Threesomes, With and Without Blame[⋆]

Jeremy G. Siek[1] and Philip Wadler[2]

[1] University of Colorado at Boulder
[2] University of Edinburgh

Abstract. The blame calculus of Wadler and Findler gives a high-level semantics to casts in higher-order languages. The coercion calculus of Henglein, on the other hand, provides an instruction set for casts whose normal forms ensure space efficiency. In this paper we address two questions: 1) can space efficiency be obtained in a high-level semantics? and 2) can we precisely characterize the relationship between the high and low-level semantics of casts? Towards answering both of these questions, we design a cast calculus that summarizes a sequence of casts as a threesome cast that contains a source type, a target type, and a third middle type that is the greatest lower bound of all the types in the sequence. We show that the threesome calculus is equivalent to the blame calculus and to one of the coercion-based, blame-tracking calculi of Siek, Garcia, and Taha. We also show that the threesome calculus is space efficient and obtain a tighter bound than that of Herman, Tomb, and Flanagan.

1 Introduction

The blame calculus of Wadler and Findler [2009] gives a high-level semantics for casts in higher-order languages and shows how to trace cast errors back to their source (i.e., *blame tracking*) using techniques from *higher-order contracts* [Findler and Felleisen, 2002]. These casts integrate dynamic typing into statically typed languages, but they also play an important role in the compilation of languages with *gradual typing* [Siek and Taha, 2006] and *hybrid typing* [Gronski et al., 2006].

Herman et al. [2007] observe that in straightforward implementations of the blame calculus, casts can consume unbounded space at run time. To solve this problem, they use the coercion calculus of Henglein [1994] to compress sequences of casts. The coercion calculus can be viewed as low-level instruction set for casts. Siek et al. [2009] show how to add blame tracking to the coercion calculus while maintaining space efficiency. We review the blame calculus in Section 2 and the coercion calculus in Section 3.

In this paper we show that space efficiency can also be achieved in a high-level calculus. The key idea is that a cast consisting of three types (threesomes) can express a sequences of normal casts (twosomes). We describe the threesome calculus without blame tracking in Section 4 and show that the threesome calculus is space efficient, obtaining a tighter bound on space than that of Herman et al. [2007]. We then show that the threesome calculus is equivalent (ignoring blame labels) to both the blame calculus and a coercion-based calculus. In Section 5 we add blame tracking to the threesome

[⋆] Workshop on Script to Program Evolution (STOP), Genova, 2009

base types B	$\supset \{\texttt{Bool}, \texttt{Int}\}$	labels $l, m, \bar{l}, \overline{m}$
types R, S, T, U	$::= B \mid S \to T \mid \texttt{Dyn}$	terms $s, t \quad ::= k \mid x \mid \lambda x{:}S.t \mid s\ t \mid \langle T \overset{l}{\Leftarrow} S\rangle s$

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

$$\frac{\Gamma \vdash s : S \qquad S \sim T}{\Gamma \vdash \langle T \overset{l}{\Leftarrow} S\rangle s : T}$$

Consistency $\boxed{S \sim T}$

$$B \sim B \qquad \texttt{Dyn} \sim T \qquad T \sim \texttt{Dyn} \qquad \frac{S \sim S' \qquad T \sim T'}{S \to T \sim S' \to T'}$$

Fig. 1. Static semantics of the blame calculus

calculus and extend the equivalence results to include blame tracking. As a corollary, we establish that the high-level blame calculus is equivalent to the low-level coercion-based calculus.

2 Twosomes

The syntax and type system of the blame calculus (minus subset types) is shown in Fig. 1. This calculus is the simply-typed lambda calculus extended with the type Dyn (the sum of all types) and casts of the form $\langle T \overset{l}{\Leftarrow} S\rangle s$. We give application higher precedence than casts. The semantics of a cast is to evaluate term s to a value v, check whether the run-time type of v matches T, and if so, return 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 dynamic semantics of the blame calculus is shown in Fig. 2. The reflexive, transitive closure of the reduction relation is written $\longrightarrow^*$.

