Threesomes, With and Without Blame *

Jeremy G. Siek
University of Colorado at Boulder
jeremy.siek@colorado.edu

Philip Wadler
University of Edinburgh
wadler@inf.ed.ac.uk

Abstract

How to integrate static and dynamic types? Recent work focuses on casts to mediate between the two. However, adding casts may degrade tail calls into a non-tail calls, increasing space consumption from constant to linear in the depth of calls.

We present a new solution to this old problem, based on the notion of a threesome. A cast is specified by a source and a target type—a twosome. Any twosome factors into a downcast from the source to an intermediate type, followed by an upcast from the intermediate to the target—a threesome. Any chain of threesomes collapses to a single threesome, calculated by taking the greatest lower bound of the intermediate types. We augment this solution with blame labels to map any failure of a threesome back to the offending twosome in the source program.

Herman, Tomb, and Flanagan (2007) solve the space problem by representing casts with the coercion calculus of Henglein (1994). While they provide a theoretical limit on the space overhead, there remains the practical question of how best to implement coercion reduction. The threesomes presented in this paper provide a streamlined data structure and algorithm for representing and normalizing coercions. Furthermore, threesomes provide a typed-based explanation of coercion reduction.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Procedures, functions, and subroutines

General Terms Languages, Theory

Keywords casts, coercions, blame tracking, lambda-calculus

1. Introduction

The old question of how to mix static and dynamic typing is attracting renewed interest. On one side, Hejlsberg (2008) brings type `dynamic` to C# 4.0, and on the other side, Tobin-Hochstadt and Felleisen (2008) integrate static types into Scheme and Wall (2009) adds optional static types to Perl 6. In these mixed settings, programmers and compilers should still be able to trust the results of the static type checker, so run-time checks are needed to safeguard the invariants established by the static type system. Recent work mediates between static and dynamic regions using casts.

Building on the higher-order contracts of Findler and Felleisen (2002), Wadler and Findler (2009) design the *blame calculus* to serve as an intermediate language that integrates static and dynamic typing. The blame calculus earns its name by tracking blame: it maps run-time type errors back to their origin in the source program. The Blame Theorem asserts that statically typed regions of a program can never be blamed for run-time type errors.

However, there is concern that the casts used in the blame calculus impose too much run-time overhead. Findler and Felleisen (2002) observed that contracts may degrade a tail call into a non-tail call and Herman et al. (2007) noted that the same is true for casts. This concern prompted the ECMAScript 4.0 committee (Hansen 2007) and the designers of Thorn (Wrigstad et al. 2009) to consider compromises such as *like types* that do not require casts.

Herman et al. (2007) use the *coercion calculus* of Henglein (1992, 1994) to represent and compress sequences of casts. Any coercion normalizes to a coercion of bounded size, thereby limiting the run-time space overhead to a constant factor. Siek et al. (2009) augment the coercion calculus with blame tracking to obtain a space-efficient implementation of the blame calculus.

In this paper we present a new solution to the space problem, based on the notion of a threesome. Traditionally, a cast is specified by a source and a target type—a twosome. We show that any twosome factors into a downcast from the source to an intermediate type, followed by an upcast from the intermediate to the target—a threesome. We also show that any chain of threesomes collapses to a single threesome, calculated by taking the greatest lower bound of the intermediate types. We then augment this solution with blame labels so as to map any failure of a threesome back to the offending twosome in the source program.

Threesomes are designed to correspond to twosomes and twosomes are designed to correspond to Henglein's coercions. So it is not surprising that threesomes correspond to Henglein's coercions. Nonetheless, it is a pleasant validation of our design that threesomes turn out to be exactly isomorphic to Henglein's coercions in normal form. Coercion normalization is an iterative process, whereas composition of threesomes is a direct recursive definition. Thus, we believe that the alternative view offered by threesomes may make possible a more efficient implementation than one based directly on the coercion calculus.

The rest of the paper begins with a review of the blame calculus (Section 2). Then, to factor the presentation of the threesome calculus, we present a simplified version that captures the main intuitions and detects cast failures appropriately, but does not track blame (Section 3). We prove that the simplified version is correct and space efficient. Section 4 presents the complete threesome calculus with support for blame tracking and proves that it is correct (equivalent to the blame calculus). Section 5 shows that the threesome calculus is isomorphic to a coercion-based calculus of Siek et al. (2009). Some of the proofs are in-line and the rest are in the Appendix. We explain the relationship between our results and prior work in Section 6.

* A preliminary version of this paper appeared in the informal proceedings of the 2009 Workshop on Script to Program Evolution.

POPL'10, January 17–23, 2010, Madrid, Spain.

2. Twosomes

The definition of the blame calculus (minus subset types) is shown in Figure 1. This blame calculus is the simply-typed lambda calculus extended with the dynamic type, written $*$, and casts of the form $\langle T \stackrel{l}{\Leftarrow} S \rangle s$. We sometimes elide the blame label l on a cast when the label is not relevant. The meta-variable x ranges over variables, k over constants, and B over base types (such as `Int` and `Bool`). We give function application higher precedence than casts. The meta-variables l, m and n range over blame labels. In an implementation, blame labels would not be visible to the programmer, but would be inserted automatically by the parser and would provide access to information such as the location of the cast in the source program.

The dynamic semantics of the cast is straightforward in the case of first-order values such as integers: the run-time check either succeeds and the cast acts like the identity function, or the check fails and the downcast is blamed.

$$\langle \texttt{Int} \stackrel{m}{\Leftarrow} * \rangle \langle * \stackrel{l}{\Leftarrow} \texttt{Int} \rangle 4 \longmapsto \langle \texttt{Int} \stackrel{m}{\Leftarrow} \texttt{Int} \rangle 4 \longmapsto 4$$
$$\langle \texttt{Bool} \stackrel{m}{\Leftarrow} * \rangle \langle * \stackrel{l}{\Leftarrow} \texttt{Int} \rangle 4 \longmapsto \langle \texttt{Bool} \stackrel{m}{\Leftarrow} \texttt{Int} \rangle 4 \longmapsto \mathbf{blame}\ m$$

Higher-Order Casts The semantics of casts is more subtle in the case of higher-order values such as functions. The complication is that one cannot immediately check whether a function respects the target type of the cast. In the example below, when the function `f` of type `Int`$\rightarrow$$*$ is cast to `Int`$\rightarrow$`Int`, there is no way to immediately tell if the function will return an integer every time it is called.

$$\begin{array}{l}\texttt{let f = } (\lambda \texttt{x:Int. if } 0 \leq x \texttt{ then } \langle * \stackrel{m}{\Leftarrow} \texttt{Int} \rangle 2 \\ \qquad\qquad\qquad\qquad \texttt{else } \langle * \stackrel{l}{\Leftarrow} \texttt{Bool} \rangle \texttt{true) in} \\ \texttt{let g = } \langle \texttt{Int} \rightarrow \texttt{Int} \stackrel{n}{\Leftarrow} \texttt{Int} \rightarrow * \rangle \texttt{f in } \ldots\end{array}$$

So long as `g` is only called with non-negative numbers, the behavior of `f` respects the cast. If `g` is ever called with a negative number, the return value of `f` will induce a cast error.

The standard solution defers the check until the function is applied to an argument. When a function passes through a cast, the cast is not reduced but remains as a wrapper around the function. When the cast-wrapped function is applied to a value, then the cast is distributed to the argument and result as specified by the following reduction rule. Note that the direction of the cast on the argument is flipped, as usual. (Also the blame label of the cast on the argument is negated; we discuss this further below.)

$$(\langle S' \rightarrow T' \stackrel{l}{\Leftarrow} S \rightarrow T \rangle v)\ w \longrightarrow \langle T' \stackrel{l}{\Leftarrow} T \rangle\ v(\langle S \stackrel{\overline{l}}{\Leftarrow} S' \rangle w)$$

Blame Tracking Because a higher-order cast is not checked immediately, it might fail at a location far removed from where it was originally applied. Continuing the above example, the cast-wrapped function bound to `g` may be applied at some point much later in the program.

$$\begin{array}{l}\texttt{let g = } \langle \texttt{Int} \rightarrow \texttt{Int} \stackrel{n}{\Leftarrow} \texttt{Int} \rightarrow * \rangle \texttt{f in} \\ \quad \cdots \texttt{ g (-1) } \cdots\end{array}$$

Blame tracking helps to diagnose such failures by mapping cast failures back to their origin in the source program. The above example reduces to $\mathbf{blame}\ n$, indicating that the cast surrounding f caused the failure. Figure 1 shows the reduction rules we use in this paper. The formulation here differs in minor ways from the rules of Wadler and Findler (2009) that will prove technically convenient. The most significant changes are that we allow casts between any two types, even if they are not compatible; and we introduce an empty type $\bot$, such that no value has type $\bot$ and $\bot$ is incompatible with every type.

Blame labels come equipped with a negation operator that is an involution. That is, if m is a blame label then $\overline{m}$ is its negation and $\overline{\overline{m}} = m$. If a program halts with **blame** m, the term contained inside the cast labeled m failed to produce a value of the appropriate type. If a program halts with **blame** $\overline{m}$, the context surrounding the cast labeled m misused the cast value, that is, applied it to a value of an inappropriate type.

The blame calculus checks higher-order casts lazily, reporting blame for an incompatible cast on a function only when the function is applied. For example,

$$\texttt{let g = } \langle \texttt{Int} \rightarrow \texttt{Int} \stackrel{n}{\Leftarrow} \texttt{Bool} \rightarrow \texttt{Bool} \rangle f \texttt{ in } \cdots$$

will fail only if `g` is applied. This is achieved by testing for shallow incompatibility between types. We write $S \# T$ to indicate shallow incompatibility. Distinct base types are shallowly incompatible, a function is shallowly incompatible with any base type, and the type $\bot$ is shallowly incompatible with any type.

In the definition of values, we add a side condition on values of the form $\langle S' \rightarrow T' \stackrel{l}{\Leftarrow} S \rightarrow T \rangle v$. We require that not all S, T, S', and T' be $*$ because otherwise rule (3) would apply.

In this paper we restrict our attention to lazy checking. Siek et al. (2009) discuss other design points, such as eager checking (where any incompatibility is reported as soon as possible). An initial assessment indicates that it would be straightforward to apply the techniques in this paper to the other designs.

Space Efficiency Herman et al. (2007) 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, a wrapper is added each time the function bound to `k` is passed between `even` and `odd`, causing a space leak proportional to `n`.

