Exploring the Design Space of Higher-Order Casts

Jeremy Siek[1,⋆], Ronald Garcia[2], and Walid Taha[2,⋆⋆]

[1] University of Colorado, Boulder, CO 80309, USA
[2] Rice University, Houston, TX 77005, USA
jeremy.siek@colorado.edu, ronald.garcia@rice.edu, taha@rice.edu

Abstract. This paper explores the surprisingly rich design space for the simply typed lambda calculus with casts and a dynamic type. Such a calculus is the target intermediate language of the gradually typed lambda calculus but it is also interesting in its own right. In light of diverse requirements for casts, we develop a modular semantic framework, based on Henglein's Coercion Calculus, that instantiates a number of space-efficient, blame-tracking calculi, varying in what errors they detect and how they assign blame. Several of the resulting calculi extend work from the literature with either blame tracking or space efficiency, and in doing so reveal previously unknown connections. Furthermore, we introduce a new strategy for assigning blame under which casts that respect traditional subtyping are statically guaranteed to never fail. One particularly appealing outcome of this work is a novel cast calculus that is well-suited to gradual typing.

1 Introduction

This paper explores the design space for $\lambda^{\langle\cdot\rangle}_{\rightarrow}$, the simply typed lambda calculus with a dynamic type and cast expressions. Variants of this calculus have been used to express the semantics of languages that combine dynamic and static typing [2, 3, 5, 6, 8–11].

The syntax of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ is given in Fig. 1. The dynamic type Dyn is assigned to values that are tagged with their run-time type. The cast expression, $\langle T \Leftarrow S\rangle^{l} e$, coerces a run-time value from type S to T or halts with a cast error if it cannot perform the coercion. More precisely, the calculus evaluates e to a value v, checks whether the run-time type of v is consistent with T, and if so, returns the coercion of v to T. Otherwise execution halts and signals that the cast at location l of the source program caused an error.

The semantics of first-order casts (casts on base types) is straightforward. For example, casting an integer to Dyn and then back to Int behaves like the identity function.

$$\langle \texttt{Int} \Leftarrow \texttt{Dyn}\rangle^{l_2} \langle \texttt{Dyn} \Leftarrow \texttt{Int}\rangle^{l_1} 4 \longmapsto^{*} 4$$

On the other hand, casting an integer to Dyn and then to Bool raises an error and reports the source location of the cast that failed.

$$\langle \texttt{Bool} \Leftarrow \texttt{Dyn}\rangle^{l_2} \langle \texttt{Dyn} \Leftarrow \texttt{Int}\rangle^{l_1} 4 \longmapsto^{*} \textbf{blame}\ l_2$$

We say that the cast at location l_2 is *blamed* for the cast error.

⋆ Supported by NSF grant CCF-0702362.
⋆⋆ Supported by NSF grants CCF-0747431 and CCF-0439017.

Base Types	B	$\supset$	$\{\texttt{Int}, \texttt{Bool}\}$
Types	S, T	$::=$	$B \mid \texttt{Dyn} \mid S \to T$
Blame labels	l	$\in$	$\mathbb{L}$ Integers $n \in \mathbb{Z}$
Constants	k	$\in$	$\mathbb{K} \supset \{n, \texttt{True}, \texttt{False}\}$
Variables	x	$\in$	$\mathbb{V}$
Expressions	e	$::=$	$x \mid k \mid \lambda x : T.\ e \mid e\ e \mid \langle T \Leftarrow S\rangle^l e$

Fig. 1. Syntax for the lambda calculus with casts ($\lambda^{\langle\cdot\rangle}_{\to}$)

Extending casts from first-order to higher-order (function) types raises several issues. For starters, higher-order casts cannot always be checked immediately. In other words, it is not generally possible to decide at the point where a higher-order cast is applied to a value whether that value will always behave according to the type ascribed by the cast. For example, when the following function is cast to $\texttt{Int} \to \texttt{Int}$, there is no way to immediately tell if the function will return an integer every time it is called.

$$\langle \texttt{Int} \to \texttt{Int} \Leftarrow \texttt{Int} \to \texttt{Dyn}\rangle(\lambda x : \texttt{Int}.\ \mathsf{if}\ 0 < x\ \texttt{then}\ \langle \texttt{Dyn} \Leftarrow \texttt{Bool}\rangle\texttt{True}\ \mathsf{else}\ \langle \texttt{Dyn} \Leftarrow \texttt{Int}\rangle 2)$$

So long as the function is only called with positive numbers, its behavior respects the cast. If it is ever called with a negative number, however, its return value will violate the invariant imposed by the cast.

The standard solution, adopted from work on higher-order contracts [1], defers checking the cast until the function is applied to an argument and then checks the cast against the particular argument and return value. This can be accomplished by using the cast as a wrapper and splitting it when the wrapped function is applied to an argument:

$$(\textsc{AppCst}) \qquad (\langle T_1 \to T_2 \Leftarrow S_1 \to S_2\rangle v_1)\ v_2 \longrightarrow \langle T_2 \Leftarrow S_2\rangle(v_1\ \langle S_1 \Leftarrow T_1\rangle v_2))$$

Because a higher-order cast is not checked immediately, it might fail in a context far removed from where it was originally applied. To help diagnose such failures, dynamic semantics are enhanced with *blame tracking*, a facility that traces failures back to their origin in the source program [1, 4, 10].

Several dynamic semantics for casts have been proposed in the literature and their differences, though subtle, produce surprisingly different results for some programs. We use the following abbreviations: **ST** for Siek and Taha [8], **HTF-L** for Herman et al. [7] (lazy variant), **HTF-E** for Herman et al. [7] (eager variant), **WF-1** for Wadler and Findler [10], and **WF-2** for [11]. Consider how these five semantics for $\lambda^{\langle\cdot\rangle}_{\to}$ produce different results for a few small examples.

The following program casts a function of type $\texttt{Int} \to \texttt{Int}$ to $\texttt{Dyn}$ and then to $\texttt{Bool} \to \texttt{Int}$. It produces a run-time cast error in **ST**, **WF-1**, and **HTF-E** but not in **WF-2** and **HTF-L**.

$$(1) \qquad \langle \texttt{Bool} \to \texttt{Int} \Leftarrow \texttt{Dyn}\rangle\langle \texttt{Dyn} \Leftarrow \texttt{Int} \to \texttt{Int}\rangle(\lambda x : \texttt{Int}.\ x)$$

With a small change, the program runs without error for four of the semantics but fails in **HTF-E**:

$$(2) \qquad \langle \texttt{Bool} \to \texttt{Int} \Leftarrow \texttt{Dyn} \to \texttt{Dyn}\rangle\langle \texttt{Dyn} \to \texttt{Dyn} \Leftarrow \texttt{Int} \to \texttt{Int}\rangle(\lambda x : \texttt{Int}.\ x)$$

It seems surprising that any of the semantics allows a function of type $\texttt{Int} \to \texttt{Int}$ to be cast to $\texttt{Bool} \to \texttt{Int}$!