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

$$\langle \texttt{Int} \overset{m}{\Leftarrow} \texttt{Dyn}\rangle\langle \texttt{Dyn} \overset{l}{\Leftarrow} \texttt{Int}\rangle 4 \longrightarrow^* 4$$
$$\langle \texttt{Bool} \overset{m}{\Leftarrow} \texttt{Dyn}\rangle\langle \texttt{Dyn} \overset{l}{\Leftarrow} \texttt{Int}\rangle 4 \longrightarrow^* \mathbf{blame}\ m$$

Higher-order Casts The semantics of casts is more subtle in the case of higher-order values such a functions. The complication is that one cannot immediately check whether a function is consistent with the target type of the cast. For example, when the following function of type $\texttt{Int} \to \texttt{Dyn}$ 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.

$$f \equiv (\lambda x : \texttt{Int}.\ \texttt{if}\ 0 < x\ \texttt{then}\ \langle \texttt{Dyn} \overset{l_2}{\Leftarrow} \texttt{Int}\rangle 2\ \texttt{else}\ \langle \texttt{Dyn} \overset{l_1}{\Leftarrow} \texttt{Bool}\rangle \texttt{true})$$
$$\ldots \langle \texttt{Int} \to \texttt{Int} \overset{l_3}{\Leftarrow} \texttt{Int} \to \texttt{Dyn}\rangle f \ldots$$

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, then the function violates the cast.

ground types	G, H	$::= B \mid \mathtt{Dyn} \to \mathtt{Dyn}$
values	v, w	$::= k \mid \lambda x{:}S.t \mid \langle \mathtt{Dyn} \overset{l}{\Leftarrow} G\rangle v \mid \langle S' \to T' \overset{l}{\Leftarrow} S \to T\rangle v$
results	r	$::= t \mid \mathbf{blame}\ l$
evaluation contexts	E	$::= \square \mid E\ t \mid v\ E \mid \langle T \overset{l}{\Leftarrow} S\rangle E$

New Reductions $\boxed{s \longrightarrow r}$

$$
\begin{aligned}
E[\langle B \overset{l}{\Leftarrow} B\rangle v] &\longrightarrow E[v]\\
E[\langle \mathtt{Dyn} \overset{l}{\Leftarrow} \mathtt{Dyn}\rangle v] &\longrightarrow E[v]\\
E[\langle G \overset{m}{\Leftarrow} \mathtt{Dyn}\rangle \langle \mathtt{Dyn} \overset{l}{\Leftarrow} G\rangle v] &\longrightarrow E[v]\\
E[\langle H \overset{m}{\Leftarrow} \mathtt{Dyn}\rangle \langle \mathtt{Dyn} \overset{l}{\Leftarrow} G\rangle v] &\longrightarrow \mathbf{blame}\ m \qquad \text{if } G \neq H\\
E[\langle \mathtt{Dyn} \overset{l}{\Leftarrow} S \to T\rangle v] &\longrightarrow E[\langle \mathtt{Dyn} \overset{l}{\Leftarrow} \mathtt{Dyn} \to \mathtt{Dyn}\rangle \langle \mathtt{Dyn} \to \mathtt{Dyn} \overset{l}{\Leftarrow} S \to T\rangle v],\\
&\qquad \text{if } S \to T \neq \mathtt{Dyn} \to \mathtt{Dyn}\\
E[\langle S \to T \overset{l}{\Leftarrow} \mathtt{Dyn}\rangle v] &\longrightarrow E[\langle S \to T \overset{l}{\Leftarrow} \mathtt{Dyn} \to \mathtt{Dyn}\rangle \langle \mathtt{Dyn} \to \mathtt{Dyn} \overset{l}{\Leftarrow} \mathtt{Dyn}\rangle v],\\
&\qquad \text{if } S \to T \neq \mathtt{Dyn} \to \mathtt{Dyn}\\
E[(\langle S' \to T' \overset{l}{\Leftarrow} S \to T\rangle u)\ v] &\longrightarrow E[\langle T' \overset{l}{\Leftarrow} T\rangle\ u(\langle S \overset{\bar{l}}{\Leftarrow} S'\rangle v)]
\end{aligned}
$$

Fig. 2. Dynamic semantics of the blame calculus.

The standard solution, adopted from work on higher-order contracts by Findler and Felleisen [2002], is to defer checking until the function is applied to an argument, at which point the argument is compared to the target parameter type, and upon function return, the return value is checked against the return type. This can be accomplished by using the cast as a wrapper and splitting the wrapper into two when the function is applied: $(\langle T_1 \to T_2 \overset{l}{\Leftarrow} S_1 \to S_2\rangle v_1)\ v_2 \longrightarrow \langle T_2 \overset{l}{\Leftarrow} S_2\rangle\ v_1(\langle S_1 \overset{\bar{l}}{\Leftarrow} T_1\rangle v_2)$.