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

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

```
let rec even(n : Int) : * =
  if (n = 0) then ⟨*⇐Bool⟩true
  else ⟨*⇐Bool⟩odd(n - 1)
and odd(n : Int) : Bool =
  if (n = 0) then false
  else ⟨Bool⇐*⟩even(n - 1)
```

Assuming tail call optimization, cast-free versions of the `even` and `odd` functions require only constant space. However, with the presence of casts, the calls to `even` and `odd` are not tail calls, so 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.

$$\begin{array}{ll} & \texttt{even}(n) \\ \longmapsto & \langle * \Leftarrow \texttt{Bool} \rangle \texttt{odd}(n-1) \\ \longmapsto & \langle * \Leftarrow \texttt{Bool} \rangle \langle \texttt{Bool} \Leftarrow * \rangle \texttt{even}(n-2) \\ \longmapsto & \langle * \Leftarrow \texttt{Bool} \rangle \langle \texttt{Bool} \Leftarrow * \rangle \langle * \Leftarrow \texttt{Bool} \rangle \texttt{odd}(n-3) \\ \longmapsto & \cdots \end{array}$$

Herman et al. (2007) recover space efficiency in a cast calculus without blame by 1) using the coercion calculus (Henglein 1994) to compactly represent sequences of casts, and 2) normalizing sequences of coercions that appear in tail-position before making function calls. Siek et al. (2009) extend this work by integrating blame tracking into the coercion calculus, thereby achieving a

Syntax

types	R, S, T, U	$::=$	$B \mid S \to T \mid * \mid \bot$
terms	s, t	$::=$	$k \mid x \mid \lambda x{:}S.t \mid s\ t \mid$ $\langle T \stackrel{l}{\Leftarrow} S \rangle s$
ground types	G, H	$::=$	$B \mid * \to *$
values	v, w	$::=$	$k \mid \lambda x{:}S.t \mid \langle * \stackrel{l}{\Leftarrow} G \rangle v \mid$ $\langle S' \to T' \stackrel{l}{\Leftarrow} S \to T \rangle v$
contractums	r	$::=$	$t \mid \mathbf{blame}\ l$
eval. contexts	E	$::=$	$\Box \mid E\ t \mid v\ E \mid \langle T \stackrel{l}{\Leftarrow} S \rangle E$

Additional Typing Rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash s : S}{\Gamma \vdash \langle T \stackrel{l}{\Leftarrow} S \rangle s : T}$$

Shallow Incompatibility $\boxed{S \# T}$

$$\frac{B \neq B'}{B \# B'} \qquad B \# T \to T' \qquad T \to T' \# B \qquad \bot \# T \qquad T \# \bot$$

Reductions $\boxed{s \longrightarrow r}$

$$(\lambda x : S.\ t)\ v \longrightarrow [x := v]t \tag{1}$$

$$k\ k' \longrightarrow \delta(k, k') \tag{2}$$

$$\langle G \stackrel{l}{\Leftarrow} G \rangle v \longrightarrow v \tag{3}$$

$$\langle * \stackrel{l}{\Leftarrow} * \rangle v \longrightarrow v \tag{4}$$

$$\langle H \stackrel{m}{\Leftarrow} * \rangle \langle * \stackrel{l}{\Leftarrow} G \rangle v \longrightarrow \langle H \stackrel{m}{\Leftarrow} G \rangle v \tag{5}$$

$$\langle * \stackrel{l}{\Leftarrow} S \to T \rangle v \longrightarrow \langle * \stackrel{l}{\Leftarrow} * \to * \rangle \langle * \to * \stackrel{l}{\Leftarrow} S \to T \rangle v \quad \text{if } S \to T \neq * \to * \tag{6}$$

$$\langle S \to T \stackrel{l}{\Leftarrow} * \rangle v \longrightarrow \langle S \to T \stackrel{l}{\Leftarrow} * \to * \rangle \langle * \to * \stackrel{l}{\Leftarrow} * \rangle v \quad \text{if } S \to T \neq * \to * \tag{7}$$

$$(\langle S' \to T' \stackrel{l}{\Leftarrow} S \to T \rangle v)\ w \longrightarrow \langle T' \stackrel{l}{\Leftarrow} T \rangle\ v (\langle S \stackrel{\bar{l}}{\Leftarrow} S' \rangle w) \quad \text{if not } S \to T = S' \to T' = * \to * \tag{8}$$

$$\langle T \stackrel{l}{\Leftarrow} S \rangle v \longrightarrow \mathbf{blame}\ l \qquad \text{if } S \# T \tag{9}$$

Single-step evaluation $\boxed{s \longmapsto r}$

$$\frac{t \longrightarrow t'}{E[t] \longmapsto E[t']} \qquad \frac{t \longrightarrow \mathbf{blame}\ l}{E[t] \longmapsto \mathbf{blame}\ l}$$

Figure 1. Twosomes

space-efficient implementation of the blame calculus. In this paper we explore an alternative based on threesomes.

3. Threesomes Without Blame

The goal of the threesome calculus is to achieve space efficiency while maintaining the high-level nature of the blame calculus, that is, expressing casts with types. The key to space efficiency is to compress sequences of casts while maintaining the same behavior. There are two aspects to the behavior of a cast: 1) detecting cast failures and 2) allocating blame to the appropriate cast in the source program. In this section we describe a simplified version of the threesome calculus that focuses on detecting cast failures. In relating the simplified threesome calculus to the blame calculus, we ignore all blame labels in the blame calculus. In Section 4 we present the complete threesome calculus with blame tracking.

When discussing a sequence of casts, we abbreviate

$$\langle T_n \Leftarrow T_{n-1} \rangle \cdots \langle T_3 \Leftarrow T_2 \rangle \langle T_2 \Leftarrow T_1 \rangle$$

as follows to avoid repeating types.

$$\langle T_n \Leftarrow T_{n-1} \cdots T_3 \Leftarrow T_2 \Leftarrow T_1 \rangle$$

Towards understanding how to compress casts, consider the following sequence:

$$\langle * \to * \ \Leftarrow\ \texttt{Bool} \to * \ \Leftarrow\ * \to \texttt{Int} \ \Leftarrow\ * \to * \rangle$$

This sequence, when applied to a function, forces the function to accept an argument of type `Bool` and to return a result of type `Int`; any other argument or result induces an error. Thus, the above sequence of casts is equivalent to just two casts where the intermediate type is $\texttt{Bool} \to \texttt{Int}$.

$$\langle * \to * \ \Leftarrow\ \texttt{Bool} \to \texttt{Int} \ \Leftarrow\ * \to * \rangle$$

The type $\texttt{Bool} \to \texttt{Int}$ represents a stronger cast, one that is more likely to fail, than either of the two intermediate types in the original sequence. This notion of stronger is captured by the *naive subtyping* relation, written $<:_n$, of Wadler and Findler (2007) (defined in Figure 2); similar relations include the $\leqslant$ ordering on coercions of Henglein (1994) and the $\lessdot$ ordering on retracts of Scott (1976). Our idea is to compress a sequence of cast into just two casts where the intermediate type is the *greatest lower bound* of the types in the original sequence.

Naive subtyping is covariant in the domain of function types instead of contravariant like ordinary subtyping. It is interesting to compare taking the greatest lower bound with respect to naive subtyping versus ordinary subtyping. In the above example, with naive, covariant subtyping we have

$$\texttt{Bool} \to \texttt{Int} <:_n * \to \texttt{Int} <:_n * \to *$$
$$\texttt{Bool} \to \texttt{Int} <:_n \texttt{Bool} \to * <:_n * \to *$$

and so the greatest lower bound of the original casts with regard to $<:_n$ is $\texttt{Bool} \to \texttt{Int}$, which has the desired effect. In contrast, with ordinary, contravariant subtyping (written $<:$) we have

$$* \to \texttt{Int} <: * \to * <: \texttt{Bool} \to *$$

So the greatest lower bound of the original casts with regard to $<:$ is $* \to \texttt{Int}$. Using this as the intermediate type gives a different collapsed cast

$$\langle * \to * \ \Leftarrow\ * \to \texttt{Int} \ \Leftarrow\ * \to * \rangle$$

But this cast is too lenient; it correctly requires the result to be `Int` but does not require the argument to be `Bool`.

We write $S \,\&\, T$ for the greatest lower bound of S and T with respect to $<:_n$; we also sometimes call this the *meet* of S and T. In general, an arbitrary sequence of casts can collapse to a contextually equivalent pair of casts where the intermediate type is the meet of all the types in the sequence, so

$$\langle T_n \Leftarrow T_{n-1} \Leftarrow \cdots \Leftarrow T_2 \Leftarrow T_1 \rangle t$$

is equivalent to

$$\langle T_n \Leftarrow T_n \,\&\, T_{n-1} \,\&\, \cdots \,\&\, T_2 \,\&\, T_1 \Leftarrow T_1 \rangle t$$

We introduce a special notation for casts of this kind. For any three types R, S, T with $R <:_n S$ and $R <:_n T$, and any term t of type S, we write

$$\langle T \stackrel{R}{\Longleftarrow} S \rangle t$$

and call it a *threesome*. It is equivalent to the pair of casts

$$\langle T \Leftarrow R \rangle \langle R \Leftarrow S \rangle t$$

The syntax of the threesome calculus without blame is given in Figure 2. Analogously to the side condition on values for twosomes,

we require that in a value of the form $\langle T \to T' \stackrel{R \to R'}{\Longleftarrow} S \to S' \rangle u$, not all of T, T', R, R', S and S' can be $*$.

Greatest lower bounds can be computed by the algorithm for $S \,\&\, T$ defined in Figure 2. The algorithm is straightforward: the meet of identical base types is that base type, of two function types is a function from the meet of the domains to the meet of the ranges, of any type with $*$ is the other type, and of any shallowly incompatible types is $\bot$. Note that the greatest lower bound always exists thanks to the inclusion of the empty type $\bot$. Since $S \,\&\, T$ is symmetric, the reader may wonder why we write $T \& S$ in the clause for function types; this is foreshadowing modifications required in the next section for tracking blame.

Our first result is to confirm that the algorithm in Figure 2 computes greatest lower bounds.

Proposition 1 (Meet algorithm returns the greatest lower bound)**.**

1. *$S \,\&\, T <:_n S$ and $S \,\&\, T <:_n T$*
2. *If $R <:_n S$ and $R <:_n T$, then $R <:_n S \,\&\, T$.*

The operational semantics of the threesome calculus is given in Figure 2. Most of the reduction rules are a straightforward adaptation from twosomes to threesomes. For example, rule (13) is equivalent to two applications of rule (3) from the blame calculus. The main difference is the addition of rule (16), which is responsible for compressing a pair of threesomes into a single threesome by taking the meet of the two intermediate types. The source and target types of a threesome do not play a role in the reduction rules and could be erased but their presence streamlines the meta-theory by giving us unicity of types.

To make sure that sequences of casts do not accumulate in tail position, we change evaluation contexts to recognize sequences of casts that need to be compressed. Our evaluation strategy reduces sequences outside-in, repeatedly compressing the two outermost casts in a sequence. An F context represents a location that is not immediately inside a cast (F is for "cast free"), which indicates an appropriate location to apply a cast-reducing rule. The context E includes both the F contexts and contexts containing sequences of casts with length at most one.

This formulation of evaluation contexts differs from the contexts used in Herman et al. (2007). Unique decomposition does not hold for their contexts, for example, in a sequence of three casts, either the outer two or the inner two can be merged. However, because the coercion calculus is confluent, the lack of unique decomposition does not pose a serious problem. Nevertheless, we prefer to use evaluation contexts that ensure a unique decomposition.

Proposition 2 (Unique Decomposition)**.** *For a well-typed closed t, either t is a value or there is a unique decomposition into a redex t' and an evaluation context E such that $t = E[t']$.*

The values of the threesome calculus differ in an important way from the blame calculus, which reflects the improved space efficiency. In the blame calculus, a value of type $T \to T'$ may contain an arbitrary number of outermost casts. In the threesome calculus, a value of type $T \to T'$ may contain at most one outermost cast.