Next consider the semantics of blame assignment. The following program results in a cast error, but which of the three casts should be blamed?

(3) $\quad (\langle \texttt{Dyn} \to \texttt{Int} \Leftarrow \texttt{Dyn}\rangle^{l_3} \langle \texttt{Dyn} \Leftarrow \texttt{Bool} \to \texttt{Bool}\rangle^{l_2} \lambda x : \texttt{Bool}.\ x)\langle \texttt{Dyn} \Leftarrow \texttt{Int}\rangle^{l_1} 1$

The semantics **ST**, **HTF-E**, and **HTF-L** do not perform blame tracking. Both **WF-1** and **WF-2** blame l_2. This is surprising because, intuitively, casting a value up to Dyn always seems safe. On the other hand, casting a dynamic value down to a concrete type is an opportunity for type mismatch.

In this paper we map out the design space for $\lambda^{\langle\cdot\rangle}_{\to}$ using two key insights. First, the semantics of higher-order casts can be categorized as detecting cast errors using an eager, partially eager, or lazy strategy (Section 2). Second, different blame tracking strategies yield different notions of a statically safe cast (a cast that will never be blamed for a run-time cast error) which are characterized by different "subtyping" relations, i.e., partial orders over types (Section 3).

In Section 5 we develop a framework based on Henglein's Coercion Calculus in which we formalize these two axes of the design space and instantiate four variants of the Coercion Calculus, each of which supports blame tracking. Two of the variants extend **HTF-E** and **HTF-L**, respectively, with blame tracking in a natural way. The lazy variant has the same blame assignment behavior as **WF-2**, thereby establishing a previously unknown connection. The other two variants use a new blame tracking strategy in which casts that respect traditional subtyping are guaranteed to never fail. Of these two, the one with eager error detection provides a compelling semantics for gradual typing, as explained in Sections 2 and 3.

In Section 6 we show how the approach of Herman et al. [7] can be applied to obtain a space-efficient reduction strategy for each of these calculi. In doing so, we provide the first space-efficient calculi that also perform blame tracking. We conclude in Section 7.

2 From Lazy to Eager Detection of Higher-Order Cast Errors

The $\lambda^{\langle\cdot\rangle}_{\to}$ cast can be seen as performing two actions: run-time type checking and coercing. Under the *lazy error detection strategy*, no run-time type checking is performed when a higher-order cast is applied to a value. Thus, higher-order casts never fail immediately; they coerce their argument to the target type and defer checking until the argument is applied. Both **HTF-L** and **WF-2** use lazy error detection, which is why neither detects cast errors in programs (1) and (2).

Under the *partially-eager error detection strategy*, a higher-order cast is checked immediately only when its source type is Dyn, otherwise checking is deferred according to the lazy strategy. Both **ST** and **WF-1** use the partially eager error detection strategy. Under this strategy program (1) produces a cast error whereas program (2) does not.

Examples like program (2) inspire the *eager error detection strategy*. Under this strategy, a higher-order cast always performs some checking immediately. Furthermore, the run-time checking is "deep" in that it not only checks that the target type is consistent with the outermost wrapper, but it also checks for consistency at every layer of wrapping including the underlying value type. Thus, when the cast $\langle \texttt{Bool} \rightarrow \texttt{Int} \rangle$ in program (2) is evaluated, it checks that $\texttt{Bool} \rightarrow \texttt{Int}$ is consistent with the prior cast $\langle \texttt{Dyn} \rightarrow \texttt{Dyn} \rangle$ (which it is) and with the type of the underlying function $\texttt{Int} \rightarrow \texttt{Int}$ (which it is not). The **HTF-E** semantics is eager in this sense.

For the authors, the main use of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ is as a target language for the gradually typed lambda calculus, so we seek the most appropriate error detection strategy for gradual typing. With gradual typing, programmers add type annotations to their programs to increase static checking and to express *invariants that they believe to be true about their program*. Thus, when a programmer annotates a parameter with the type $\texttt{Bool} \rightarrow \texttt{Int}$, she is expressing the invariant that all the run-time values bound to this parameter will behave according to the type $\texttt{Bool} \rightarrow \texttt{Int}$. With this in mind, it makes sense that the programmer is notified as soon as possible if the invariant does not hold. The eager error detection strategy does this.

3 Blame Assignment and Subtyping

When programming in a language based on $\lambda^{\langle\cdot\rangle}_{\rightarrow}$, it helps to statically know which parts of the program might cause cast errors and which parts never will. A graphical development environment, for instance, could use colors to distinguish safe casts, unsafe casts which might fail, and inadmissible casts which the system rejects because they always fail. The inadmissible casts are statically detected using the consistency relation $\sim$ of Siek and Taha [8], defined in Fig. 2. A cast $\langle T \Leftarrow S \rangle e$ is rejected if $S \not\sim T$. Safe casts are statically captured by subtyping relations: if the cast *respects subtyping*, meaning $S <: T$, then it is safe. For unsafe casts, $S \sim T$ but $S \not<: T$[1].

$$\frac{}{T \sim \texttt{Dyn}} \qquad \frac{}{\texttt{Dyn} \sim T} \qquad \frac{}{B \sim B} \qquad \frac{S_1 \sim T_1 \qquad S_2 \sim T_2}{S_1 \rightarrow S_2 \sim T_1 \rightarrow T_2}$$

Fig. 2. The consistency relation

Formally establishing that a particular subtype relation is sound with respect to the semantics requires a theorem of the form:

> *If there is a cast at location l of the source program that respects subtyping, then no execution of the program will result in a cast error that blames l.*

Wadler and Findler [10], [11] prove this property for their two semantics and their definitions of subtyping, respectively. Fig. 3 shows their two subtyping relations as well as the traditional subtype relation with Dyn as its top element.

We consider the choice of subtype relation to be a critical design decision because it directly affects the programmer, i.e., it determines which casts are statically safe and

[1] Subtyping is a conservative approximation, so some of the unsafe casts are "false positives" and will never cause cast errors.

Traditional subtyping:

$$\frac{}{T <: \texttt{Dyn}} \qquad \frac{}{B <: B} \qquad \frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$$

Subtyping of **WF-1**:

$$\frac{}{B <: B} \qquad \frac{}{\texttt{Dyn} <: \texttt{Dyn}} \qquad \frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$$

Subtyping of **WF-2**:

$$\frac{}{B <: B} \qquad \frac{}{\texttt{Dyn} <: \texttt{Dyn}} \qquad \frac{S <: G}{S <: \texttt{Dyn}} \qquad \frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$$

where $G ::= B \mid \texttt{Dyn} \to \texttt{Dyn}$

Fig. 3. Three subtyping relations

unsafe. The traditional subtype relation is familiar to programmers, relatively easy to explain, and matches our intuitions about which casts are safe. This raises the question: is there a blame tracking strategy for which the traditional subtype relation is sound?