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 [Findler and Felleisen, 2002, Gronski and Flanagan, 2007, Siek et al., 2009, Wadler and Findler, 2009].

Space Efficiency Herman et al. [2007] observe two circumstances where higher-order casts can lead to unbounded space consumption. First, some programs repeatedly apply casts to the same function, which can build up to an arbitrarily large number of casts. Second, a cast applied to the result of a function call can turn what would have been a tail recursive function into one that is no longer tail recursive, thereby causing the function to consume space proportional to the depth of the recursion. We refer the reader to Herman et al. [2007] for specific examples of these space efficiency problems. Herman et al. [2007] solve these problems by using the coercion calculus of Henglein [1994] to compactly represent sequences of casts, which we discuss next.

3 The Coercion Calculus

Henglein [1994] introduced a sub-language called the coercion calculus to express casts. Instead of casts of the form $\langle T \overset{l}{\Leftarrow} S\rangle s$, 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 $\langle T \overset{l}{\Leftarrow} S\rangle s$ are compiled into casts of the form $\langle c\rangle s$. The original coercion calculus did not include blame tracking, so here we use the L $\cup$ UD coercion-based calculus of Siek et al. [2009], shown in Fig. 3.

The coercion $G!$ injects a value into Dyn whereas the coercion $G?^l$ projects a value out of Dyn, blaming location l in the case of a type mismatch. 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. In addition to Henglein's coercions, we adopt the $\mathtt{Fail}^l_{T \Leftarrow G}$ coercion of Herman et al. [2007], which compactly represents coercions that are destined to fail but have not yet been applied to a value.

We use a different formulation of evaluation context than that of Herman et al. [2007]. (Siek et al. [2009] use the same formulation as 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.

Lemma 1 (Unique Decomposition). *For a well-typed closed t, either t is a value or there is a unique decomposition into a redex t' and a context E such that $t = E[t']$.*

4 Threesomes without blame

Our goal in the design of the threesome calculus is to achieve space efficiency while maintaining the high-level nature of the calculus, that is, expressing casts in terms of types. The key to space efficiency is to find a way to compress sequences of casts into a single cast while maintaining the same behavior. We will compress two successive casts into a threesome cast, and show how to compress two successive threesome casts into a single, equivalent threesome cast; we can thus compress any sequence of two-way casts into a single threesome cast by a sequence of compressions. In this section, for simplicity, we will ignore blame labels, and restore them in the next section.

The simplest idea would be to coerce the pair of casts $\langle U \Leftarrow T\rangle\langle T \Leftarrow S\rangle s$ into the single cast $\langle U \Leftarrow S\rangle s$, but this is too simple. For instance, the cast

$$\langle \mathtt{Dyn} \Leftarrow \mathtt{Bool}\rangle\langle \mathtt{Bool} \Leftarrow \mathtt{Dyn}\rangle s$$

is not equivalent to $\langle \mathtt{Dyn} \Leftarrow \mathtt{Dyn}\rangle s$ because the former requires s to be the dynamic embedding of a boolean, while the latter is just the identity. Therefore, we use threesome casts that are the equivalent of a pair of two-way casts, writing

$$\langle U \overset{T}{\Longleftarrow} S\rangle s$$

to stand for $\langle U \Leftarrow T\rangle\langle T \Leftarrow S\rangle s$. In particular, the sequence of casts above is written $\langle \mathtt{Dyn} \overset{\mathtt{Bool}}{\Longleftarrow} \mathtt{Dyn}\rangle s$.

However, we must extend the notion of type that we use for middle types to also include a bottom type. For instance, the cast

$$\langle \mathtt{Int} \Leftarrow \mathtt{Dyn}\rangle\langle \mathtt{Dyn} \Leftarrow \mathtt{Bool}\rangle s$$

will collapse to $\langle \mathtt{Int} \overset{\bot}{\Longleftarrow} \mathtt{Bool}\rangle s$. A threesome cast with middle $\bot$ reduces to $\mathbf{blame}$.

non-dynamic types	S^*, T^*	$::= B \mid S \to T$
coercions	c, d	$::= \iota_T \mid G! \mid G?^l \mid d \circ c \mid c \to d \mid \mathtt{Fail}^l_{T \Leftarrow S^*}$
terms	s, t	$::= k \mid x \mid \lambda x.t \mid s\ t \mid \langle c \rangle s$