Lemma 1 (Canonical Forms)**.**

1. *If $\emptyset \vdash v : B$, then v is a constant.*
2. *If $\emptyset \vdash v : T \to T'$, then v has one of the following forms: k, $\lambda x{:}S.t$, or $\langle T \to T' \stackrel{R \to R'}{\Longleftarrow} S \to S' \rangle u$.*
3. *If $\emptyset \vdash v : *$, then v has the form $\langle * \stackrel{R}{\Longleftarrow} S \rangle u$.*
4. *No value has type $\bot$.*

Proof. The proofs are by case analysis on v and inversion on the typing rules. □

Syntax

$$
\begin{array}{lllll}
\text{terms} & s, t & ::= & k \mid x \mid \lambda x{:}S.t \mid s\,t \mid \langle T \stackrel{R}{\Longleftarrow} S \rangle s \\
\text{uncoerced val.} & u & ::= & k \mid \lambda x{:}S.\ t \\
\text{values} & v, w & ::= & u \mid \langle * \stackrel{R}{\Longleftarrow} S \rangle u \mid \\
 & & & \langle T \to T' \stackrel{R \to R'}{\Longleftarrow} S \to S' \rangle u \\
\text{contractums} & r & ::= & t \mid \mathbf{blame} \\
\text{cast-free contexts} & F & ::= & \Box \mid E[\Box\ t] \mid E[v\ \Box] \\
\text{eval. contexts} & E & ::= & F \mid F[\langle T \stackrel{R}{\Longleftarrow} S \rangle \Box]
\end{array}
$$

Naive subtyping $\boxed{S <:_n T}$

$$B <:_n B \quad S <:_n * \quad \bot <:_n T \quad \frac{S <:_n T \quad S' <:_n T'}{S \to S' <:_n T \to T'}$$

Additional typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash s : S \quad R <:_n S \quad R <:_n T}{\Gamma \vdash \langle T \stackrel{R}{\Longleftarrow} S \rangle s : T}$$

Meet algorithm (computes the greatest lower bound) $\boxed{S \,\&\, T}$

$$
\begin{aligned}
* \,\&\, T &= T \\
S \,\&\, * &= S \\
B \,\&\, B &= B \\
(S \to S') \,\&\, (T \to T') &= (T \,\&\, S) \to (S' \,\&\, T') \\
S \,\&\, T &= \bot \quad \text{if } S \# T
\end{aligned}
$$

Reductions $\boxed{t \longrightarrow t}$

$$(\lambda x : S.t)\ v \longrightarrow [x := v]t \tag{10}$$

$$k\ k' \longrightarrow \delta(k, k') \tag{11}$$

$$(\langle T \to T' \stackrel{R \to R'}{\Longleftarrow} S \to S' \rangle u)\ w \longrightarrow \langle T' \stackrel{R'}{\Longleftarrow} S' \rangle u\ (\langle S \stackrel{R}{\Longleftarrow} T \rangle w)$$
$$\text{if not } T \to T' = R \to R' = S \to S' = * \to * \tag{12}$$

Cast reductions $\boxed{t \longrightarrow_c r}$

$$\langle G \stackrel{G}{\Longleftarrow} G \rangle u \longrightarrow_c u \tag{13}$$

$$\langle * \stackrel{*}{\Longleftarrow} * \rangle u \longrightarrow_c u \tag{14}$$

$$\langle T \stackrel{\bot}{\Longleftarrow} S \rangle u \longrightarrow_c \mathbf{blame} \tag{15}$$

$$\langle T \stackrel{R'}{\Longleftarrow} S' \rangle \langle S' \stackrel{R}{\Longleftarrow} S \rangle s \longrightarrow_c \langle T \stackrel{R' \& R}{\Longleftarrow} S \rangle s \tag{16}$$

Single-step evaluation $\boxed{t \longmapsto r}$

$$\frac{t \longrightarrow t'}{E[t] \longmapsto E[t']} \qquad \frac{t \longrightarrow_c t'}{F[e] \longmapsto F[t']} \qquad \frac{t \longrightarrow_c \mathbf{blame}}{F[t] \longmapsto \mathbf{blame}}$$

Figure 2. Threesomes without blame

3.1 Correspondence Between Twosomes and Threesomes

We translate from the blame calculus to the threesome calculus by mapping casts of the form $\langle T \Leftarrow S \rangle s$ to $\langle T \stackrel{T \& S}{\Longleftarrow} S \rangle s$. The definitions for compiling twosomes to threesomes, and decompiling threesomes to twosomes, written $\langle\!\langle t \rangle\!\rangle$ and $\langle\!\langle t \rangle\!\rangle^{-1}$ respectively, are given in Figure 3.

The proof that the threesomes calculus without blame is equivalent to the blame calculus (ignoring blame labels) hinges on two facts. First, the cast $\langle T \Leftarrow S \rangle$ is equivalent to $\langle T \Leftarrow T \,\&\, S \Leftarrow S \rangle$. This is necessary to justify the compilation from twosomes to threesomes. Second, $\langle T \Leftarrow R' \Leftarrow S' \Leftarrow R \Leftarrow S \rangle$ is equivalent to

Compilation from twosomes to threesomes $\boxed{\langle\!\langle t \rangle\!\rangle = t}$

$$\langle\!\langle x \rangle\!\rangle = x \qquad \langle\!\langle k \rangle\!\rangle = k$$
$$\langle\!\langle \lambda x : S.\ t \rangle\!\rangle = \lambda x : S.\ \langle\!\langle t \rangle\!\rangle \qquad \langle\!\langle t\ s \rangle\!\rangle = \langle\!\langle t \rangle\!\rangle\ \langle\!\langle s \rangle\!\rangle$$
$$\langle\!\langle \langle T \Leftarrow S \rangle s \rangle\!\rangle = \langle T \overset{T \& S}{\Longleftarrow} S \rangle \langle\!\langle s \rangle\!\rangle$$

Decompilation from threesomes to twosomes $\boxed{\langle\!\langle t \rangle\!\rangle^{-1} = t}$

$$\langle\!\langle x \rangle\!\rangle^{-1} = x \qquad \langle\!\langle k \rangle\!\rangle^{-1} = k$$
$$\langle\!\langle \lambda x : S.\ t \rangle\!\rangle^{-1} = \lambda x : S.\ \langle\!\langle t \rangle\!\rangle^{-1} \qquad \langle\!\langle t\ s \rangle\!\rangle^{-1} = \langle\!\langle t \rangle\!\rangle^{-1}\ \langle\!\langle s \rangle\!\rangle^{-1}$$
$$\langle\!\langle \langle T \overset{R}{\Longleftarrow} S \rangle s \rangle\!\rangle^{-1} = \langle T \Leftarrow R \rangle \langle R \Leftarrow S \rangle \langle\!\langle s \rangle\!\rangle^{-1}$$

Figure 3. From twosomes to threesomes and back again.

$\langle T \Leftarrow R' \,\&\, R \Leftarrow S \rangle$ provided $R' <:_n T$, $R' <:_n S'$, $R <:_n S'$, and $R <:_n S$. This fact is necessary to justify the threesome reduction rule (16), which compresses two threesomes. Both of these facts are corollaries of the following fundamental lemma. In a sequence $\langle T \Leftarrow R \Leftarrow S \rangle$, it is safe to bypass R if going from $S \,\&\, T$ to R is an upcast.

Towards formalizing this in terms of contextual equivalence, we define *contexts* and *results* as follows.

$$\begin{array}{llll} \text{contexts} & C & ::= & \Box \mid C\ t \mid t\ C \mid \lambda x{:}S.\ C \mid \langle T \Leftarrow S \rangle C \\ \text{results} & f & ::= & v \mid \mathbf{blame} \end{array}$$

A program t is said to *converge*, written $t \downarrow$, if $\exists f.\ t \longmapsto^* f$, where $\longmapsto^*$ is the reflexive, transitive closure of $\longmapsto$. These definitions for twosomes carry over easily to the threesome calculus.

Definition 1. *A term t is* contextually equivalent *to another term t', written $t =_{\mathrm{ctx}} t'$, if for any context C, $C[t] \downarrow$ iff $C[t'] \downarrow$.*

Lemma 2 (Fundamental Property of Casts)**.**
If $T \,\&\, S <:_n R$, then $\langle T \Leftarrow R \Leftarrow S \rangle t =_{\mathrm{ctx}} \langle T \Leftarrow S \rangle t$.

The proof of the fundamental property of casts is in the Appendix. With this property in hand we can prove the two key facts.

Corollary 1.

1. *$\langle T \Leftarrow S \rangle s =_{\mathrm{ctx}} \langle T \Leftarrow T \,\&\, S \Leftarrow S \rangle s$, and*
2. *$\langle T \Leftarrow R' \Leftarrow S' \Leftarrow R \Leftarrow S \rangle s =_{\mathrm{ctx}} \langle T \Leftarrow R' \,\&\, R \Leftarrow S \rangle$ provided $R' <:_n T$, $R' <:_n S'$, $R <:_n S'$, and $R <:_n S$.*

Proof.

1. Note that subtyping is reflexive and then apply the fundamental property of casts.
2. To prove the second part, we apply the fundamental property of casts several times as follows.

$$\begin{aligned} & \langle T \Leftarrow R' \Leftarrow S' \Leftarrow R \Leftarrow S \rangle s \\ =_{\mathrm{ctx}}\ & \langle T \Leftarrow R' \Leftarrow R \Leftarrow S \rangle s \\ =_{\mathrm{ctx}}\ & \langle T \Leftarrow R' \Leftarrow R' \,\&\, R \Leftarrow R \Leftarrow S \rangle s \\ =_{\mathrm{ctx}}\ & \langle T \Leftarrow R' \,\&\, R \Leftarrow R \Leftarrow S \rangle s \\ =_{\mathrm{ctx}}\ & \langle T \Leftarrow R' \,\&\, R \Leftarrow S \rangle s \end{aligned}$$

□

We need a few more facts before proving the correctness theorem.

Lemma 3. $t =_{\mathrm{ctx}} \langle\!\langle \langle\!\langle t \rangle\!\rangle \rangle\!\rangle^{-1}$.

Proof. The proof is by induction on t. The case for casts relies on Corollary 1 (part 1). □

Lemma 4.

1. *If $t =_{\mathrm{ctx}} \langle\!\langle t_3 \rangle\!\rangle^{-1}$ and $t_3 \longmapsto^* t_3'$, then $t =_{\mathrm{ctx}} \langle\!\langle t_3' \rangle\!\rangle^{-1}$.*
2. *If $t_3 =_{\mathrm{ctx}} \langle\!\langle t \rangle\!\rangle$ and $t \longmapsto^* t'$, then $t_3 =_{\mathrm{ctx}} \langle\!\langle t' \rangle\!\rangle$.*

Proof. 1. The proof is by induction on $t_3 \longmapsto^* t_3'$ and by cases on $\longrightarrow$. The case for rule (16) relies on Corollary 1 (part 2). The other cases rely on the fact $t \longmapsto t'$ implies $t =_{\mathrm{ctx}} t'$.

2. The proof is by induction on $t \longmapsto^* t'$ and by cases on $\longrightarrow$. The cases rely on Lemma 2 and the fact $t_3 \longmapsto t_3'$ implies $t_3 =_{\mathrm{ctx}} t_3'$.

□

Applying compilation or decompilation to results produces terms that converge.

Lemma 5. *$\langle\!\langle f \rangle\!\rangle \downarrow$ and $\langle\!\langle f_3 \rangle\!\rangle^{-1} \downarrow$.*

Proof. The proofs are by induction on f and f_3. □

We can now state and prove the correctness theorem.

Theorem 1 (Correctness of the threesome calculus without blame)**.**
$\langle\!\langle t \rangle\!\rangle \downarrow$ if and only if $t \downarrow$.

Proof. By Lemma 3 we have $t =_{\mathrm{ctx}} \langle\!\langle \langle\!\langle t \rangle\!\rangle \rangle\!\rangle^{-1}$.

1. ($\Longrightarrow$) We have an f_3 such that $\langle\!\langle t \rangle\!\rangle \longmapsto^* f_3$. Then by Lemma 4 (part 1) we have $t =_{\mathrm{ctx}} \langle\!\langle f_3 \rangle\!\rangle^{-1}$. Then by Lemma 5, $t \downarrow$.
2. ($\Longleftarrow$) We have an f such that $t \longmapsto^* f$. Then by Lemma 4 (part 2) we have $\langle\!\langle t \rangle\!\rangle =_{\mathrm{ctx}} \langle\!\langle f \rangle\!\rangle$. Then by Lemma 5, $\langle\!\langle t \rangle\!\rangle \downarrow$.

□

3.2 Space Efficiency

The main task in bounding the size of casts during execution is to put a bound on the result of merging two casts. The approach taken by Herman et al. (2007) for the coercion calculus is to show that the height of a composed coercion is no greater than the height of the two coercions. Then, because normalized coercions are trees with limited branching, it follows that the size of the composed coercion is bounded by roughly 2^h where h is the height.