First, it is instructive to see why traditional subtyping is not sound with respect to the blame tracking in **WF-2** (**WF-1** is similar in this respect). Consider program (3).

$$(\langle\texttt{Dyn} \to \texttt{Int} \Leftarrow \texttt{Dyn}\rangle^{l_3} \langle\texttt{Dyn} \Leftarrow \texttt{Bool} \to \texttt{Bool}\rangle^{l_2} \lambda x : \texttt{Bool}.\ x)\langle\texttt{Dyn} \Leftarrow \texttt{Int}\rangle^{l_1} 1$$

The cast at location l_2 respects the traditional subtyping relation: $\texttt{Bool} \to \texttt{Bool} <: \texttt{Dyn}$. The following reduction sequence uses the blame tracking strategy of **WF-2**. The expression $\texttt{Dyn}_G(v)$ represents values that have been injected into the dynamic type. The subscript G records the type of v and is restricted to base types, the dynamic type, and the function type $\texttt{Dyn} \to \texttt{Dyn}$. Their interpretation of a cast is closer to that of an obligation expression [1], so each blame label has a *polarity*, marked by the presence or absence of an overline, which directs blame toward the interior or exterior of the cast.

$$\begin{array}{ll}
 & (\langle\texttt{Dyn} \to \texttt{Int} \Leftarrow \texttt{Dyn}\rangle^{l_3} \langle\texttt{Dyn} \Leftarrow \texttt{Bool} \to \texttt{Bool}\rangle^{l_2} \lambda x : \texttt{Bool}.\ x)\langle\texttt{Dyn} \Leftarrow \texttt{Int}\rangle^{l_1} 1 \\
\longrightarrow & (\langle\texttt{Dyn} \to \texttt{Int} \Leftarrow \texttt{Dyn}\rangle^{l_3} \texttt{Dyn}_{\texttt{Dyn}\to\texttt{Dyn}}(\langle\ \texttt{Dyn}\ \to \texttt{Dyn} \Leftarrow\ \texttt{Bool}\ \to \texttt{Bool}\rangle^{l_2} \lambda x : \texttt{Bool}.\ x))\langle\texttt{Dyn} \Leftarrow \texttt{Int}\rangle^{l_1} 1 \\
\longrightarrow & (\langle\texttt{Dyn} \to \texttt{Int} \Leftarrow \texttt{Dyn} \to \texttt{Dyn}\rangle^{l_3} \langle\ \texttt{Dyn}\ \to \texttt{Dyn} \Leftarrow\ \texttt{Bool}\ \to \texttt{Bool}\rangle^{l_2} \lambda x : \texttt{Bool}.\ x)\langle\texttt{Dyn} \Leftarrow \texttt{Int}\rangle^{l_1} 1 \\
\longrightarrow & \langle\texttt{Int} \Leftarrow \texttt{Dyn}\rangle^{l_3} ((\langle\ \texttt{Dyn}\ \to \texttt{Dyn} \Leftarrow\ \texttt{Bool}\ \to \texttt{Bool}\rangle^{l_2} \lambda x : \texttt{Bool}.\ x)\langle\texttt{Dyn} \Leftarrow \texttt{Dyn}\rangle^{\overline{l_3}} \langle\texttt{Dyn} \Leftarrow \texttt{Int}\rangle^{l_1} 1 \\
\longrightarrow & \langle\texttt{Int} \Leftarrow \texttt{Dyn}\rangle^{l_3} ((\langle\ \texttt{Dyn}\ \to \texttt{Dyn} \Leftarrow\ \texttt{Bool}\ \to \texttt{Bool}\rangle^{l_2} \lambda x : \texttt{Bool}.\ x)\langle\texttt{Dyn} \Leftarrow \texttt{Int}\rangle^{l_1} 1 \\
\longrightarrow & \langle\texttt{Int} \Leftarrow \texttt{Dyn}\rangle^{l_3} \langle\texttt{Dyn} \Leftarrow \texttt{Bool}\rangle^{l_2} ((\lambda x : \texttt{Bool}.\ x)\ \langle\texttt{Bool} \Leftarrow \texttt{Dyn}\rangle\ ^{\overline{l_2}} \langle\texttt{Dyn} \Leftarrow \texttt{Int}\rangle^{l_1} 1) \\
\longrightarrow & \ \mathbf{blame}\ \overline{l_2}
\end{array}$$

The example shows that under this blame tracking strategy, a cast like l_2 can respect traditional subtyping yet still be blamed. We trace back to the source of the cast error by highlighting the relevant portions of the casts in gray. The source of the cast error is the transition that replaces the cast $\langle\texttt{Dyn} \Leftarrow \texttt{Bool} \to \texttt{Bool}\rangle^{l_2}$ with $\langle\texttt{Dyn} \to \texttt{Dyn} \Leftarrow \texttt{Bool} \to \texttt{Bool}\rangle^{l_2}$. This reduction rule follows from restrictions on the structure of $\mathtt{Dyn}_G(v)$: the only function type allowed for G is $\mathtt{Dyn} \to \mathtt{Dyn}$. This choice forces casts from function types to Dyn to always go through $\mathtt{Dyn} \to \mathtt{Dyn}$. However, adding the intermediate step does not preserve traditional subtyping: $S \to T <: \mathtt{Dyn}$ is always true, but because of the contravariance of subtyping in the argument position, it is not always the case that $S \to T <: \mathtt{Dyn} \to \mathtt{Dyn}$. For instance, if $S = \mathtt{Bool}$, then it is not the case that $\mathtt{Dyn} <: \mathtt{Bool}$.

It seems reasonable, however, to inject higher-order types directly into Dyn. Consider the following alternative injection and and projection rules for Dyn:

$$\begin{array}{rcll}
\langle \mathtt{Dyn} \Leftarrow S\rangle^l v & \longrightarrow_s & \mathtt{Dyn}_S(v) & \\
\langle T \Leftarrow \mathtt{Dyn}\rangle^l \mathtt{Dyn}_S(v) & \longrightarrow_s & \langle T \Leftarrow S\rangle^l v & \text{if } S \sim T \\
\langle T \Leftarrow \mathtt{Dyn}\rangle^l \mathtt{Dyn}_S(v) & \longrightarrow_s & \mathbf{blame}\ l & \text{if } S \not\sim T
\end{array}$$

We define the *simple blame tracking semantics*, written $\longrightarrow_s$, to include the above rules together with AppCst and the standard β and δ reduction rules. The following is the corresponding reduction sequence for program (3).