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

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

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

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

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

Compilation of casts to coercions $\boxed{\langle\!\langle T \Leftarrow^l S \rangle\!\rangle = c}$

$$\langle\!\langle B \Leftarrow^l B \rangle\!\rangle = \iota_B \qquad \langle\!\langle \mathtt{Dyn} \Leftarrow^l \mathtt{Dyn} \rangle\!\rangle = \iota_{\mathtt{Dyn}} \qquad \langle\!\langle \mathtt{Dyn} \Leftarrow^l B \rangle\!\rangle = B! \qquad \langle\!\langle B \Leftarrow^l \mathtt{Dyn} \rangle\!\rangle = B?^l$$
$$\langle\!\langle \mathtt{Dyn} \Leftarrow^l S \to T \rangle\!\rangle = (\mathtt{Dyn} \to \mathtt{Dyn})! \circ (\langle\!\langle S \Leftarrow^l \mathtt{Dyn} \rangle\!\rangle \to \langle\!\langle \mathtt{Dyn} \Leftarrow^l T \rangle\!\rangle)$$
$$\langle\!\langle S \to T \Leftarrow^l \mathtt{Dyn} \rangle\!\rangle = (\langle\!\langle \mathtt{Dyn} \Leftarrow^l S \rangle\!\rangle \to \langle\!\langle T \Leftarrow^l \mathtt{Dyn} \rangle\!\rangle) \circ (\mathtt{Dyn} \to \mathtt{Dyn})?^l$$
$$\langle\!\langle S' \to T' \Leftarrow^l S \to T \rangle\!\rangle = \langle\!\langle S \Leftarrow^l S' \rangle\!\rangle \to \langle\!\langle T' \Leftarrow^l T \rangle\!\rangle$$

Compilation from the blame calculus to the coercion-based calculus $\boxed{\langle\!\langle t \rangle\!\rangle_c = t}$

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

Run-time Structures			
	normalized coercions	$\overline{c}$	
	coercion contexts	C	$::= C \circ c \mid d \circ C \mid C \to d \mid c \to C$
	uncoerced values	u	$::= k \mid \lambda x : S.t$
	values	v, w	$::= u \mid \langle \overline{c} \rangle u$
	cast-free context	F	$::= \square \mid E[\square\ t] \mid E[v\ \square]$
	evaluation contexts	E	$::= F \mid F[\langle c \rangle \square]$

Coercion equality (refl., symm., trans., and compatible closure of the following rule) $\boxed{c = d}$

$$c_1 \circ c_2 \circ c_3 = (c_1 \circ c_2) \circ c_3$$

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

$$\begin{aligned}
C[G?^l \circ G!] &\longrightarrow C[\iota_G] \\
C[H?^l \circ G!] &\longrightarrow C[\mathtt{Fail}^l_{H \Leftarrow G}] \qquad \text{if } G \neq H \\
C[(d_1 \to d_2) \circ (c_1 \to c_2)] &\longrightarrow C[(c_1 \circ d_1) \to (d_2 \circ c_2)] \\
C[\iota_T \circ c] &\longrightarrow C[c] \\
C[d \circ \iota_T] &\longrightarrow C[d] \\
C[d \circ \mathtt{Fail}^l_{S \Leftarrow S^*}] &\longrightarrow C[\mathtt{Fail}^l_{T \Leftarrow S^*}] \qquad \text{where } \vdash d : T \Leftarrow S \\
C[\mathtt{Fail}^l_{T \Leftarrow T^*} \circ (c \to d)] &\longrightarrow C[\mathtt{Fail}^l_{T \Leftarrow S^*}] \qquad \text{where } \vdash c \to d : T^* \Leftarrow S^*
\end{aligned}$$

New Reductions $\boxed{s \longrightarrow r}$

$$\begin{aligned}
E[\langle \iota_B \rangle u] &\longrightarrow E[u] \\
F[\langle \overline{d} \rangle \langle \overline{c} \rangle v] &\longrightarrow F[\langle \overline{c'} \rangle v] \qquad \text{if } \overline{d} \circ \overline{c} \longrightarrow^* \overline{c'} \\
F[\langle \mathtt{Fail}^l_{T \Leftarrow G} \rangle u] &\longrightarrow \mathbf{blame}\ l \\
E[(\langle \overline{c} \to \overline{d} \rangle u)\ w] &\longrightarrow E[\langle \overline{d} \rangle\ u (\langle \overline{c} \rangle w)]
\end{aligned}$$

Fig. 3. The L ∪ UD coercion-based calculus

It is worth noticing the following invariant: the middle type is smaller or equal to the source and target types with respect to naive subtyping. In naive subtyping (Fig. 4), Dyn is top and $\perp$ is bottom. Function types are naive subtypes if their corresponding parameters and return types are naive subtypes, both covariantly. The next example shows how this covariance agrees with the blame calculus reduction rules.