We obtain a tighter bound for the threesome calculus that takes into account that when two casts are composed, there is often considerable overlap between the two intermediate types, and therefore the resulting size of the new intermediate type is not much larger. (We take the size of a type to be the number of nodes in the type when considered as an abstract syntax tree.) A strawman for the bound is the size of the greatest lower bound of all the types that occur in the program. The problem with this strawman is that the greatest lower bound of two types can sometimes be smaller, thereby not providing an upper bound on size. For example, $\mathtt{Int} \,\&\, (* \to *) = \bot$. Instead we need to take the maximum of the structure of the two types. To accomplish this we map types to their *shadow*, written $\lceil T \rceil$, and then compute the greatest lower bound. We define the shadow of a type as follows.

$$\begin{aligned} \lceil B \rceil &= * \\ \lceil S \to T \rceil &= \lceil S \rceil \to \lceil T \rceil \\ \lceil * \rceil &= * \\ \lceil \bot \rceil &= * \end{aligned}$$

With the previous example, we have $\lceil \mathtt{Int} \rceil \,\&\, \lceil * \to * \rceil = * \to *$.

Proposition 3 (Properties of shadows)**.**

1. *$size(T) = size(\lceil T \rceil)$.*
2. *If $R <:_n \lceil S \rceil$ and $R <:_n \lceil T \rceil$ then $R <:_n \lceil S \,\&\, T \rceil$.*
3. *If $\lceil S \rceil <:_n \lceil T \rceil$, then $size(\lceil T \rceil) \leq size(\lceil S \rceil)$.*

We prove that the sizes of types in the program during reduction are bounded above by the greatest lower bound of the shadows of the types in the original program. The main lemma below shows that the greatest lower bound remains a lower bound during reduction. We can then apply the above property 3 to show that the size of the greatest lower bound is an upper bound on the size of any type in the program. Towards formally stating these properties, we give the following definitions.

$$T \in t \equiv \text{type } T \text{ syntactically occurs in term } t$$
$$\lceil t \rceil \equiv \{\lceil T \rceil \mid T \in t\}$$
$$S <:_n \mathcal{T} \equiv \forall T \in \mathcal{T}.\ S <:_n T$$
$$\&\{T_1, \ldots, T_n\} \equiv T_1 \& \cdots \& T_n$$

Lemma 6 (Preservation of lower bounds)**.**
If $T <:_n \lceil t \rceil$, and $t \longrightarrow t'$, then $T <:_n \lceil t' \rceil$.

Lemma 7 (Preservation of compilation)**.**
If $T <:_n \lceil t \rceil$, then $T <:_n \lceil \langle\!\langle t \rangle\!\rangle \rceil$.

Theorem 2 (Bound on size of types and therefore casts)**.**
If $\langle\!\langle t \rangle\!\rangle \longrightarrow^* t'$, then for any $T \in t'$, $size(T) \leq size(\&\lceil t \rceil)$.

Let $|t|$ be the term t with all the casts erased. The size of a threesome program during execution does not differ by more than a constant factor compared to itself with all the casts erased.

Theorem 3 (Space efficiency)**.**
For any program t, if $t \longrightarrow^* t'$, then

$$size(t') \leq size(\&\lceil t \rceil) \cdot size(|t'|).$$

Proof. The proof is essentially the same as in Herman et al. (2007), except we use Theorem 2 for the bound on the size of types. □

4. Threesomes with Blame

In the previous section we showed that a simplified threesome calculus implements the blame calculus ignoring blame labels. We now turn our attention to adding correct blame allocation to the threesome calculus. Consider what would happen if we were to naively add blame labels to threesome casts, say, writing the blame label next to the intermediate type. When merging two casts, we would need to choose between label m and l.

$$\langle T \overset{U,m}{\Longleftarrow} S' \rangle \langle S' \overset{R,l}{\Longleftarrow} S \rangle s \longrightarrow \langle T \overset{U \& R,n}{\Longleftarrow} S \rangle s$$

Should n be m or l? Unfortunately, either choice is wrong. Consider the following example in which different casts within the same sequence are allocated blame.

$$g \equiv \lambda f : * \to * .\ \langle * \to * \overset{l}{\Leftarrow} \texttt{Bool} \to * \overset{n}{\Leftarrow} * \to \texttt{Bool} \overset{m}{\Leftarrow} * \to * \rangle f$$
$$g\ (\lambda x : * .\ \langle * \overset{o}{\Leftarrow} \texttt{Int} \rangle 1)\ \langle * \overset{p}{\Leftarrow} \texttt{Bool} \rangle \mathit{true} \longrightarrow^* \mathbf{blame}\ m$$
$$g\ (\lambda x : * .\ x)\ \langle * \overset{o}{\Leftarrow} \texttt{Int} \rangle 1 \longrightarrow^* \mathbf{blame}\ l$$

Having just one blame label per cast is not enough.

The solution we propose is to incorporate blame labels into the intermediate type of a threesome. The above function g is then represented in the threesome calculus as follows.

$$g = \lambda f : * \to * .\ \langle * \to * \overset{\texttt{Bool}^l \to^\epsilon \texttt{Bool}^m}{\Longleftarrow} * \to * \rangle f$$

The label on the codomain of the intermediate type is m because cast m is the first (from right to left) to project from $*$ to `Bool` in the codomain. The label on the domain is l because cast l is the first (from left to right) to project from $*$ to `Bool`. The function type itself is given the empty label ϵ because in this sequence of casts, the function type cannot induce blame.

The definition of the threesome calculus with blame is given in Figure 4. The erasure of a labeled type P to a type is written $|P|$.

Syntax

optional labels	p, q	::=	$l \mid \epsilon$
labeled types	P, Q	::=	$B^p \mid P \to^p Q \mid * \mid \bot^{lGp}$
terms	s, t	::=	$k \mid x \mid \lambda x\!:\!S.t \mid s\ t \mid \langle T \overset{P}{\Longleftarrow} S \rangle s$
values	v, w	::=	$u \mid \langle * \overset{P}{\Longleftarrow} S \rangle u \mid \langle S' \to T' \overset{P \to^\epsilon Q}{\Longleftarrow} S \to T \rangle u$
contractums	r	::=	$t \mid \mathbf{blame}\ l$
cast-free contexts	F	::=	$\Box \mid E[\Box\ t] \mid E[v\ \Box]$
evaluation contexts	E	::=	$F \mid F[\langle T \overset{P}{\Longleftarrow} S \rangle \Box]$

Additional typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash s : S \qquad |P| <:_n S \qquad |P| <:_n T}{\Gamma \vdash \langle T \overset{P}{\Longleftarrow} S \rangle s : T}$$

Composition $\boxed{Q \circ P}$

$$B^q \circ B^p = B^p$$
$$P \circ * = P$$
$$* \circ P = P$$
$$Q^{Hm} \circ P^{Gp} = \bot^{mGp} \qquad \text{if } G \neq H$$
$$Q \circ \bot^{mGp} = \bot^{mGp}$$
$$\bot^{mGq} \circ P^{Gp} = \bot^{mGp}$$
$$\bot^{mHl} \circ P^{Gp} = \bot^{lGp} \qquad \text{if } G \neq H$$
$$(P' \to^q Q') \circ (P \to^p Q) = (P \circ P') \to^p (Q' \circ Q)$$

Reductions $\boxed{s \longrightarrow r}$

$$(\lambda x\!:\!S.t)\ v \longrightarrow [x := v]t \tag{17}$$
$$k\ k' \longrightarrow \delta(k, k') \tag{18}$$
$$(\langle T \to T' \overset{P \to^\epsilon P'}{\Longleftarrow} S \to S' \rangle u)\ w \longrightarrow \langle T' \overset{P'}{\Longleftarrow} S' \rangle u\ (\langle S \overset{P}{\Longleftarrow} T \rangle w)$$
$$\text{if not } T \to T' = |P \to^\epsilon P'| = S \to S' = * \to * \tag{19}$$

Cast reductions $\boxed{t \longrightarrow_c r}$

$$\langle G \overset{G^\epsilon}{\Longleftarrow} G \rangle u \longrightarrow_c u \tag{20}$$
$$\langle * \overset{*}{\Longleftarrow} * \rangle u \longrightarrow_c u \tag{21}$$
$$\langle T \overset{\bot^{lG\epsilon}}{\Longleftarrow} S \rangle u \longrightarrow_c \mathbf{blame}\ l \tag{22}$$
$$\langle T \overset{Q}{\Longleftarrow} S' \rangle \langle S' \overset{P}{\Longleftarrow} S \rangle s \longrightarrow_c \langle T \overset{Q \circ P}{\Longleftarrow} S \rangle s \tag{23}$$

Single-step evaluation $\boxed{t \longmapsto r}$

$$\frac{t \longrightarrow t'}{E[t] \longmapsto E[t']} \qquad \frac{t \longrightarrow_c t'}{F[e] \longmapsto F[t']} \qquad \frac{t \longrightarrow_c \mathbf{blame}\ l}{F[t] \longmapsto \mathbf{blame}\ l}$$

Figure 4. Threesomes with blame

We require that in a value of the form $\langle S' \to T' \overset{P \to^\epsilon Q}{\Longleftarrow} S \to T \rangle u$, not all of S', T', P, Q, S, and T can be $*$.

Next we discuss the semantics of the threesome calculus. With the addition of blame labels, we must replace the meet operator with something that takes blame into account. The order of failure becomes observable so the operator is no longer symmetric. Thus, we call the new operator "composition" and use the symbol $\circ$.

Before giving the definition of the composition operator, we establish some auxiliary notation. We write $gnd(P) = G^p$, where

G is the ground type that is shallowly compatible with $|P|$, and p is the topmost blame label in P.

$$gnd(B^p) = B^p \qquad gnd(P \to^p Q) = (* \to *)^p$$

In patterns, we write P^{Gp} to indicate that P is a labeled type with $gnd(P) = G^p$.

The composition of labeled types is defined in Figure 4. In general, the label on the right-hand type takes precedence over the left-hand type. This is because the right-hand type induces a cast error before the left-hand side. The composition of two function types is a function type whose blame label is taken from the right-hand function type, whose range is the composition of the ranges of the two function types and whose domain is the *reverse* composition of the domains. This reversal mimics the contravariant behavior of function casts in the blame calculus; see rule (8).

Our treatment of the labeled bottom type $\bot^{lGp}$ deserves some explanation. We initially tried to label bottom types with a single label, as in $\bot^l$. However, that approach fails to capture the correct blame tracking behavior. Consider the following examples.

$$\langle \texttt{Int} \overset{l}{\Leftarrow} * \overset{m}{\Leftarrow} \texttt{Bool} \overset{n}{\Leftarrow} * \overset{o}{\Leftarrow} \texttt{Int} \rangle 1 \longrightarrow \textbf{blame}\ n$$

$$\langle \texttt{Int} \overset{l}{\Leftarrow} * \overset{m}{\Leftarrow} \texttt{Bool} \overset{n}{\Leftarrow} * \overset{o}{\Leftarrow} \texttt{Bool} \rangle \mathit{true} \longrightarrow \textbf{blame}\ l$$

Recall that in the threesome calculus, casts are merged outside-in. The two outermost casts would be merged into cast with middle type $\bot^l$. For the next merge, we could choose to produce either $\bot^l$ or $\bot^n$. However, either choice would be wrong for one of the above examples. Our solution is to label bottom types with not only a label, but also with a labeled ground type. So in this case, the second merge results in $\bot^{l\texttt{Bool}n}$.

One might wonder why the bottom type only needs to remember two labels and not more. A close inspection of the composition rules dealing with bottom reveals why this is the case. When composing $\bot^{mHq}$ with a labeled type P^{Gp} on the right, either H is equal G, so p overshadows q and the result is $\bot^{mGp}$, or H is not equal to G, and we have a new cast failure that needs to blame q. So we can forget the label m and the result is $\bot^{qGp}$

The following proposition expresses the relationship between composition and meet.

Proposition 4. $|Q \circ P| = |Q| \,\&\, |P|$

The main difference between the reduction rules for the threesome calculus, shown in Figure 4, and the rules for the simplified calculus is the use of the composition operator $\circ$ in place of the meet operator $\&$ in rule (23).