$$\begin{array}{ll}
 & (\langle \mathtt{Dyn} \to \mathtt{Int} \Leftarrow \mathtt{Dyn}\rangle^{l_3} \langle \mathtt{Dyn} \Leftarrow \mathtt{Bool} \to \mathtt{Bool}\rangle^{l_2} \lambda x : \mathtt{Bool}.\ x)\ \langle \mathtt{Dyn} \Leftarrow \mathtt{Int}\rangle^{l_1} 1 \\
\longrightarrow_s & (\langle \mathtt{Dyn} \to \mathtt{Int} \Leftarrow \mathtt{Dyn}\rangle^{l_3} \mathtt{Dyn}_{\mathtt{Bool}\to\mathtt{Bool}}(\lambda x : \mathtt{Bool}.\ x))\ \langle \mathtt{Dyn} \Leftarrow \mathtt{Int}\rangle^{l_1} 1 \\
\longrightarrow_s & (\langle \mathtt{Dyn} \to \mathtt{Int} \Leftarrow \mathtt{Bool} \to \mathtt{Bool}\rangle^{l_3} (\lambda x : \mathtt{Bool}.\ x))\ \langle \mathtt{Dyn} \Leftarrow \mathtt{Int}\rangle^{l_1} 1 \\
\longrightarrow_s & (\langle \mathtt{Dyn} \to \mathtt{Int} \Leftarrow \mathtt{Bool} \to \mathtt{Bool}\rangle^{l_3} (\lambda x : \mathtt{Bool}.\ x))\ \mathtt{Dyn}_{\mathtt{Int}}(1) \\
\longrightarrow_s & \langle \mathtt{Int} \Leftarrow \mathtt{Bool}\rangle^{l_3} ((\lambda x : \mathtt{Bool}.\ x)\ \langle \mathtt{Bool} \Leftarrow \mathtt{Dyn}\rangle^{l_3} \mathtt{Dyn}_{\mathtt{Int}}(1)) \\
\longrightarrow_s & \mathbf{blame}\ l_3
\end{array}$$

Under this blame tracking strategy, the downcast from Dyn to $\mathtt{Dyn} \to \mathtt{Int}$ at location l_3 is blamed instead of the upcast at location l_2. This particular result better matches our intuitions about what went wrong, and in general the simple blame strategy never blames a cast that respects the traditional subtype relation.

Theorem 1 (Soundness of subtyping wrt. the simple semantics)
If there is a cast labeled l in program e that respects subtyping, then $e \not\longrightarrow_s^* \mathbf{blame}\ l$.

Proof. The proof is a straightforward induction on $\longrightarrow_s^*$ once the statement is generalized to say "all casts labeled l". This is necessary because the AppCst rule turns one cast into two casts with the same label.

While the simple blame tracking semantics assigns blame in a way that respects traditional subtyping, it does not perform eager error detection; it is partially eager. We conjecture that the simple semantics could be augmented with deep checks to achieve eager error detection. However, there also remains the issue of space efficiency. In the next section we discuss the problems regarding space efficiency and how these problems can be solved by moving to a framework based on the semantics of Herman et al. [7] which in turn uses the Coercion Calculus of Henglein [6]. We then show how the variations in blame tracking and eager checking can be realized in that framework.

4 Space Efficiency

Herman et al. [7] observe two circumstances where the wrappers used for higher-order casts can lead to unbounded space consumption. First, some programs repeatedly apply casts to the same function, resulting in a build-up of wrappers. In the following example, each time the function bound to `k` is passed between `even` and `odd` a wrapper is added, causing a space leak proportional to `n`.

```
let rec even(n : Int, k : Dyn→Bool) : Bool =
   if (n = 0) then k(⟨Dyn ⇐ Bool⟩True)
   else odd(n - 1, ⟨Bool → Bool ⇐ Dyn → Bool⟩k)
and odd(n : Int, k : Bool→Bool) : Bool =
   if (n = 0) then k(False)
   else even(n - 1, ⟨Dyn → Bool ⇐ Bool → Bool⟩k)
```

Second, some casts break proper tail recursion. Consider the following example in which the return type of `even` is `Dyn` and `odd` is `Bool`.

```
let rec even(n : Int) : Dyn =
  if (n = 0) then True else ⟨Dyn ⇐ Bool⟩odd(n - 1)
and odd(n : Int) : Bool =
  if (n = 0) then False else ⟨Bool ⇐ Dyn⟩even(n - 1)
```

Assuming tail call optimization, cast-free versions of the `even` and `odd` functions require only constant space, but because the call to `even` is no longer a tail call, the run-time stack grows with each call and space consumption is proportional to `n`. The following reduction sequence for a call to `even` shows the unbounded growth.

$$\begin{aligned}
\mathtt{even}(n) &\longmapsto \langle \mathtt{Dyn} \Leftarrow \mathtt{Bool}\rangle \mathtt{odd}(n-1) \\
&\longmapsto \langle \mathtt{Dyn} \Leftarrow \mathtt{Bool}\rangle\langle \mathtt{Bool} \Leftarrow \mathtt{Dyn}\rangle \mathtt{even}(n-2) \\
&\longmapsto \langle \mathtt{Dyn} \Leftarrow \mathtt{Bool}\rangle\langle \mathtt{Bool} \Leftarrow \mathtt{Dyn}\rangle\langle \mathtt{Dyn} \Leftarrow \mathtt{Bool}\rangle \mathtt{odd}(n-3) \\
&\longmapsto \cdots
\end{aligned}$$

Herman et al. [7] show that space efficiency can be recovered by 1) merging sequences of casts into a single cast, 2) ensuring that the size of a merged cast is bounded by a constant, and 3) checking for sequences of casts in tail-position and merging them before making function calls.

5 Variations on the Coercion Calculus

The semantics of $\lambda^{\langle\cdot\rangle}_{\rightarrow}$ in Henglein [6] and subsequently in Herman et al. [7] use a special sub-language called the Coercion Calculus to express casts. Instead of casts of the form $\langle T \Leftarrow S\rangle e$ they have casts of the form $\langle c\rangle e$ where c is a coercion expression. The Coercion Calculus can be viewed as a fine-grained operational specification of casts. It is not intended to be directly used by programmers, but instead casts of the form $\langle T \Leftarrow S\rangle e$ are compiled into casts of the form $\langle c\rangle e$. In this section we define a translation function $\langle\!\langle T \Leftarrow S\rangle\!\rangle$ that maps the source and target of a cast to a coercion. We define $\langle\!\langle e\rangle\!\rangle$ to be the natural extension of this translation to expressions. The syntax and type system of the Coercion Calculus is shown in Fig. 4. We add blame labels to the syntax to introduce blame tracking to the Coercion Calculus.

The coercion $T!$ injects a value into `Dyn` whereas the coercion $T?$ projects a value out of `Dyn`. For example, the coercion `Int`! takes an integer and injects it into the type `Dyn`, and conversely, the coercion `Int`$?^l$ takes a value of type `Dyn` and projects it to

Syntax: Coercions $c, d ::= \iota \mid T! \mid T?^l \mid c \rightarrow d \mid d \circ c \mid \texttt{Fail}^l$

Coercion contexts $C ::= \square \mid C \rightarrow c \mid c \rightarrow C \mid c \circ C \mid C \circ c$

Type system:

$$\frac{}{\vdash \iota : T \Leftarrow T} \qquad \frac{}{\vdash T! : \texttt{Dyn} \Leftarrow T} \qquad \frac{}{\vdash T?^l : T \Leftarrow \texttt{Dyn}} \qquad \frac{}{\vdash \texttt{Fail}^l : T \Leftarrow S}$$

$$\frac{\vdash c : S_1 \Leftarrow T_1 \quad \vdash d : T_2 \Leftarrow S_2}{\vdash c \rightarrow d : (T_1 \rightarrow T_2) \Leftarrow (S_1 \rightarrow S_2)} \qquad \frac{\vdash d : T_3 \Leftarrow T_2 \quad \vdash c : T_2 \Leftarrow T_1}{\vdash d \circ c : T_3 \Leftarrow T_1}$$

