ \mathsf{sf}\ \bar{p}$ in the second case. The third case (resp. the fourth case) follows from the first and second cases and Lemma 3 (resp. Lemma 4).

5 CPS Transformation

The semantics of programming languages with control operators has often been established by transformation of programs with control operators to continuation passing style (CPS), a programming style where continuations appear in a program as arguments of functions. For example, programs with Reynolds's escape operator [19], call/cc in Scheme, shift/reset [6], and so on can be transformed to CPS form.

As a proof of correctness of our approach, we define a CPS transformation from terms in our calculus to those in the simply typed blame calculus of Ahmed et al. [1] and show that a well typed source term is transformed to a well typed target term and, for any source terms such that one reduces to the other, their CPS-transformation results are equivalent in the target calculus. The equational system is based on call-by-value axioms [20] due to blame, which is effectful.

Before giving the CPS transformation, we modify the syntax and the reduction rule (R_Ground) of our calculus slightly in order to transform a ground value of the form $v : \star/\star \to \star/\star \Rightarrow \star$ to a value with a cast (the reason is detailed later). To assign a blame label to the cast, the syntax is changed as follows:

$$v ::= \ldots \mid v : G \Rightarrow_p \star \qquad s ::= \ldots \mid s : G \Rightarrow_p \star$$

Blame labels in ground terms and values are given as subscripts for ease of distinction from casts. The reduction rule (R_Ground) takes the following form:

$$v : A \Rightarrow^p \star \longrightarrow (v : A \Rightarrow^p G) : G \Rightarrow_p \star \text{ (if } A \sim G \text{ and } A \neq \star) \text{ R\_Ground}$$

Our CPS transformation, which mostly follows Danvy and Filinski [6], is shown in Figure 6 in three parts: transformation for types, values, and terms. We use variable κ to denote continuations. The CPS transformation for types is standard. A function of type $A/\alpha \to B/\beta$ takes an argument of A, would pass a value of B to a continuation that returns α, and results in a value of β as the computation result. The CPS transformation for values maps values in our calculus to those in the blame calculus without shift/reset. The definition shown in Figure 6 is easy to understand except for ground values where the ground type is a function type. We might expect that the CPS-transformation result of ground value $v : G \Rightarrow_p \star$ can be defined as $v^* : [\![G]\!] \Rightarrow \star$. However, that form would not be a valid term in the target calculus if the ground type G is a function type, because the ground function type in the target calculus takes only the form $\star \to \star$ but $[\![\star/\star \to \star/\star]\!] = \star \to (\star \to \star) \to \star$. Expecting a value will be translated to a value in the target calculus, we set a ground value $v : \star/\star \to \star/\star \Rightarrow_p \star$ to be mapped to *a value to which* $v^* : [\![G]\!] \Rightarrow^p \star$ *reduces*, instead. (Notice the superscript on $\Rightarrow$. A term $v^* : [\![G]\!] \Rightarrow^p \star$ is a cast and always valid.) In the result, we omit the trivial cast $x : \star \Rightarrow^{\bar{p}} \star$. The CPS transformation for terms is self-explanatory.

It is straightforward to show that well typed source terms are transformed to well typed target terms. For any typing context Γ, we write $[\![\Gamma]\!]$ for the typing context obtained by applying the CPS transformation to types mapped by Γ.

$\boxed{[\![A]\!]}$ **CPS Transformation (Types)**

$$[\![\iota]\!] = \iota \qquad [\![\star]\!] = \star \qquad [\![A/\alpha \to B/\beta]\!] = [\![A]\!] \to ([\![B]\!] \to [\![\alpha]\!]) \to [\![\beta]\!]$$

$\boxed{v^*}$ **CPS Transformation (Values)**

$$x^* = x \qquad c^* = c \qquad (\lambda x.\, s)^* = \lambda x.\, [\![s]\!] \qquad (v : \iota \Rightarrow \star)^* = v^* : \iota \Rightarrow \star$$
$$(v : \star/\star \to \star/\star \Rightarrow_p \star)^* = (\lambda x.\, (v^*\, x) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star$$