4.1 Correspondence Between Twosomes and Threesomes

The compilation of the blame calculus to the threesome calculus, written $\langle\!\langle t \rangle\!\rangle$, is given in Figure 5. The case for casts is a bit more complicated than for the simplified calculus in that we must also define how to propagate labels into the intermediate type; function $\langle\!\langle T \overset{l}{\Leftarrow} S \rangle\!\rangle$ performs this duty and is defined in Figure 5. Its definition is straightforward, though two things are worth pointing out. The label is negated when going under the left-hand side of a function type and we put the empty label ϵ on types in situations where the cast could not produce blame, such as identity casts and injections. The following is the relationship between $\langle\!\langle T \overset{l}{\Leftarrow} S \rangle\!\rangle$ and meet.

Proposition 5. $|\langle\!\langle T \overset{l}{\Leftarrow} S \rangle\!\rangle| = T \,\&\, S.$

With the addition of blame labels, it is not as straightforward to establish the equivalence between the threesome calculus and the blame calculus because we can no longer merge casts in the blame calculus itself. However, we are able to construct a bisimulation

Compile casts to labeled types $\boxed{\langle\!\langle T \overset{l}{\Leftarrow} S \rangle\!\rangle = P}$

$$\langle\!\langle B \overset{l}{\Leftarrow} B \rangle\!\rangle = B^\epsilon \quad \langle\!\langle * \overset{l}{\Leftarrow} * \rangle\!\rangle = * \quad \langle\!\langle B \overset{l}{\Leftarrow} * \rangle\!\rangle = B^l \quad \langle\!\langle * \overset{l}{\Leftarrow} B \rangle\!\rangle = B^\epsilon$$
$$\langle\!\langle T \overset{l}{\Leftarrow} S \rangle\!\rangle = \bot^{lG\epsilon} \quad \text{if } S \# T, \text{ where } G = gnd(S)$$
$$\langle\!\langle S' \to T' \overset{l}{\Leftarrow} S \to T \rangle\!\rangle = \langle\!\langle S \overset{\bar{l}}{\Leftarrow} S' \rangle\!\rangle \to^\epsilon \langle\!\langle T' \overset{l}{\Leftarrow} T \rangle\!\rangle$$
$$\langle\!\langle S \to T \overset{l}{\Leftarrow} * \rangle\!\rangle = \langle\!\langle * \overset{\bar{l}}{\Leftarrow} S \rangle\!\rangle \to^l \langle\!\langle T \overset{l}{\Leftarrow} * \rangle\!\rangle$$
$$\langle\!\langle * \overset{l}{\Leftarrow} S \to T \rangle\!\rangle = \langle\!\langle S \overset{\bar{l}}{\Leftarrow} * \rangle\!\rangle \to^\epsilon \langle\!\langle * \overset{l}{\Leftarrow} T \rangle\!\rangle$$

Compile blame terms to threesome terms $\boxed{\langle\!\langle t \rangle\!\rangle = t}$

$$\langle\!\langle x \rangle\!\rangle = x \quad \langle\!\langle k \rangle\!\rangle = k \quad \langle\!\langle \lambda x\!:\!S.\ t \rangle\!\rangle = \lambda x : S.\ \langle\!\langle t \rangle\!\rangle \quad \langle\!\langle t\ s \rangle\!\rangle = \langle\!\langle t \rangle\!\rangle\ \langle\!\langle s \rangle\!\rangle$$
$$\langle\!\langle \langle T \overset{l}{\Leftarrow} S \rangle s \rangle\!\rangle = \langle T \overset{R}{\Longleftarrow} S \rangle \langle\!\langle s \rangle\!\rangle \quad \text{where } R = \langle\!\langle T \overset{l}{\Leftarrow} S \rangle\!\rangle$$

Figure 5. Compilation of twosomes to threesomes (with blame)

$$k \approx k \qquad x \approx x \qquad \frac{s_2 \approx s_3 \qquad t_2 \approx t_3}{s_2\ t_2 \approx s_3\ t_3}$$
$$\frac{s_2 \approx s_3}{\lambda x : S.\ s_2 \approx \lambda x : S.\ s_3} \qquad \textbf{blame}\ l \approx \textbf{blame}\ l$$
$$\frac{s_2 \approx s_3 \qquad P = \langle\!\langle T \overset{l}{\Leftarrow} S \rangle\!\rangle}{\langle T \overset{l}{\Leftarrow} S \rangle s_2 \approx \langle T \overset{P}{\Longleftarrow} S \rangle s_3} \tag{24}$$
$$\frac{s_2 \approx \langle T \overset{P}{\Longleftarrow} S \rangle s_3 \qquad Q = \langle\!\langle U \overset{l}{\Leftarrow} T \rangle\!\rangle}{\langle U \overset{l}{\Leftarrow} T \rangle s_2 \approx \langle U \overset{Q \circ P}{\Longleftarrow} S \rangle s_3} \tag{25}$$
$$\frac{t_2 \approx (\langle T_1 \to T_2 \overset{P}{\Longleftarrow} S \rangle s_3)\ (\langle T_1 \overset{\langle\!\langle T_1 \overset{l}{\Leftarrow} U_1 \rangle\!\rangle}{\Longleftarrow} U_1 \rangle t_3) \qquad Q = \langle\!\langle U_1 \to U_2 \overset{l}{\Leftarrow} T_1 \to T_2 \rangle\!\rangle \qquad \neg(T_1 \to T_2 = |P| = S)}{\langle U_2 \overset{l}{\Leftarrow} T_2 \rangle t_2 \approx (\langle U_1 \to U_2 \overset{Q \circ P}{\Longleftarrow} S \rangle s_3)\ t_3} \tag{26}$$
$$\frac{t_2 \approx s_3\ (\langle S_1 \overset{\langle\!\langle S_1 \overset{l}{\Leftarrow} U_1 \rangle\!\rangle}{\Longleftarrow} U_1 \rangle t_3) \qquad Q = \langle\!\langle U_1 \to U_2 \overset{l}{\Leftarrow} S_1 \to S_2 \rangle\!\rangle}{\langle U_2 \overset{l}{\Leftarrow} S_2 \rangle t_2 \approx (\langle U_1 \to U_2 \overset{Q}{\Longleftarrow} S_1 \to S_2 \rangle s_3)\ t_3} \tag{27}$$

Figure 6. Bisimulation relating twosomes and threesomes

relation, shown in Figure 6, linking reduction in the two systems. The bisimulation relates a sequence of twosomes with a single threesome. To accomplish this, rule (25) peels off a twosome from the sequence and recursively relates a modified threesome with the rest of the sequence. The modification removes the contribution of the one twosome from the threesome's intermediate type.

Rule (26) is necessary to relate states during the application of a cast-wrapped function to a value. On the blame calculus side, the function is wrapped in a sequence of casts and there is a sequence of reductions via rule (8). On the threesome side, there is only one cast and one reduction via rule (19). Rule (26) therefore relates the intermediate steps of the blame calculus back to the state of the threesomes before the reduction via rule (19). Rule (27) is similar to rule (26) but handles the special case when $T_1 \to T_2 = |P| = S$.

Lemma 8 (Bisimulation between the blame and threesome calculi)**.** *If $s_2 \approx s_3$ and both s_2 and s_3 are well typed, then*

1. *if $s_2 \longmapsto r_2$, then $s_3 \longmapsto^* r_3$ and $r_2 \approx r_3$ for some r_3.*
2. *if $s_3 \longmapsto r_3$, then $s_2 \longmapsto^* r_2$ and $r_2 \approx r_3$ for some r_2.*

Towards proving the correctness of the threesome calculus, we show that compilation returns a threesome calculus program that is bisimilar to the source program.

Lemma 9 (Compilation returns a bisimilar program.)**.** $t \approx \langle\!\langle t \rangle\!\rangle$

Also, we show that the bisimulation is sound with respect to observable behavior.

Lemma 10. *If $t_2 \approx t_3$, then $t_2 \downarrow$ if and only if $t_3 \downarrow$.*

The correctness of the threesome calculus is a straightforward consequence of these lemmas.

Theorem 4 (Correctness of the threesome calculus)**.**
$\langle\!\langle t \rangle\!\rangle \downarrow$ if and only if $t \downarrow$.

4.2 Space Efficiency

The addition of labels to the intermediate types of the threesomes only adds a constant factor increase in space, so the proof of space efficiency from Section 3.2 carries over to the full threesome calculus in a straightforward manner.

5. Relation to the Coercion Calculus

Henglein (1994) introduces a sub-language named the *coercion calculus* to express casts. Instead of casts of the form $\langle T \Leftarrow S \rangle s$, Henglein uses casts of the form $\langle c \rangle s$ where c is a term of the coercion calculus. The coercion calculus is not intended to be directly used by programmers, but instead casts of the form $\langle T \Leftarrow S \rangle s$ are compiled into casts of the form $\langle c \rangle s$. The coercion calculus predates blame tracking, but Siek et al. (2009) augment the coercion calculus with blame, obtaining the calculus shown in Figure 7.

The coercion $G!$ injects a value into $*$ whereas the coercion $G?^l$ projects a value out of $*$, blaming location l in the case of a type mismatch. The identity coercion ι_T behaves like the identity function. The type annotation T on the identity coercion is always either a ground type or $*$. A function coercion $c \to d$ 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. We consider coercions equal up to associativity of composition. In addition to Henglein's coercions, there is the $\mathtt{Fail}^l_{T \Leftarrow S}$ coercion of Herman et al. (2007), which compactly represents coercions that are destined to fail but have not yet been applied to a value.

The coercion reduction system we present here corresponds to Henglein's notion of ϕ-reduction: $G?^l \circ G! \longrightarrow \iota_G$. Henglein also studied ψ-reduction: $G! \circ G?^l \longrightarrow \iota_G$. While ψ-reduction is useful when considering how best to compile dynamically typed languages (as in Henglein's work), ψ-reduction is not suitable in our setting as it would allow some errors to go uncaught.

Rule (34) for composing failures on the left is subtle. Herman et al. (2007) instead use the reduction

$$\mathtt{Fail} \circ c \longrightarrow \mathtt{Fail}$$

However, with the addition of blame tracking, that rule would make the calculus non-deterministic. Siek et al. (2009) restrict c to injections.

$$\mathtt{Fail}^l_{T \Leftarrow S} \circ G! \longrightarrow \mathtt{Fail}^l_{T \Leftarrow G}$$

But that rule is useless because failure coercions never have $*$ as a source type. To see why this is the case, first consider rule (29) which introduces failures: the source type is G. Next, the only other way to obtain a failure with source type $*$ would be to have a rule of the form

$$\mathtt{Fail}^l_{T \Leftarrow S} \circ G?^p \longrightarrow \mathtt{Fail}^l_{T \Leftarrow *} \qquad \text{(Hypothetical!)}$$

but we certainly don't want such a rule because that would introduce non-determinism. (The presence of the useless rule in the semantics of Siek et al. (2009) does not invalidate any of their results.)

In this paper, because we are using lazy cast checking, we can allow the coercion on the right to be a function coercion. With lazy checking, any errors in the domain or codomain of a cast appear after errors concerning the function type itself, even if the error concerning the function type appears later in a series of casts. For example, the following program allocates blame to l_4 even though there is also a potential error at l_2.

$$\langle \mathtt{Bool} \overset{l_4}{\Leftarrow} * \overset{l_3}{\Leftarrow} \mathtt{Int} \to \mathtt{Bool} \overset{l_2}{\Leftarrow} * \overset{l_1}{\Leftarrow} \mathtt{Int} \to \mathtt{Int} \rangle \lambda x : \mathtt{Int}.\ x \longmapsto \mathbf{blame}\ l_4$$

We give an inductive characterization of the ϕ-normal forms of the coercion calculus in Figure 7, writing $\mathsf{nm}\ c$ for this inductive predicate.

Proposition 6.
Suppose $\vdash c : T \Leftarrow S$. $\mathsf{nm}\ c$ if and only if $\nexists c'.\ c \longmapsto c'$.

Figure 9 gives the definition of a coercion-based calculus with blame tracking similar to that of Siek et al. (2009). In this presentation we choose to keep coercions in normal form and perform coercion normalization in rule (39).

The correspondence between the threesome calculus and the coercion-based calculus is quite strong: threesome casts are isomorphic to well-typed coercions in normal form. Figure 8 defines the function and its inverse that witnesses this isomorphism.