Fig. 4. Syntax and type system for the Coercion Calculus

type Int, checking to make sure the value is an integer, blaming location l otherwise. Our presentation of injections and projections differs from that of Henglein [6] in that the grammar in Fig. 4 allows arbitrary types in $T!$ and $T?^l$. When modeling Henglein's semantics, we restrict T to base types and function types of the form $\texttt{Dyn} \rightarrow \texttt{Dyn}$. Thus, $(\texttt{Dyn} \rightarrow \texttt{Dyn})!$ is equivalent to Henglein's Func! and $(\texttt{Dyn} \rightarrow \texttt{Dyn})?^l$ is equivalent to Func?. The calculus also has operators for aggregating coercions. The function coercion $c \rightarrow d$ applies coercion c to a function's argument and d to its return value. Coercion composition $d \circ c$ applies coercion c then coercion d.[2] In addition to Henglein's coercions, we adopt the $\texttt{Fail}^l$ coercion of Herman et al. [7], which compactly represents coercions that are destined to fail but have not yet been applied to a value.

In this section we add blame tracking to the Coercion Calculus using two different blame assignment strategies, one that shares the blame between higher-order upcasts and downcasts, thereby modeling **WF-1** and **WF-2**, and a new strategy that places responsibility on downcasts only. To clearly express not only these two blame assignment strategies but also the existing strategies for eager and lazy error detection (**HTF-E** and **HTF-L**), we organize the reduction rules into four sets that can be combined to create variations on the Coercion Calculus.

L The core set of reduction rules that is used by all variations. This set of rules, when combined with either **UD** or **D**, performs *lazy* error detection.
E The additional rules needed to perform *eager* error detection.
UD The rules for blame assignment that share responsibility between higher-order *upcasts* and *downcasts*.
D The rules for blame assignment that place all the responsibility on *downcasts*.

Fig 5 shows how the sets of reduction rules can be combined to create four distinct coercion calculi.

All of the reduction strategies share the following parameterized rule for single-step evaluation, where X stands for a set of reduction rules.

$$\frac{c \cong C[c_1] \quad c_1 \longrightarrow_X c_2 \quad C[c_2] \cong c'}{c \longmapsto_X c'}$$

[2] We use the notation $d \circ c$ instead of the notation $c; d$ of Henglein to be consistent with the right to left orientation of our type-based cast expressions.

	Lazy error detection	Eager error detection
Blame upcasts and downcasts	$\mathbf{L} \cup \mathbf{UD}$	$\mathbf{L} \cup \mathbf{UD} \cup \mathbf{E}$
Blame downcasts	$\mathbf{L} \cup \mathbf{D}$	$\mathbf{L} \cup \mathbf{D} \cup \mathbf{E}$

Fig. 5. Summary of the Coercion Calculi

The above rule relies on a congruence relation, written $\cong$, to account for the associativity of coercion composition: $(c_3 \circ c_2) \circ c_1 \cong c_3 \circ (c_2 \circ c_1)$. The reduction rules simplify pairs of adjacent coercions into a single coercion. The congruence relation is used during evaluation to reassociate a sequence of coercions so that a pair of adjacent coercions can be reduced. A coercion c is *normalized* if $\nexists c'.\ c \longmapsto_X c'$ and we indicate that a coercion is normalized with an overline, as in $\overline{c}$.

$$
\begin{array}{rclrcl}
B?^l \circ B! & \longrightarrow & \iota & B'?^l \circ B! & \longrightarrow & \mathtt{Fail}^l \\
\iota \to \iota & \longrightarrow & \iota & B?^l \circ (S_1 \to S_2)! & \longrightarrow & \mathtt{Fail}^l \\
c \circ \iota & \longrightarrow & c & (T_1 \to T_2)?^l \circ B! & \longrightarrow & \mathtt{Fail}^l \\
\iota \circ c & \longrightarrow & c & d \circ \mathtt{Fail}^l & \longrightarrow & \mathtt{Fail}^l \\
 & & & \mathtt{Fail}^l \circ T! & \longrightarrow & \mathtt{Fail}^l \quad (\textsc{FailIn})
\end{array}
$$
$$(d_1 \to d_2) \circ (c_1 \to c_2) \longrightarrow (c_1 \circ d_1) \to (d_2 \circ c_2)$$

Fig. 6. The core reduction rules (**L**)

The set **L** of core reduction rules is given in Fig. 6. These rules differ from those of Herman et al. [7] in several ways. First, the rules propagate blame labels. Second, we factor the rule for handling injection-projection pairs over functions types into **UD** and **D**. Third, we omit a rule of the form

$$(\textsc{FailL}) \qquad \mathtt{Fail}^l \circ c \longrightarrow \mathtt{Fail}^l$$

This change is motivated by the addition of blame tracking which makes it possible to distinguish between failures with different causes. If we use FAILL, then the optimizations for space efficiency change how blame is assigned. Suppose there is a context waiting on the stack of the form $\langle \mathtt{Fail}^{l_2} \circ \mathtt{Int?}^{l_1} \rangle \square$ and the value returned to this context is $\langle \mathtt{Bool!} \rangle \mathtt{True}$. Then in the un-optimized semantics we have

$$\langle \mathtt{Fail}^{l_2} \circ \mathtt{Int?}^{l_1} \rangle \square \longmapsto \langle \mathtt{Fail}^{l_2} \circ \mathtt{Int?}^{l_1} \rangle \langle \mathtt{Bool!} \rangle \mathtt{True} \longmapsto \langle \mathtt{Fail}^{l_2} \circ \mathtt{Int?}^{l_1} \circ \mathtt{Bool!} \rangle \mathtt{True}$$
$$\longmapsto \langle \mathtt{Fail}^{l_2} \circ \mathtt{Fail}^{l_1} \rangle \mathtt{True} \longmapsto \langle \mathtt{Fail}^{l_1} \rangle \mathtt{True} \longmapsto \mathbf{blame}\ l_1$$

whereas in the optimized semantics we have

$$\langle \mathtt{Fail}^{l_2} \circ \mathtt{Int?}^{l_1} \rangle \square \longmapsto \langle \mathtt{Fail}^{l_2} \rangle \square \longmapsto \langle \mathtt{Fail}^{l_2} \rangle \langle \mathtt{Bool!} \rangle \mathtt{True} \longmapsto \langle \mathtt{Fail}^{l_2} \circ \mathtt{Bool!} \rangle \mathtt{True}$$
$$\longmapsto \langle \mathtt{Fail}^{l_2} \rangle \mathtt{True} \longmapsto \mathbf{blame}\ l_2$$

On the other hand, the rule FAILIN is harmless because an injection can never fail. When we embed a coercion calculus in $\lambda^{\langle\cdot\rangle}_{\to}$, we do not want an expression such as