$\boxed{[\![s]\!]}$ **CPS Transformation (Terms)**

$$\begin{aligned}
[\![v]\!] &= \lambda\kappa.\, \kappa\, v^* \\
[\![op(\overline{t_i}^{\,i})]\!] &= \lambda\kappa.\, [\![t_1]\!]\, (\lambda x_1.\, \ldots\, [\![t_n]\!]\, (\lambda x_n.\, \kappa\, op(\overline{x_i}^{\,i}))\, \ldots) \\
[\![s\ t]\!] &= \lambda\kappa.\, [\![s]\!]\, (\lambda x.\, [\![t]\!]\, (\lambda y.\, x\ y\ \kappa)) \\
[\![\langle s \rangle]\!] &= \lambda\kappa.\, \kappa\, ([\![s]\!]\, (\lambda x.\, x)) \\
[\![\mathcal{S}k.\, s]\!] &= \lambda\kappa.\, ([\![s]\!]\, (\lambda x.\, x))\, [k := \lambda x.\, \lambda\kappa'.\, \kappa'\, (\kappa\, x)] \\
[\![s : A \Rightarrow^p B]\!] &= \lambda\kappa.\, [\![s]\!]\, (\lambda x.\, \kappa\, (x : [\![A]\!] \Rightarrow^p [\![B]\!])) \\
[\![s : G \Rightarrow \star]\!] &= \lambda\kappa.\, [\![s]\!]\, (\lambda x.\, \kappa\, (x : G \Rightarrow \star)^*) \\
[\![\mathsf{blame}\, p]\!] &= \lambda\kappa.\, \mathsf{blame}\, p
\end{aligned}$$

Fig. 6. CPS transformation.

Theorem 3 (Preservation of Type). *If* $\Gamma; \alpha \vdash s : A; \beta$, *then* $[\![\Gamma]\!] \vdash [\![s]\!] : ([\![A]\!] \to [\![\alpha]\!]) \to [\![\beta]\!]$.

Next, we define an equational system in the target calculus. The system consists of axioms about casts as well as usual call-by-value axioms [20]. In what follows, we use metavariables $\mathbb{e}$, $\mathbb{v}$, $\mathbb{E}$, and $\mathbb{A}$ (and $\mathbb{B}$) to denote terms, values, evaluation contexts, and types in the target calculus, respectively, and write $fv\,(\mathbb{v})$ and $fv\,(\mathbb{E})$ for the sets of free variables in $\mathbb{v}$ and $\mathbb{E}$, respectively. In addition, let the relation $\Longrightarrow$ be the evaluation relation in the target calculus.

Definition 1 (Term Equality) *The relation* $\approx$ *is the least congruence that contains the following axioms:*

$$\frac{\mathbb{e}_1 \Longrightarrow \mathbb{e}_2}{\mathbb{e}_1 \approx \mathbb{e}_2} \qquad \frac{x \notin fv\,(\mathbb{v})}{\lambda x.\, \mathbb{v}\, x \approx \mathbb{v}} \qquad \frac{x \notin fv\,(\mathbb{E})}{(\lambda x.\, \mathbb{E}[x])\, \mathbb{e} \approx \mathbb{E}[\mathbb{e}]}$$
$$\mathbb{e} : \star \Rightarrow^p \star \approx \mathbb{e} \qquad \mathbb{e} : \star \Rightarrow^p \star \to \star \Rightarrow^p \mathbb{A} \to \mathbb{B} \approx \mathbb{e} : \star \Rightarrow^p \mathbb{A} \to \mathbb{B}$$

We think that the last two axioms about casts are reasonable. The former, which skips the trivial cast, is found in another blame calculus [24]. This axiom is introduced mainly to ignore redundant casts that often happen in CPS-transformation results. The latter axiom, which collapses two casts into one, is used to show terms reduced by (R_Collapse) are equivalent after CPS transformation.