Suppose that the two casts are higher-order, casting between function types:

$$\langle U_1 \to U_2 \Leftarrow T_1 \to T_2\rangle\langle T_1 \to T_2 \Leftarrow S_1 \to S_2\rangle s$$

The checking of higher-order casts is delayed, but later on the cast could be applied to an argument t, causing the casts to be split as follows.

$$\langle U_2 \Leftarrow T_2\rangle\langle T_2 \Leftarrow S_2\rangle\ s(\langle S_1 \Leftarrow T_1\rangle\langle T_1 \Leftarrow U_1\rangle t)$$

To simulate this behavior with threesome casts we simply use the following reduction rule for splitting higher-order threesomes.

$$\langle U_1 \to U_2 \overset{T_1\to T_2}{\Longleftarrow} S_1 \to S_2\rangle s \longrightarrow \langle U_2 \overset{T_2}{\Longleftarrow} S_2\rangle\ s(\langle S_1 \overset{T_1}{\Leftarrow} U_1\rangle t)$$

Because $T_1 \to T_2 <:_n S_1 \to S_2$ we have $T_1 <:_n S_1$ and $T_2 <:_n S_2$, so the subtyping invariant continues to hold on the middle types thanks to the covariance on functions.

Next, we consider how to compress two successive threesome casts into a single equivalent threesome cast. The middle type of the new cast will be some function of the middle types of the two casts. Let us use $\&$ for this function.

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

First, we observe that the blame calculus checks the head symbol of a type to decide whether to signal an error. The following gnd function isolates the head symbol.

$$gnd(B) = B \qquad gnd(S \to T) = \texttt{Dyn} \to \texttt{Dyn}$$

So we should have $T\&T' = \perp$ if $gnd(T) \neq gnd(T')$. On the the other hand, suppose T and T' are the same base type B. Then the two threesomes represent

$$\langle R \Leftarrow B\rangle\langle B \Leftarrow U\rangle\langle U \Leftarrow B\rangle\langle B \Leftarrow S\rangle s$$

and the type system forces U to be either Dyn or B, so the middle two casts are the identity. Thus, the result should be $\langle R \overset{B}{\Longleftarrow} S\rangle s$ and we have $B\&B = B$.

Next, suppose that T is Dyn. Then both S and U must also be Dyn by the subtyping invariant. It is then easy to see then that we should have $T'\&\texttt{Dyn} = T'$.

$$\langle R \overset{T'}{\Longleftarrow} \texttt{Dyn}\rangle\langle \texttt{Dyn} \overset{\texttt{Dyn}}{\Longleftarrow} \texttt{Dyn}\rangle s \longrightarrow \langle R \overset{T'}{\Longleftarrow} \texttt{Dyn}\rangle s$$

Similar reasoning gives us $\texttt{Dyn}\&T = T$.

For function types, we can derive the definition of $\&$ from the observation that merging higher-order casts should be equivalent to splitting higher-order casts at function applications then merging the results. The following reduction shows splitting then merging.

$$\begin{aligned}
&(\langle R_1 \to R_2 \overset{T_1' \to T_2'}{\Longleftarrow} U_1 \to U_2\rangle\langle U_1 \to U_2 \overset{T_1 \to T_2}{\Longleftarrow} S_1 \to S_2\rangle s)\ t \\
&\longrightarrow^2 \langle R_2 \overset{T_2'}{\Longleftarrow} U_2\rangle\langle U_2 \overset{T_2}{\Longleftarrow} S_2\rangle\ s(\langle S_1 \overset{T_1}{\Longleftarrow} U_1\rangle\langle U_1 \overset{T_1'}{\Longleftarrow} R_1\rangle t) \\
&\longrightarrow \langle R_2 \overset{T_2' \& T_2}{\Longleftarrow} S_2\rangle\ s(\langle S_1 \overset{T_1 \& T_1'}{\Longleftarrow} R_1\rangle t)
\end{aligned}$$

So the definition of & for function types should be

$$(T_1' \to T_2')\&(T_1 \to T_2) = (T_1 \& T_1') \to (T_2' \& T_2)$$

We have now derived all of the equation for & except for $\perp$, which we leave for the reader. Interestingly enough, the & function computes the greatest lower bound with respect to naive subtyping.