Syntax:

Expressions	$e ::= x \mid k \mid \lambda x : T.\ e \mid e\ e \mid \langle c \rangle e$	
Simple Values	$s ::= k \mid \lambda x : T.\ e$	
Regular Coercions	$\hat{c} ::= \overline{c}$	where $\overline{c} \neq \iota$ and $\overline{c} \neq \mathtt{Fail}^l$
Values	$v ::= s \mid \langle \hat{c} \rangle s$	
Evaluation contexts	$E ::= \Box \mid E\ e \mid v\ E \mid \langle c \rangle E$	

Type system:

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash k : \mathit{typeof}(k)} \qquad \frac{\Gamma[x \mapsto S] \vdash e : T}{\Gamma \vdash \lambda x : S.\ e : S \to T}$$

$$\frac{\Gamma \vdash e_1 : S \to T \quad \Gamma \vdash e_2 : S}{\Gamma \vdash e_1\ e_2 : T} \qquad \frac{\vdash c : T \Leftarrow S \quad \Gamma \vdash e : S}{\Gamma \vdash \langle c \rangle e : T}$$

Reduction rules:

(β)	$(\lambda x : T.e)\ v$	$\longrightarrow$	$e[x \mapsto v]$	
(δ)	$k\ v$	$\longrightarrow$	$\delta(k, v)$	
(StepCst)	$\langle c \rangle s$	$\longrightarrow$	$\langle c' \rangle s$	if $c \longmapsto_X c'$
(IdCst)	$\langle \iota \rangle s$	$\longrightarrow$	s	
(CmpCst)	$\langle d \rangle \langle \hat{c} \rangle s$	$\longrightarrow$	$\langle d \circ \hat{c} \rangle s$	
(AppCst)	$\langle \overline{c} \to \overline{d} \rangle s\ v$	$\longrightarrow$	$\langle d \rangle (s\ \langle c \rangle v)$	
(FailCst)	$\langle \mathtt{Fail}^l \rangle s$	$\longrightarrow$	$\mathbf{blame}\ l$	
(FailFC)	$\langle \mathtt{Fail}^l \circ (\overline{c} \to \overline{d}) \rangle s$	$\longrightarrow$	$\mathbf{blame}\ l$	

Single-step evaluation:

$$\frac{e \longrightarrow e}{E[e] \longmapsto E[e']} \qquad \frac{e \longrightarrow \mathbf{blame}\ l}{E[e] \longmapsto \mathbf{blame}\ l}$$

Fig. 7. A semantics for $\lambda^{\langle\cdot\rangle}_{\to}$ based on coercion calculi

$\langle \mathtt{Fail}^l \circ (\iota \to \mathtt{Int!}) \rangle (\lambda x : \mathtt{Int}.\ x)$ to be a value. It should instead reduce to **blame** l. Instead of trying to solve this in the coercion calculi, we add a reduction rule (FailFC) to $\lambda^{\langle\cdot\rangle}_{\to}$ to handle this situation.

Fig. 7 shows a semantics for $\lambda^{\langle\cdot\rangle}_{\to}$ based on coercion calculi (it is parameterized on the set of coercion reduction rules X). We write $\lambda^{\langle\cdot\rangle}_{\to}(X)$ to refer to an instantiation of the semantics with the coercion calculus X. The semantics includes the usual rules for the lambda calculus and several rules that govern the behavior of casts. The rule StepCst simplifies a cast expression by taking one step of evaluation inside the coercion. The rule IdCst discards an identity cast and CmpCst turns a pair of casts into a single cast with composed coercions. The AppCst rule applies a function wrapped in a cast. The cast is split into a cast on the argument and a cast on the return value. The rule FailCst signals a cast error when the coercion is $\mathtt{Fail}^l$.

5.1 Blame Assignment Strategies

In this section we present two blame assignment strategies: the strategy shared by **WF-1** and **WF-2**, where upcasts and downcasts share responsibility for blame, and a new strategy where only downcasts are responsible for blame. The first strategy will be modeled by the set of reduction rules **UD** (for upcast-downcast) and the second by the set of reduction rules **D** (for downcast).

The UD Blame Assignment Strategy. As discussed in Section 3, the blame assignment strategy that shares responsibility for blame between upcasts and downcasts is based on the notion that a cast between an arbitrary function type and Dyn must always go through $\texttt{Dyn} \to \texttt{Dyn}$. As a result, at the level of the coercion calculus, the only higher-order injections and projections are $(\texttt{Dyn} \to \texttt{Dyn})!$ and $(\texttt{Dyn} \to \texttt{Dyn})?^l$. The compilation of type-based casts to coercion-based casts is responsible for introducing the indirection through $\texttt{Dyn} \to \texttt{Dyn}$. Fig. 8 shows the compilation function. The last two lines of the definition handle higher-order upcasts and downcasts and produce coercions that go through $\texttt{Dyn} \to \texttt{Dyn}$. Consider the coercion produced from the higher-order cast that injects $\texttt{Bool} \to \texttt{Bool}$ into Dyn.

$$\langle\!\langle \texttt{Dyn} \Leftarrow \texttt{Bool} \to \texttt{Bool} \rangle\!\rangle^l = (\texttt{Dyn} \to \texttt{Dyn})! \circ (\texttt{Bool}?^l \to \texttt{Bool!})$$

The projection $\texttt{Bool}?^l$ in the resulting coercion can cause a run-time cast error, which shows how, with this blame assignment strategy, higher-order upcasts such as $\texttt{Dyn} \Leftarrow \texttt{Bool} \to \texttt{Bool}$ share the responsibility for cast errors.

The set of reduction rules for **UD** is given in Fig. 8. With **UD**, the only higher-order coercions are to and from $\texttt{Dyn} \to \texttt{Dyn}$, so the only injection-projection case missing from the **L** rules is the case handled by the INOUTDD rule in Fig. 8.

The combination $\mathbf{L} \cup \mathbf{UD}$ simulates the semantics of **WF-2**.

Theorem 2. *If* $e \longrightarrow^* e'$ *in* ***WF-2*** *and* e' *is a value or* **blame** l*, then there is an* e'' *such that* $\langle\!\langle e \rangle\!\rangle \longmapsto^*_{\mathbf{L}\cup\mathbf{UD}} e''$ *and* $\langle\!\langle e' \rangle\!\rangle = e''$.

Proof. The proof is a straightforward induction on $\longrightarrow^*$.

The combination $\mathbf{L} \cup \mathbf{UD}$ can also be viewed as the natural way to add blame tracking to **HTF-L**, revealing an interesting and new connection between **HTF-L** and **WF-2**.

The D Blame Assignment Strategy. To obtain a blame assignment strategy that coincides with traditional subtyping, we lift the restriction on injections and projections to allow direct coercions between arbitrary function types and Dyn, analogous to what we did in Section 3. With this change the compilation from type-based casts to coercions no longer needs to go through the intermediate $\texttt{Dyn} \to \texttt{Dyn}$. Fig. 9 shows the new compilation function. Consequently the reduction rules for **D** need to be more general to handle arbitrary higher-order projections and injections. The rule INOUTFF in Fig. 9 does just that. The blame label l used to create the coercion on the right-hand side is from the projection. This places all of the responsibility for a potential error on the projection.