Now, we show that the relationship between our semantics in direct-style and the CPS transformation.

Theorem 4 (Preservation of Equality). *If* $s \longmapsto t$*, then* $[\![s]\!] \approx [\![t]\!]$.

6 Related Work

Gradual typing and Blame Theorem. Blame calculi are variants of lambda calculi for gradual typing by Siek and Taha [22], a mechanism to integrate static and dynamic typing. Since the seminal work by Siek and Taha, the notion of gradual typing has spread over various programming constructs—e.g., higher-order functions [22], objects [23], mutable references [16], polymorphism [1], etc. The property that values that flow from typed code never trigger cast failure was studied first in the context of contract checking [27]. Wadler and Findler [30] adopted blame of finer forms (positive and negative blame), following Findler and Felleisen's work [13], and investigated conditions under which blame does not happen. They discovered that the notion of being "more precisely typed" can be formalized as naive subtyping.

Delimited-control operators. Roughly speaking, there have been two major families of delimited-control operators: so-called "static" control operators, including shift/reset [5, 6], and so-called "dynamic" control operators, including control/prompt [10, 15]. In this work, we choose shift/reset because their type system and CPS transformation are well studied. In fact, CPS transformation for shift/reset has served as a guide to designing our cast mechanism. Given recent studies on relationship of control/prompt to their CPS transformation [21, 7, 9] and a type system for control/prompt [18], we leave an extension of blame calculi to control/prompt for interesting future work.

Gradual typing with delimited-control operators. Most closely related work is Takikawa et al. [26]; they have also studied integration of static and dynamic typing in the presence of control operators. They proposed a contract system for programs with control operators in Racket [15] and showed that values from typed parts never trigger blame (in the sense of Tobin-Hochstadt and Felleisen [27]) through the complete monitoring property [8]. Aside from an obvious difference in the choice of control operators, our calculus has finer-grained control over how typed and untyped parts can be mixed: e.g., a function of type $\mathsf{int} \to \star$ cannot be expressed in Takikawa et al. because there are only fully typed and fully untyped modules. We also define a CPS transformation for our calculus, and investigate the relationship between our calculus and the CPS transformation. Although shift and reset can be implemented by using control operators in Racket [15], it is not very clear whether their contract system can simulate our casts for function types with answer types naturally.

7 Conclusion

We have proposed a new cast-based mechanism to monitor all communications between typed and untyped code in the presence of shift/reset. It is inspired by Danvy and Filinski's type system. To justify the design of our cast semantics, we have defined a simply typed blame calculus with shift/reset and shown the Blame Theorem and soundness of the CPS transformation. We have found additional axioms for the equational system in the target language in proving the soundness.

There are many directions for future work. First is an extension of our blame calculus with refinement types. Effects in refinements are obviously problematic. One possible solution would be to restrict refinements to be pure. It is interesting to investigate how such purity restriction can be relaxed. Second is to apply succeeding work about blame calculi, such as space-efficiency [16] and parametricity [1], to our calculus. In particular, an extension with parametricity would be challenging because it is not clear how control operators and the ν-operator interact with each other. Finally, we would like to develop a contract system corresponding to our calculus and to inspect more detailed relationship to the contract system of Takikawa et al.

Acknowledgments. We would like to thank Matthias Felleisen, Robby Findler, Philip Wadler, and anonymous reviewers of APLAS 2015 for valuable comments. This work was supported in part by Grant-in-Aid for Scientific Research (B) No. 25280024 from MEXT of Japan. The title is derived from that of a paper by Kameyama, Kiselyov, and Shan [17].