Proposition 7 ($(\!|\cdot|\!)$ produces a coercion in normal form.)**.**
If $P <:_n T$ and $P <:_n S$, then $\mathsf{nm}\ (\!|T \overset{P}{\Longleftarrow} S|\!)$.

Lemma 11 (Bijection between threesomes and coercions)**.**
If $|P| <:_n S$ and $|P| <:_n T$ and $\mathsf{nm}\ c$, then $(\!|(\!|T \overset{P}{\Longleftarrow} S|\!)|\!)^{-1} = \langle T \overset{P}{\Longleftarrow} S \rangle$, and $(\!|(\!|c|\!)^{-1}|\!) = c$.

The isomorphism is structure-preserving with respect to reduction. To prove that the coercion-based reduction rule (39) is equivalent to the threesome rule (23), we need to establish that

$$(\!|T \overset{Q}{\Longleftarrow} S'|\!) \circ (\!|S' \overset{P}{\Longleftarrow} S|\!) \longmapsto^* (\!|T \overset{Q \circ P}{\Longleftarrow} S|\!)$$

To prove this, we need to establish that mapping from coercions back to threesomes is invariant with respect to coercion reduction.

Lemma 12 (Coercion reduction preserves $(\!|\cdot|\!)^{-1}$.)**.**
Suppose c is well-typed. If $c \longmapsto^* c'$, then $(\!|c|\!)^{-1} = (\!|c'|\!)^{-1}$.

Corollary 2. $(\!|T \overset{Q}{\Longleftarrow} R|\!) \circ (\!|R \overset{P}{\Longleftarrow} S|\!) \longmapsto^* (\!|T \overset{Q \circ P}{\Longleftarrow} S|\!)$

Proof. First, we have $(\!|(\!|T \overset{Q}{\Longleftarrow} R|\!) \circ (\!|R \overset{P}{\Longleftarrow} S|\!)|\!)^{-1} = T \overset{Q \& P}{\Longleftarrow} S$. Then because coercion reduction is strongly normalizing, there is a c' such that $(\!|T \overset{Q}{\Longleftarrow} R|\!) \circ (\!|R \overset{P}{\Longleftarrow} S|\!) \longmapsto^* c'$ and $\mathsf{nm}\ c$. Then by Lemma 12 we have $(\!|c'|\!)^{-1} = T \overset{Q \& P}{\Longleftarrow} S$ and therefore $c' = (\!|T \overset{Q \& P}{\Longleftarrow} S|\!)$. □

Putting these results together, we have an isomorphism between the threesome calculus and the coercion-based calculus of Figure 7.

Theorem 5 (Isomorphism between the threesome calculus and the coercion-based calculus)**.**

1. *$(\!|\cdot|\!)$ is bijective, that is $(\!|(\!|t_3|\!)|\!)^{-1} = t_3$ and $(\!|(\!|t_c|\!)^{-1}|\!) = t_c$ given that t_3 and t_c are well-typed.*
2. *$(\!|\cdot|\!)$ is structure (reduction) preserving, that is, $t_3 \longmapsto t_3'$ if and only if $(\!|t_3|\!) \longmapsto (\!|t_3'|\!)$.*

Syntax

$$\begin{array}{lrcl} \text{coercions} & c, d & ::= & \iota_T \mid G! \mid G?^l \mid d \circ c \mid c{\rightarrow}d \mid \mathtt{Fail}^l_{T \Leftarrow S} \\ \text{contexts} & C & ::= & C \circ c \mid d \circ C \mid C{\rightarrow}d \mid c{\rightarrow}C \end{array}$$

Well-typed coercions $\boxed{c : T \Leftarrow S}$

$$\frac{}{\iota_T : T \Leftarrow T} \qquad \frac{}{\mathtt{Fail}^l_{T \Leftarrow S} : T \Leftarrow S} \qquad \frac{}{G! : * \Leftarrow G}$$

$$\frac{}{G?^l : G \Leftarrow *} \qquad \frac{c : S \Leftarrow T \quad d : T' \Leftarrow S'}{c{\rightarrow}d : (T{\rightarrow}T') \Leftarrow (S{\rightarrow}S')}$$

$$\frac{d : U \Leftarrow T \quad \vdash c : T \Leftarrow S}{d \circ c : U \Leftarrow S}$$

Additional typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash s : S \quad c : T \Leftarrow S}{\Gamma \vdash \langle c \rangle s : T}$$

Compile casts to coercions $\boxed{\langle\!\langle T \overset{l}{\Leftarrow} S \rangle\!\rangle = c}$

$$\langle\!\langle B \overset{l}{\Leftarrow} B \rangle\!\rangle = \iota_B \qquad \langle\!\langle * \overset{l}{\Leftarrow} * \rangle\!\rangle = \iota_*$$

$$\langle\!\langle * \overset{l}{\Leftarrow} B \rangle\!\rangle = B! \qquad \langle\!\langle B \overset{l}{\Leftarrow} * \rangle\!\rangle = B?^l$$

$$\langle\!\langle * \overset{l}{\Leftarrow} S \to T \rangle\!\rangle = (*{\rightarrow}*)! \circ (\langle\!\langle S \overset{\bar{l}}{\Leftarrow} * \rangle\!\rangle {\rightarrow} \langle\!\langle * \overset{l}{\Leftarrow} T \rangle\!\rangle)$$

$$\langle\!\langle S \to T \overset{l}{\Leftarrow} * \rangle\!\rangle = (\langle\!\langle * \overset{\bar{l}}{\Leftarrow} S \rangle\!\rangle {\rightarrow} \langle\!\langle T \overset{l}{\Leftarrow} * \rangle\!\rangle) \circ (*{\rightarrow}*)?^l$$

$$\langle\!\langle S' \to T' \overset{l}{\Leftarrow} S \to T \rangle\!\rangle = \langle\!\langle S \overset{\bar{l}}{\Leftarrow} S' \rangle\!\rangle {\rightarrow} \langle\!\langle T' \overset{l}{\Leftarrow} T \rangle\!\rangle$$

Coercion reductions $\boxed{c \longrightarrow c}$

$$G?^l \circ G! \longrightarrow \iota_G \tag{28}$$

$$H?^l \circ G! \longrightarrow \mathtt{Fail}^l_{H \Leftarrow G} \quad \text{if } G \neq H \tag{29}$$

$$(d_1{\rightarrow}d_2) \circ (c_1{\rightarrow}c_2) \longrightarrow (c_1 \circ d_1){\rightarrow}(d_2 \circ c_2) \tag{30}$$

$$\iota_T \circ c \longrightarrow c \tag{31}$$

$$d \circ \iota_T \longrightarrow d \tag{32}$$

$$d \circ \mathtt{Fail}^l_{T \Leftarrow S} \longrightarrow \mathtt{Fail}^l_{U \Leftarrow S} \quad \text{if } d : U \Leftarrow T \tag{33}$$

$$\mathtt{Fail}^l_{U \Leftarrow T} \circ (c{\rightarrow}d) \longrightarrow \mathtt{Fail}^l_{U \Leftarrow S} \quad \text{if } c{\rightarrow}d : T \Leftarrow S \tag{34}$$

Single-step evaluation $\boxed{c \longmapsto c}$

$$\frac{c = C[c_0] \quad c_0 \longrightarrow c_1 \quad c' = C[c_1]}{c \longmapsto c'}$$

Normal forms $\boxed{\mathsf{nm}\ c}$

$$\mathsf{nm}\ \iota_T \qquad \mathsf{nm}\ G?^l \qquad \mathsf{nm}\ G! \qquad \mathsf{nm}\ G! \circ G?^l$$

$$\mathsf{nm}\ \mathtt{Fail}^l_{T \Leftarrow S} \qquad \mathsf{nm}\ \mathtt{Fail}^l_{T \Leftarrow G} \circ G?^m$$

$$\frac{\mathsf{nm}\ c \quad \mathsf{nm}\ d}{\mathsf{nm}\ (*{\rightarrow}*)! \circ (c{\rightarrow}d)} \qquad \frac{\mathsf{nm}\ c \quad \mathsf{nm}\ d}{\mathsf{nm}\ c{\rightarrow}d}$$

$$\frac{\mathsf{nm}\ c \quad \mathsf{nm}\ d}{\mathsf{nm}\ (c{\rightarrow}d) \circ (*{\rightarrow}*)?^l} \qquad \frac{\mathsf{nm}\ c \quad \mathsf{nm}\ d}{\mathsf{nm}\ (*{\rightarrow}*)! \circ (c{\rightarrow}d) \circ (*{\rightarrow}*)?^l}$$

Figure 7. Coercion calculus with blame

Map threesomes to coercions $\boxed{(\!|T \overset{P}{\Longleftarrow} S|\!) = c}$

$$(\!|G \overset{G^\epsilon}{\Longleftarrow} G|\!) = \iota_G$$

$$(\!|* \overset{*}{\Longleftarrow} *|\!) = \iota_*$$

$$(\!|G \overset{G^l}{\Longleftarrow} *|\!) = G?^l$$

$$(\!|* \overset{G^\epsilon}{\Longleftarrow} G|\!) = G!$$

$$(\!|* \overset{G^l}{\Longleftarrow} *|\!) = G! \circ G?^l$$

$$(\!|T \overset{\bot^{lH\epsilon}}{\Longleftarrow} S|\!) = \mathtt{Fail}^l_{T \Leftarrow S} \qquad \text{if } S \neq *$$

$$(\!|T \overset{\bot^{lGm}}{\Longleftarrow} *|\!) = \mathtt{Fail}^l_{T \Leftarrow G} \circ G?^m$$

$$(\!|T_1{\rightarrow}T_2 \overset{P_1 \overset{\epsilon}{\rightarrow} P_2}{\Longleftarrow} S_1{\rightarrow}S_2|\!) = (\!|S_1 \overset{P_1}{\Longleftarrow} T_1|\!){\rightarrow}(\!|T_2 \overset{P_2}{\Longleftarrow} S_2|\!)$$

$$(\!|T_1{\rightarrow}T_2 \overset{P_1 \overset{l}{\rightarrow} P_2}{\Longleftarrow} *|\!) = (\!|T_1{\rightarrow}T_2 \overset{P_1 \overset{\epsilon}{\rightarrow} P_2}{\Longleftarrow} *{\rightarrow}*|\!) \circ (*{\rightarrow}*)?^l$$

$$(\!|* \overset{P_1 \overset{l}{\rightarrow} P_2}{\Longleftarrow} *|\!) = (*{\rightarrow}*)! \circ (\!|*{\rightarrow}* \overset{P_1 \overset{\epsilon}{\rightarrow} P_2}{\Longleftarrow} *{\rightarrow}*|\!) \circ (*{\rightarrow}*)?^l$$

$$(\!|* \overset{P_1 \overset{\epsilon}{\rightarrow} P_2}{\Longleftarrow} S_1{\rightarrow}S_2|\!) = (*{\rightarrow}*)! \circ (\!|*{\rightarrow}* \overset{P_1 \rightarrow P_2}{\Longleftarrow} S_1{\rightarrow}S_2|\!)$$