We now prove that the **D** strategy fulfills its design goal: traditional subtyping should capture the notion of a safe cast, i.e., a cast that is guaranteed not to be blamed for any run-time cast errors. It turns out that this is rather straightforward to prove because a cast from S to T, where $S <: T$, compiles to a coercion with no projection or failure coercions. In fact, the coercion will not contain any blame labels.

Compilation from type-based casts to coercions:

$$
\begin{array}{lll}
\langle\!\langle B \Leftarrow B \rangle\!\rangle^l & = \iota & \\
\langle\!\langle B' \Leftarrow B \rangle\!\rangle^l & = \mathtt{Fail}^l & \text{if } B \neq B' \\
\langle\!\langle \mathtt{Dyn} \Leftarrow \mathtt{Dyn} \rangle\!\rangle^l & = \iota & \\
\langle\!\langle \mathtt{Dyn} \Leftarrow B \rangle\!\rangle^l & = B! & \\
\langle\!\langle B \Leftarrow \mathtt{Dyn} \rangle\!\rangle^l & = B?^l & \\
\langle\!\langle B \Leftarrow S_1 \to S_2 \rangle\!\rangle^l & = \mathtt{Fail}^l & \\
\langle\!\langle T_1 \to T_2 \Leftarrow B \rangle\!\rangle^l & = \mathtt{Fail}^l & \\
\langle\!\langle T_1 \to T_2 \Leftarrow S_1 \to S_2 \rangle\!\rangle^l & = \begin{cases} \iota & \text{if } c = \iota \text{ and } d = \iota \\ \mathtt{Fail}^l & \text{if } c = \mathtt{Fail}^l \text{ or } d = \mathtt{Fail}^l \\ c \to d & \text{otherwise} \end{cases} & \\
& \text{where } c = \langle\!\langle S_1 \Leftarrow T_1 \rangle\!\rangle^l, d = \langle\!\langle T_2 \Leftarrow S_2 \rangle\!\rangle^l & \\
\langle\!\langle \mathtt{Dyn} \Leftarrow S_1 \to S_2 \rangle\!\rangle^l & = (\mathtt{Dyn} \to \mathtt{Dyn})! \circ \langle\!\langle \mathtt{Dyn} \to \mathtt{Dyn} \Leftarrow S_1 \to S_2 \rangle\!\rangle^l & \\
\langle\!\langle T_1 \to T_2 \Leftarrow \mathtt{Dyn} \rangle\!\rangle^l & = \langle\!\langle T_1 \to T_2 \Leftarrow \mathtt{Dyn} \to \mathtt{Dyn} \rangle\!\rangle^l \circ (\mathtt{Dyn} \to \mathtt{Dyn})?^l &
\end{array}
$$

Reduction rules:

$$(\textsc{InOutDD})\ (\mathtt{Dyn} \to \mathtt{Dyn})?^l \circ (\mathtt{Dyn} \to \mathtt{Dyn})! \longrightarrow \iota$$

Fig. 8. The **UD** blame assignment strategy

Lemma 1 (Subtype coercions do not contain blame labels)
If $S <: T$ then $labels(\langle\!\langle T \Leftarrow S \rangle\!\rangle^l) = \emptyset$, where $labels(c)$ is the labels that occur in c.

Proof. The proof is by strong induction on the sum of the height of the two types followed by case analysis on the types. Each case is straightforward.

Next we show that coercion evaluation does not introduce blame labels.

Lemma 2 (Coercion evaluation monotonically decreases labels)
If $c \longmapsto_{\mathbf{L} \cup \mathbf{D}} c'$ then $labels(c') \subseteq labels(c)$.

Proof. The proof is a straightforward case analysis on $\longmapsto_{\mathbf{L} \cup \mathbf{D}}$ and $\longrightarrow_{\mathbf{L} \cup \mathbf{D}}$.

The same can be said of $\lambda^{\langle\cdot\rangle}_{\to}(\mathbf{L} \cup \mathbf{D})$ evaluation.

Lemma 3 ($\lambda^{\langle\cdot\rangle}_{\to}$(L ∪ D) evaluation monotonically decreases labels)
If $e \longmapsto e'$ then $labels(e') \subseteq labels(e)$, where $labels(e)$ is the labels that occur in e.

Proof. The proof is a straightforward case analysis on $\longmapsto$ and $\longrightarrow$.

Thus, when a cast failure occurs, the label must have come from a coercion in the original program, but it could not have been from a cast that respects subtyping because such casts produce coercions that do not contain any blame labels.

Theorem 3 (Soundness of subtyping wrt. $\lambda^{\langle\cdot\rangle}_{\to}$(L ∪ D))
If every cast labeled l in program e respects subtyping, then $e \not\longmapsto^* \mathbf{blame}\ l$.

Proof. The proof is a straightforward induction on $\longmapsto^*$.

Compilation from type-based casts to coercions:

$$\begin{aligned} &\vdots \text{ (same as in Fig. 8)} \\ \langle\!\langle \mathtt{Dyn} \Leftarrow S_1 \to S_2 \rangle\!\rangle^l &= (S_1 \to S_2)! \\ \langle\!\langle T_1 \to T_2 \Leftarrow \mathtt{Dyn} \rangle\!\rangle^l &= (T_1 \to T_2)?^l \end{aligned}$$

Reduction rules:

$$(\textsc{InOutFF})\ (T_1 \to T_2)?^l \circ (S_1 \to S_2)! \longrightarrow \langle\!\langle T_1 \to T_2 \Leftarrow S_1 \to S_2 \rangle\!\rangle^l$$

Fig. 9. The **D** blame assignment strategy

5.2 An Eager Error Detection Strategy for the Coercion Calculus

To explain the eager error detection strategy in **HTF-E**, we first review why the reduction rules in **L** detect higher-order cast errors in a lazy fashion. Consider the reduction sequence for program (2) under $\mathbf{L} \cup \mathbf{D}$:

$$\begin{aligned} &\langle\!\langle \langle \mathtt{Bool} \to \mathtt{Int} \Leftarrow \mathtt{Dyn} \to \mathtt{Dyn} \rangle^{l_2} \langle \mathtt{Dyn} \to \mathtt{Dyn} \Leftarrow \mathtt{Int} \to \mathtt{Int} \rangle^{l_1} (\lambda x : \mathtt{Int}.\ x) \rangle\!\rangle \\ = \ &\langle \mathtt{Bool!} \to \mathtt{Int?}^{l_2} \rangle \langle \mathtt{Int?}^{l_1} \to \mathtt{Int!} \rangle (\lambda x : \mathtt{Int}.\ x) \\ \longmapsto \ &\langle (\mathtt{Bool!} \to \mathtt{Int?}^{l_2}) \circ (\mathtt{Int?}^{l_1} \to \mathtt{Int!}) \rangle (\lambda x : \mathtt{Int}.\ x) \\ \longmapsto \ &\langle (\mathtt{Int?}^{l_1} \circ \mathtt{Bool!}) \to (\mathtt{Int?}^{l_2} \circ \mathtt{Int!}) \rangle (\lambda x : \mathtt{Int}.\ x) \\ \longmapsto \ &\langle \mathtt{Fail}^{l_1} \to \iota \rangle (\lambda x : \mathtt{Int}.\ x) \end{aligned}$$

The cast $\langle \mathtt{Fail}^{l_1} \to \iota \rangle$ is in normal form and, because the $\mathtt{Fail}^{l_1}$ does not propagate to the top of the cast, the lazy reduction strategy does not signal a failure in this case.