References

1. Ahmed, A., Findler, R.B., Siek, J.G., Wadler, P.: Blame for all. In: Proc. of ACM POPL. pp. 201–214 (2011)
2. Asai, K., Kameyama, Y.: Polymorphic delimited continuations. In: Proc. of APLAS. LNCS, vol. 4807, pp. 239–254 (2007)
3. Asai, K., Kameyama, Y.: Polymorphic delimited continuations. CS-TR-07-10, Dept. of Computer Science, University of Tsukuba (2007)
4. Bonnaire-Sergeant, A., Davies, R., Tobin-Hochstadt, S.: Practical optional types for Clojure, unpublished draft
5. Danvy, O., Filinski, A.: A functional abstraction of typed contexts. 89/12, DIKU, University of Copenhagen (1989)
6. Danvy, O., Filinski, A.: Abstracting control. In: LISP and Functional Programming. pp. 151–160 (1990)
7. Dariusz Biernacki, O.D., Millikin, K.: A dynamic continuation-passing style for dynamic delimited continuations. Research Series RS-06-15, BRICS, DAIMI (2006)
8. Dimoulas, C., Tobin-Hochstadt, S., Felleisen, M.: Complete monitors for behavioral contracts. In: Proc. of ESOP. LNCS, vol. 7211, pp. 214–233 (2012)
9. Dybvig, R.K., Peyton Jones, S.L., Sabry, A.: A monadic framework for delimited continuations. J. Funct. Program. 17(6), 687–730 (2007)
10. Felleisen, M.: The theory and practice of first-class prompts. In: Proc. of ACM POPL. pp. 180–190 (1988)

11. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theor. Comput. Sci. 103(2), 235–271 (Sep 1992)
12. Filinski, A.: Representing monads. In: Proc. of ACM POPL. pp. 446–457 (1994)
13. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proc. of ACM ICFP. pp. 48–59 (2002)
14. Flanagan, C.: Hybrid type checking. In: Proc. of ACM POPL. pp. 245–256 (2006)
15. Flatt, M., Yu, G., Findler, R.B., Felleisen, M.: Adding delimited and composable control to a production programming environment. In: Proc. of ACM ICFP. pp. 165–176 (2007)
16. Herman, D., Tomb, A., Flanagan, C.: Space-efficient gradual typing. In: Trends in Functional Prog. pp. 1–18 (2007)
17. Kameyama, Y., Kiselyov, O., Shan, C.: Shifting the stage: Staging with delimited control. In: Proc. of ACM PEPM. pp. 111–120 (2009)
18. Kameyama, Y., Yonezawa, T.: Typed dynamic control operators for delimited continuations. In: Proc. of FLOPS. LNCS, vol. 4989, pp. 239–254 (2008)
19. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proc. of ACM Annual Conference. pp. 717–740 (1972)
20. Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. Lisp and Symbolic Computation 6(3-4), 289–360 (1993)
21. Shan, C.: Shift to control. In: Scheme and Functional Programming Workshop. pp. 99–107 (2004)
22. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop. pp. 81–92 (2006)
23. Siek, J.G., Taha, W.: Gradual typing for objects. In: Proc. of ECOOP. LNCS, vol. 4609, pp. 2–27 (2007)
24. Siek, J.G., Wadler, P.: Threesomes, with and without blame. In: Proc. of ACM POPL. pp. 365–376 (2010)
25. Sitaram, D.: Handling control. In: Proc. of ACM PLDI. pp. 147–155 (1993)
26. Takikawa, A., Strickland, T.S., Tobin-Hochstadt, S.: Constraining delimited control with contracts. In: Proc. of ESOP. LNCS, vol. 7792, pp. 229–248 (2013)
27. Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: From scripts to programs. In: Dynamic Language Symposium. pp. 964–974 (2006)
28. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. In: Proc. of ACM POPL. pp. 395–406 (2008)
29. Vitousek, M.M., Kent, A.M., Siek, J.G., Baker, J.: Design and evaluation of gradual typing for Python. In: Dynamic Language Symposium. pp. 45–56 (2014)
30. Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: Proc. of ESOP. LNCS, vol. 5502, pp. 1–16 (2009)
31. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115(1), 38–94 (1994)