Map coercions to threesomes $\boxed{(\!|c|\!)^{-1} = \langle S \overset{P}{\Longleftarrow} T \rangle}$

$$(\!|\iota_G|\!)^{-1} = \langle G \overset{G^\epsilon}{\Longleftarrow} G \rangle$$

$$(\!|\iota_*|\!)^{-1} = \langle * \overset{*}{\Longleftarrow} * \rangle$$

$$(\!|G!|\!)^{-1} = \langle * \overset{G^\epsilon}{\Longleftarrow} G \rangle$$

$$(\!|G?^l|\!)^{-1} = \langle G \overset{G^l}{\Longleftarrow} * \rangle$$

$$(\!|\mathtt{Fail}^l_{T \Leftarrow S}|\!)^{-1} = \langle T \overset{\bot^{lG\epsilon}}{\Longleftarrow} S \rangle \quad \text{where } gnd(S) = G$$

$$(\!|c{\rightarrow}d|\!)^{-1} = \langle T_1{\rightarrow}T_2 \overset{P_1 \overset{\epsilon}{\rightarrow} P_2}{\Longleftarrow} S_1{\rightarrow}S_2 \rangle$$

$$\text{where } \langle T_1 \overset{P_1}{\Longleftarrow} S_1 \rangle = (\!|c|\!), \langle T_2 \overset{P_2}{\Longleftarrow} S_2 \rangle = (\!|d|\!)$$

$$(\!|d \circ c|\!)^{-1} = \langle T \overset{Q \circ P}{\Longleftarrow} S \rangle$$

$$\text{where } \langle R \overset{P}{\Longleftarrow} S \rangle = (\!|c|\!), \langle T \overset{Q}{\Longleftarrow} R \rangle = (\!|d|\!)$$

Figure 8. Isomorphism between threesomes and coercions

6. Related Work

The integration of static and dynamic typing has roots in the 1970's and 1980's, with the `any` type and `force` expression in CLU (Liskov et al. 1979) and the `Dynamic` type and `coerce` expression in Amber (Cardelli 1986). Similar constructs appear in Cedar (Lampson 1983), Modula-2 (Rovner 1986), and Modula-3 (Cardelli et al. 1989). Casts are generalized to `typecase` in Modula-2 and 3 their semantics is formalized by Abadi et al. (1989, 1991). In this early work on casts and the `Dynamic` type, run-time checks are based on type equality. Thatte (1990) observes that type equality requires programmers to explicitly reason about the run-time tags on values. To relax this restriction he proposes that casts succeed whenever the value can be coerced to the target type, similar to the coercion based models of subtyping (Mitchell 1984, Breazu-Tannen et al. 1989). Thatte's work includes upcasts and downcasts, with `Dynamic` as top.

In studying the compilation of dynamically typed languages to statically typed languages, Henglein (1992, 1994) develops the coercion calculus, a sub-language for expressing casts involving the type `Dynamic`. We notice similarities between Henglein's coercion calculus and the retracts of Scott (1976) but we are not aware of any work that formally connects retracts and the coercion calculus.

Henglein shows that his coercions factor: any coercion is ϕ-reducible to a sequence consisting of a negative, neutral, and positive coercion: $c^+ \circ c^* \circ c^-$. A negative coercion is a downcast with

Syntax

terms	s, t	$::=$	$k \mid x \mid \lambda x{:}S.t \mid s\ t \mid \langle c\rangle s$ where $\mathsf{nm}\ c$
uncoerced values	u	$::=$	$k \mid \lambda x : S.t$
values	v, w	$::=$	$u \mid \langle c\rangle u$ where $\mathsf{nm}\ c$
contractums	r	$::=$	$t \mid \mathbf{blame}\ l$
cast-free contexts	F	$::=$	$\Box \mid E[\Box\ t] \mid E[v\ \Box]$
evaluation contexts	E	$::=$	$F \mid F[\langle c\rangle\Box]$

Additional typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash s : S \quad \vdash c : T \Leftarrow S}{\Gamma \vdash \langle c\rangle s : T}$$

Compile blame calculus to coercion-based calculus $\boxed{\langle\!\langle t\rangle\!\rangle = t}$

$$\langle\!\langle x\rangle\!\rangle = x \qquad \langle\!\langle k\rangle\!\rangle = k \qquad \langle\!\langle \lambda x : S.\ t\rangle\!\rangle = \lambda x : S.\ \langle\!\langle t\rangle\!\rangle$$

$$\langle\!\langle t\ s\rangle\!\rangle = \langle\!\langle t\rangle\!\rangle\ \langle\!\langle s\rangle\!\rangle \qquad \langle\!\langle \langle T \overset{l}{\Leftarrow} S\rangle s\rangle\!\rangle = \langle \langle\!\langle T \overset{l}{\Leftarrow} S\rangle\!\rangle \langle\!\langle s\rangle\!\rangle$$

Reductions $\boxed{s \longrightarrow r}$

$$(\lambda x : S.t)\ v \longrightarrow [x := v]t \tag{35}$$

$$k\ k' \longrightarrow \delta(k, k') \tag{36}$$

$$(\langle c \to d\rangle u)\ w \longrightarrow \langle d\rangle\ u(\langle c\rangle w) \tag{37}$$

Cast reductions $\boxed{t \longrightarrow_c r}$

$$\langle \iota_T\rangle u \longrightarrow_c u \tag{38}$$

$$\langle d\rangle\langle c\rangle s \longrightarrow_c \langle c'\rangle s \qquad \text{if } d \circ c \longmapsto^* c' \text{ and } \mathsf{nm}\ c' \tag{39}$$

$$\langle \mathtt{Fail}^l_{T \Leftarrow S}\rangle u \longrightarrow \mathbf{blame}\ l \tag{40}$$

Single-step evaluation $\boxed{t \longmapsto r}$

$$\frac{t \longrightarrow t'}{E[t] \longmapsto E[t']} \qquad \frac{t \longrightarrow_c t'}{F[e] \longmapsto F[t']} \qquad \frac{t \longrightarrow_c \mathbf{blame}\ l}{F[t] \longmapsto \mathbf{blame}\ l}$$

Figure 9. Coercion-based lambda calculus with blame

respect to $<:_n$, a positive coercion is an upcast, and a neutral coercion represents failure. We conjecture that a threesome $\langle T \overset{R}{\Longleftarrow} S\rangle$, where R does not contain $\bot$, is equivalent to a cast with factoring $c^+ \circ c^* \circ c^-$ where c^- is a downcast from S to R, and c^* is the identity coercion on R, and c^+ is an upcast from R to T.

The primary goal of Henglein's work was to minimize the number of casts inserted during compilation, which explains his use of the ψ-reduction to remove projection-injection pairs. In contrast, the focus of this paper is on integrating static and dynamic typing, and the boundary between static and dynamic regions require certain run-time checks to maintain the integrity of the static region. To maintain the integrity of the static types, we only use ϕ-reduction. That being said, an interesting direction for future work would be to try and reduce the number of inserted casts using Henglein's techniques while remaining sound.

Gray et al. (2005) propose adding the type `Dynamic` to Java and apply the technology of contract checking to casts. Towards formalizing the ideas of Gray et al. (2005), Siek and Taha (2006) and Gronski et al. (2006) generalize Thatte's up and down-casts to a single cast capable of up, down, and cross casts. Furthermore, Siek and Taha (2006) observe that subtyping is not suitable for statically characterizing casts involving type `Dynamic` and propose the compatibility relation, written $\sim$, to fill this role. These generalized casts play an important role in the intermediate languages of *gradual typing* (Siek and Taha 2006, 2007) and *hybrid typing* (Gronski et al. 2006, Flanagan 2006, Flanagan et al. 2006).

Tobin-Hochstadt and Felleisen (2006) formalize the interaction between static and dynamic typing at the granularity of modules and develop a precursor to the Blame Theorem. Wadler and Findler (2007, 2009) design the blame calculus, adding blame tracking to the generalized casts of Siek and Taha (2006) and Gronski et al. (2006), drawing on the blame tracking of higher-order contracts (Findler and Felleisen 2002). Wadler and Findler (2009) formulate and prove the Blame Theorem: statically typed regions of a program can't be blamed for a cast error. Gronski and Flanagan (2007) show that casts with a single blame label are just as expressive as casts with two blame labels or labels with polarity.

Herman et al. (2007) observe that in a straightforward implementation of the generalized casts, there can be unbounded runtime space overhead. To solve this problem, Herman et al. (2007) perform stack inspection to find sequences of casts and use the coercion calculus of Henglein (1994) to compress those sequences. Siek and Taha (2007) describe a partial solution to the space efficiency problem using a merge operator that is a precursor to the meet algorithm given in this paper. Siek et al. (2009) show how to augment the coercion calculus with blame tracking, thereby obtaining a theoretical bound on space consumption for blame calculus implementations.

7. Conclusion

In this paper we present threesomes, a typed-based solution to the space-efficiency problem of higher-order casts. We first present a simplified threesome calculus where sequences of casts are compressed by taking the greatest lower bound with respect to naive subtyping. We prove that the simplified threesome calculus is equivalent to the blame calculus (ignoring blame allocation) and we prove that the simplified threesome calculus is space efficient. We then present the threesome calculus with full support for blame tracking and prove that it is equivalent to the blame calculus. We prove that threesomes are isomorphic to Henglein's coercions in normal form and that threesome composition is equivalent to coercion composition followed by normalization.

References

M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–227, New York, NY, USA, 1989. ACM.

Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance and explicit coercion. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 112–129, Piscataway, NJ, USA, 1989. IEEE Press.

L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The modula–3 type system. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 202–212, New York, NY, USA, 1989. ACM.

Luca Cardelli. Amber. In *Combinators and Functional Programming Languages*, volume 242, pages 21–70, 1986.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ACM International Conference on Functional Programming*, October 2002.

Cormac Flanagan. Hybrid type checking. In *POPL 2006: The 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, Charleston, South Carolina, January 2006.

Cormac Flanagan, Stephen N. Freund, and Aaron Tomb. Hybrid types, invariants, and refinements for imperative objects. In *FOOL/WOOD '06:*

International Workshop on Foundations and Developments of Object-Oriented Languages, 2006.

Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 231–245, New York, NY, USA, 2005. ACM Press.

Jessica Gronski and Cormac Flanagan. Unifying hybrid types and contracts. In *Trends in Functional Prog. (TFP)*, 2007.

Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

Lars T. Hansen. Evolutionary programming and gradual typing in ECMAScript 4 (tutorial). Technical report, ECMA TG1 working group, November 2007.

Anders Hejlsberg. The future of C#. http://channel9.msdn.com/pdc2008/TL16/, October 2008.

Fritz Henglein. Dynamic typing. In *ESOP '92: Proceedings of the 4th European Symposium on Programming*, pages 233–253, London, UK, 1992. Springer-Verlag.

Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.

David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, page XXVIII, April 2007.

Butler W. Lampson. A description of the Cedar language. Technical report, Xerox PARC, 1983.

Barbara Liskov, Russ Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler, and Alan Snyder. CLU reference manual. Technical Report LCS-TR-225, MIT, October 1979.

John C. Mitchell. Coercion and type inference. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 175–185, New York, NY, USA, 1984. ACM Press.

P. Rovner. Extending Modula-2 to build large, integrated systems. *IEEE Software*, 3(6):46–57, 1986. ISSN 0740-7459. doi: http://doi.ieeecomputersociety.org/10.1109/MS.1986.229476.

Dana Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3): 522–587, 1976.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *ECOOP 2007*, volume 4609 of *LCNS*, pages 2–27. Springer Verlag, August 2007.

Jeremy G. Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In *European Symposium on Programming*, March 2009.

Satish Thatte. Quasi-static typing. In *POPL 1990*, pages 367–381, New York, NY, USA, 1990. ACM Press.

Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *OOPSLA'06 Companion*, pages 964–974, NY, 2006. ACM.

Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *POPL 2008: The 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2008.

Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Workshop on Scheme and Functional Programming*, pages 15–26, 2007.

Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, 2009.

Larry Wall. Synopsis 2: Bits and pieces. Technical report, June 2009.

Tobias Wrigstad, Patrick Eugster, John Field, Nate Nystrom, and Jan Vitek. Software hardening: A research agenda. In *International Workshop on Script to Program Evolution*, July 2009.

Appendix

7.1 Correctness of Simplified Threesomes

Proof of Proposition 1. The proof is by strong induction on the sum of the height of S and T. □

We define the append operation @ for contexts as follows.

$$\begin{aligned} E@\Box &= E \\ E@E'[\Box\ t] &= (E@E')[\Box\ t] \\ E@E'[v\ \Box] &= (E@E')[v\ \Box] \\ E@F[\langle T \overset{R}{\Longleftarrow} S\rangle^l \Box] &= (E@F)[\langle T \overset{R}{\Longleftarrow} S\rangle^l \Box] \\ &\qquad \text{if } E@F \neq F'[\langle T' \overset{R'}{\Longleftarrow} S'\rangle^l \Box] \end{aligned}$$

Proof of Unique Decomposition 2. The proof is by induction on the typing derivation $\emptyset \vdash t : T$.