The eager error detection strategy therefore adds reduction rules that propagate failures up through function coercions. Fig 10 shows the two reduction rules for the **E** strategy. Note that in FAILFR we require the coercion in argument position to be normalized but not be a failure. This restriction is needed for confluence.

Using the eager reduction rules, program (2) produces a cast error.

$$\cdots \longmapsto \langle \mathtt{Fail}^{l_1} \to \iota \rangle (\lambda x : \mathtt{Int}.\ x) \longmapsto \langle \mathtt{Fail}^{l_1} \rangle (\lambda x : \mathtt{Int}.\ x) \longmapsto \mathbf{blame}\ l_1$$

The combination $\mathbf{L} \cup \mathbf{UD} \cup \mathbf{E}$ can be viewed as adding blame tracking to **HTF-E**. The combination $\mathbf{L} \cup \mathbf{D} \cup \mathbf{E}$ is entirely new and particularly appealing as an intermediate language for gradual typing because it provides thorough error detection and intuitive blame assignment. The combination $\mathbf{L} \cup \mathbf{D}$ is well-suited to modeling languages that combine dynamic and static typing in a manner that admits as many correctly-behaving programs as possible because it avoids reporting cast errors until they are immediately relevant and provides intuitive guidance when failure occurs.

$$\begin{aligned} &(\textsc{FailFL})\ (\mathtt{Fail}^l \to d) \longrightarrow \mathtt{Fail}^l \\ &(\textsc{FailFR})\ (\overline{c} \to \mathtt{Fail}^l) \longrightarrow \mathtt{Fail}^l \qquad \text{where } \overline{c} \neq \mathtt{Fail}^{l'} \end{aligned}$$

Fig. 10. The eager detections reduction rules (**E**)

6 A Space-Efficient Semantics for $\lambda^{\langle\cdot\rangle}_{\rightarrow}(X)$

The semantics of $\lambda^{\langle\cdot\rangle}_{\rightarrow}(X)$ given in Fig. 7 is only partially space-efficient. Rule CmpCst merges adjacent casts and the normalization of coercions sufficiently compresses them, but casts can still accumulate in the tail position of recursive calls. It is straightforward to parameterize the space-efficient semantics and proofs of Herman et al. [7] with respect to coercion calculi, thereby obtaining a space-efficient semantics for $\lambda^{\langle\cdot\rangle}_{\rightarrow}(X)$.

The proof of space-efficiency requires several properties that depend on the choice of X. First, the size of a coercion (number of AST nodes) in normal form must be bounded by its height.

Lemma 4 (Coercion size bounded by height)
For each coercion calculus X in this paper, if $\vdash c : T \Leftarrow S$ and c is in normal form for X, then $\mathit{size}(c) \leq 5(2^{\mathit{height}(c)} - 1)$.

Proof. The proofs are by structural induction on c. In the worst-case, c has the form $(\mathtt{Dyn} \rightarrow \mathtt{Dyn})! \circ (c_1 \rightarrow c_2) \circ (\mathtt{Dyn} \rightarrow \mathtt{Dyn})?$. Thus, $\mathit{size}(c) = 5 + \mathit{size}(c_1) + \mathit{size}(c_2)$. Applying the induction hypothesis to $\mathit{size}(c_1)$ and $\mathit{size}(c_2)$, we have
$\mathit{size}(c) \leq 5 + 2 \cdot 5(2^{\mathit{height}(c)-1} - 1) = 5(2^{\mathit{height}(c)} - 1)$.

Second, the height of the coercions produced by compilation from type-based casts is bounded by the the sum of the source and target type.

Lemma 5. *If $c = \langle\langle T \Leftarrow S \rangle\rangle^l$ then $\mathit{height}(c) \leq \mathit{max}(\mathit{height}(S), \mathit{height}(T))$.*

Third, coercion evaluation must never increase the height of a coercion.

Lemma 6 (Coercion height never increases)
For each coercion calculi X in this paper, if $c \longmapsto_X c'$ then $\mathit{height}(c') \leq \mathit{height}(c)$.

7 Conclusion and Future Work

In this paper we explore the design space of higher-order casts along two axes: blame assignment strategies and eager versus lazy error detection. This paper introduces a framework based on Henglein's Coercion Calculus and instantiates four variants, each of which supports blame tracking and guarantees space efficiency. Of the four variants, one extends the semantics of Herman et al. [7] with blame tracking in a natural way. This variant has the same blame tracking behavior as [11], thereby establishing a previously unknown connection between these works. One of the variants combines eager error detection with a blame tracking strategy in which casts that respect traditional subtyping are guaranteed to never fail. This variant provides a compelling dynamic semantics for gradual typing.

Our account of the design space for cast calculi introduces new open problems. The **UD** blame strategy has a constant-factor speed advantage over the **D** strategy because **D** must generate coercions dynamically. We would like an implementation model for **D** that does not need to generate coercions. We are also interested in characterizations of statically safe casts that achieve greater precision than subtyping.

Bibliography

[1] Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: ACM International Conference on Functional Programming (October 2002)

[2] Flanagan, C.: Hybrid type checking. In: POPL 2006: The 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, pp. 245–256 (January 2006)

[3] Flanagan, C., Freund, S.N., Tomb, A.: Hybrid types, invariants, and refinements for imperative objects. In: FOOL/WOOD2006: International Workshop on Foundations and Developments of Object-Oriented Languages (2006)

[4] Gronski, J., Flanagan, C.: Unifying hybrid types and contracts. In: Trends in Functional Prog. (TFP) (2007)

[5] Henglein, F.: Dynamic typing. In: Krieg-Brückner, B. (ed.) ESOP 1992. LNCS, vol. 582, pp. 233–253. Springer, Heidelberg (1992)

[6] Henglein, F.: Dynamic typing: syntax and proof theory. Science of Computer Programming 22(3), 197–230 (1994)

[7] Herman, D., Tomb, A., Flanagan, C.: Space-efficient gradual typing. In: Trends in Functional Prog. (TFP), p. XXVIII (April 2007)

[8] Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop, pp. 81–92 (September 2006)

[9] Siek, J.G., Taha, W.: Gradual typing for objects. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 2–27. Springer, Heidelberg (2007)

[10] Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: Workshop on Scheme and Functional Programming, pp. 15–26 (2007)

[11] Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 1–16. Springer, Heidelberg (2009)