1. $\emptyset \vdash k : typeof(k)$: k is a value.
2. $\emptyset \vdash x : T$: vacuously true.
3. $\emptyset \vdash \lambda x : S.s$: $\lambda x : S.s$ is a value.
4. $\dfrac{\emptyset \vdash s : T \to S \quad \emptyset \vdash t : T}{\emptyset \vdash s\ t : S}$: Either s is a value, or not.
   (a) Suppose s is a value. Either t is a value, or not.
      i. Suppose t is a value. By the canonical forms lemma, s is either a function or a functional cast. In either case $s\ t$ is a redex. Furthermore, the only context that decomposes $s\ t$ into a redex is $\Box$.
      ii. Suppose t is not a value. Then we apply the induction hypothesis to get a unique context E and redex t' where $t = E[t']$. The unique context for $s\ \ t$ is therefore $\Box[s\Box]@E$.
   (b) Suppose s is not a value. Then we apply the induction hypothesis to get a unique context E and redex s' where $s = E[s']$. The unique context for $s\ t$ is therefore $\Box[\Box t]@E$.
5. $\dfrac{\emptyset \vdash s : S}{\emptyset \vdash \langle T \overset{R}{\Longleftarrow} S\rangle s : T}$: Either s is a value, or not.
   (a) Suppose s is a value. Then either $\langle T \overset{R}{\Longleftarrow} S\rangle s$ is also a value or it is a redex. If it is a redex, then the unique context is $\Box$.
   (b) Suppose s is not a value. We have two cases to consider. If s is a cast, then $\langle T \overset{R}{\Longleftarrow} S\rangle s$ is a redex and the unique context is $\Box$. (Any other context that puts the hole inside s would contain adjacent casts, which is impossible.) If s is not a cast, we apply the induction hypothesis to get a unique context E and redex s' where $s = E[s']$. The unique context for $\langle T \overset{R}{\Longleftarrow} S\rangle s$ is therefore $\Box[\langle T \overset{R}{\Longleftarrow} S\rangle \Box]@E$. Because s is not a cast, the top of E is not a cast and appending $\Box[\langle T \overset{R}{\Longleftarrow} S\rangle \Box]$ with E will not result in adjacent casts and therefore produces a well-formed context. □

Towards proving the fundamental property of casts, we define the bisimulation relation $\dot{\approx}$ by closing the following two rules with respect to contexts. The second rule is necessary for $\dot{\approx}$ to be a bisimulation.

$$\frac{t \mathrel{\dot{\approx}} t' \qquad T\&S <:_n R}{\langle T \Leftarrow S\rangle t \mathrel{\dot{\approx}} \langle T \Leftarrow R \Leftarrow S\rangle t'} \qquad \frac{t \mathrel{\dot{\approx}} t'}{t \mathrel{\dot{\approx}} \langle S \to T \Leftarrow S \to T\rangle t'}$$

Lemma 13 (Substitution preserves $\dot{\approx}$)**.** *If $s \mathrel{\dot{\approx}} t$ and $v_s \mathrel{\dot{\approx}} v_t$, then $[x := v_s]s \mathrel{\dot{\approx}} [x := v_t]t$.*

Proof. The proof is by induction on the derivation of $s \mathrel{\dot{\approx}} t$. □

Lemma 14 ($\dot{\approx}$ is a weak bisimulation.)**.** *Suppose $s \mathrel{\dot{\approx}} t$.*

1. If $s \longmapsto r_s$, then there exists r_t where $t \longmapsto^* r_t$ and $r_s \mathrel{\dot{\approx}} r_t$.

2. *If $t \longmapsto r_t$, then there exists r_s where $s \longmapsto^* r_s$ and $r_s \mathrel{\dot{\approx}} r_t$.*

Proof.

1. The proof is by a lengthy case analysis on $s \longmapsto r_s$. The case for function application uses Lemma 13.
2. The proof is by a lengthy case analysis on $t \longmapsto r_t$. The case for function application uses Lemma 13. □

Lemma 15. *If $s \mathrel{\dot{\approx}} t$, then $s \downarrow$ if and only if $t \downarrow$.*

Proof.

1. ($\Longrightarrow$) The proof is by induction on the reduction $s \longmapsto^* f$.
   (a) Case $s = f$: From $s \mathrel{\dot{\approx}} t$ and Lemma 14 we have $t \longmapsto^* f'$ and therefore $t \downarrow$.
   (b) Case $s \longmapsto s'$ and $s' \longmapsto^* f$: From $s \mathrel{\dot{\approx}} t$ and Lemma 14 we have $t \longmapsto^* t'$ with $s' \mathrel{\dot{\approx}} t'$. Then by the induction hypothesis, $t \downarrow$.
2. ($\Longleftarrow$) The proof is symmetric to the above case. □

Lemma 16. *If $s \mathrel{\dot{\approx}} t$, then $C[s] \mathrel{\dot{\approx}} C[t]$.*

Proof. The proof by induction on C. □

Lemma 17. *If $s \mathrel{\dot{\approx}} t$, then $s =_{\mathrm{ctx}} t$.*

Proof. We fix C and need to show that $C[s] \downarrow$ if and only if $C[t] \downarrow$. By Lemma 16 we have $C[s] \mathrel{\dot{\approx}} C[t]$. We conclude by applying Lemma 15. □

Lemma 18. $t \mathrel{\dot{\approx}} t$

Proof. By induction on t. □

Proof of Lemma 2. Assuming $T \& S <:_n R$, we need to show that $\langle T \Leftarrow S \rangle t =_{\mathrm{ctx}} \langle T \Leftarrow R \Leftarrow S \rangle t$. We have $t \mathrel{\dot{\approx}} t$ and therefore $\langle T \Leftarrow S \rangle t \mathrel{\dot{\approx}} \langle T \Leftarrow R \Leftarrow S \rangle t$. We conclude by Lemma 17. □

7.2 Proof of Space Efficiency

Proof of Proposition 3.

1. The proof is a straightforward induction on the type T.
2. The proof is by strong induction on the sum of the heights of S and T.
3. The proof is by induction on type T with case analysis on S. □

Proof of Lemma 6. The proof is by inversion on $t \longmapsto t'$ and cases on $\longrightarrow$. The case for rule (16) uses Proposition 3 (part 2). □

Proof of Lemma 7. The proof is by induction on t. The only interesting case is for casts. We have $T <:_n \lceil \langle R \Leftarrow S \rangle s \rceil$ and therefore $T <:_n \lceil R \rceil$ and $T <:_n \lceil S \rceil$. Then by Proposition 3 (part 2) we have that $T <:_n \lceil R \& S \rceil$. By the induction hypothesis we have $T <:_n \langle\!\langle s \rangle\!\rangle$. Therefore $T <:_n \lceil \langle R \overset{R \& S}{\Longleftarrow} S \rangle \langle\!\langle s \rangle\!\rangle \rceil$. □

Proof of Theorem 2. First, we have $\& \lceil t \rceil <:_n \lceil t \rceil$ by Proposition 1. We have $\& \lceil t \rceil <:_n \lceil \langle\!\langle t \rangle\!\rangle \rceil$ by Lemma 7. We then proceed by induction on $\langle\!\langle t \rangle\!\rangle \longmapsto^* t'$.

1. In the base case we have $t' = \langle\!\langle t \rangle\!\rangle$. So $\& \lceil t \rceil <:_n \lceil t' \rceil$ and then by Proposition 3 (part 3) we can conclude that $size(T) \leq size(\& \lceil t \rceil)$ for any T in t'.
2. For the induction step, we have $\langle\!\langle t \rangle\!\rangle \longmapsto t_1$ and $t_1 \longmapsto^* t'$. By Lemma 6 we have $\& \lceil t \rceil <:_n t_1$. Then by the induction hypothesis we have $\& \lceil t \rceil <:_n t'$. We conclude that $size(T) \leq size(\& \lceil t \rceil)$ for any T in t' by applying Proposition 3 (part 1 and 3). □

7.3 Correctness of Threesomes

Proof of Proposition 5. The proof is by induction on the sum of the heights of S and T. □

Proof of Proposition 4. The proof is by induction on the sum of the heights of Q and P. □

Lemma 19. *Suppose $s_2 \approx \langle T \overset{\perp \, lG^{\epsilon}}{\Longleftarrow} S \rangle u$. Then $s_2 \longmapsto^* \mathbf{blame}\ l$.*

Proof of Lemma 8.

1. The proof is by case analysis on $t_2 \longmapsto r_2$.
2. The proof is by case analysis on $t_3 \longmapsto r_3$. In the case $F[\langle T \overset{\perp \, lG^{\epsilon}}{\Longleftarrow} S \rangle u] \longmapsto \mathbf{blame}\ l$, we apply Lemma 19 to show that $t_2 \longmapsto^* \mathbf{blame}\ l$.

□

Proof of Lemma 9. The proof is by induction on the term t, using rule (24) for relating casts. □

Proof of Lemma 19. The proof is by induction on the derivation of $s_2 \approx \langle T \overset{\perp \, lG^{p}}{\Longleftarrow} S \rangle v$. □

Proof of Lemma 10. We need to show that if $t_2 \approx t_3$, then $t_2 \downarrow$ if and only if $t_3 \downarrow$.

1. ($\Longrightarrow$) The proof is by induction on the reduction $t_2 \longmapsto^* f_2$.
2. ($\Longleftarrow$) The proof is by induction on the reduction $t_3 \longmapsto^* f_3$.

□

Proof of Theorem 4. We need to show that $\langle\!\langle t \rangle\!\rangle \downarrow$ iff $t \downarrow$. By Lemma 9 we have $t \approx \langle\!\langle t \rangle\!\rangle$. We conclude by Lemma 10. □

7.4 Isomorphism Between Threesomes and Coercions

Proof of Proposition 6.

1. To show that $\mathsf{nm}\ c$ implies $\nexists c'.\ c \longmapsto c'$, perform induction on the derivation of $\mathsf{nm}\ c$.
2. We need to show that $\nexists c'.\ c \longmapsto c'$ implies $\mathsf{nm}\ c$. Towards a contradiction, we assume $\neg \mathsf{nm}\ c$. We proceed by induction on c. Each of the base cases is vacuously true. In the case where $c = c_1 \to c_2$, we apply the induction hypothesis to get a reduction in c_1 or c_2 and therefore a contradiction. In the case where $c = c_2 \circ c_1$, there are many cases to consider, but in each case there is a reduction. □

Proof of Proposition 7. The proof is by induction on P. □

Proof of Lemma 11.

1. We prove $(\!|(\!|T \overset{P}{\Longleftarrow} S|\!)|\!)^{-1} = \langle T \overset{P}{\Longleftarrow} S \rangle$ by induction on P.
2. We prove $(\!|(\!|c|\!)^{-1}|\!) = c$ by induction on the derivation of c. □

Proof of Lemma 12. The proof is by induction on the reduction sequence, with case analysis on each reduction. □

Proof of Theorem 5.

1. The proof of $(\!|(\!|t_3|\!)|\!)^{-1} = t_3$ is by induction on the typing derivation for t_3 and the proof of $(\!|(\!|t_c|\!)^{-1}|\!) = t_c$ is by induction on the typing derivation of t_c.
2. We prove that $t_3 \longmapsto t_3'$ if and only if $(\!|t_3|\!) \longmapsto (\!|t_3'|\!)$.
   (a) We show that $t_3 \longmapsto t_3'$ implies $(\!|t_3|\!) \longmapsto (\!|t_3'|\!)$. The proof is by cases on reduction, using Corollary 2 for rule (23).
   (b) We show that $(\!|t_3|\!) \longmapsto (\!|t_3'|\!)$ implies $t_3 \longmapsto t_3'$. The proof is by cases on reduction, using Corollary 2 for rule (39).

□