Lemma 2 (& is the greatest lower bound).

1. *$S\&T <:_n S$ and $S\&T <:_n T$*
2. *$R <:_n S$ and $R <:_n T$ implies $R <:_n S\&T$*

Equivalence to the (twosome) blame calculus Towards proving the threesome calculus equivalent to the blame calculus (ignoring blame labels), we define a bisimulation relation $\approx$.

$$\frac{}{x \approx x} \qquad \frac{}{k \approx k} \qquad \frac{s_2 \approx s_3 \quad t_2 \approx t_3}{s_2\ t_2 \approx s_3\ t_3}$$

$$\frac{t_2 \approx t_3}{\lambda x : S.\ t_2 \approx \lambda x : S.\ t_3} \qquad \frac{s_2 \approx s_3}{s_2 \approx s_3} \qquad \frac{}{\mathbf{blame}\ l \approx \mathbf{blame}}$$

$$\frac{s_2 \approx s_3}{\langle T \overset{l}{\Leftarrow} S\rangle s_2 \approx \langle T \overset{T\&S}{\Longleftarrow} S\rangle s_3} \qquad \frac{t_2 \approx \langle T \overset{P}{\Longleftarrow} S\rangle s_3 \quad Q = U\&T}{\langle U \overset{l}{\Leftarrow} T\rangle t_2 \approx \langle U \overset{Q\&P}{\Longleftarrow} S\rangle s_3}$$

$$\frac{t_2 \approx ([\![T_1 \to T_2 \overset{P}{\Longleftarrow} S]\!] s_3)\ (\langle T_1 \overset{T_1\&U_1}{\Longleftarrow} U_1\rangle t_3) \quad Q = (U_1 \to U_2)\&(T_1 \to T_2)}{\langle U_2 \overset{l}{\Leftarrow} T_2\rangle t_2 \approx (\langle U_1 \to U_2 \overset{Q\&P}{\Longleftarrow} S\rangle s_3)\ t_3}$$

$$\text{where } [\![T \overset{P}{\Longleftarrow} S]\!] s = \begin{cases} s & \text{if } S = P = T \\ \langle T \overset{P}{\Longleftarrow} S\rangle s & \text{otherwise} \end{cases}$$

Lemma 3 (Bisimulation between the blame calculus and threesome calculus).
If $t_2 \approx t_3$ and both t_2 and t_3 are well typed, then

1. *if $t_2 \longrightarrow r_2$, then $t_3 \longrightarrow^* r_3$ and $r_2 \approx r_3$ for some r_3.*
2. *if $t_3 \longrightarrow r_3$, then $t_2 \longrightarrow^* r_2$ and $r_2 \approx r_3$ for some r_2.*

Lemma 4. $t \approx \langle\!\langle t \rangle\!\rangle_3$

Theorem 1 (Equivalence of the blame calculus and threesome calculus).

1. *$t_2 \longrightarrow^* v_2$ implies $\langle\!\langle t_2 \rangle\!\rangle_3 \longrightarrow_3^* v_3$ and $v_2 \approx v_3$ for some v_3.*
2. *$\langle\!\langle t_2 \rangle\!\rangle_3 \longrightarrow_3^* v_3$ implies $t_2 \longrightarrow^* v_2$ and $v_2 \approx v_3$ for some v_2.*
3. *$t_2 \longrightarrow^* \mathbf{blame}\ l$ for some l iff $\langle\!\langle t_2 \rangle\!\rangle_3 \longrightarrow_3^* \mathbf{blame}$.*

New Syntax

middle types	P, Q	$::= B \mid P \to Q \mid \texttt{Dyn} \mid \bot$
terms	s, t	$::= \dots \mid \langle T \overset{P}{\Longleftarrow} S \rangle s$
values	v, w	$::= u \mid \langle \texttt{Dyn} \overset{P}{\Longleftarrow} S \rangle u \mid \langle S' \to T' \overset{P \to Q}{\Longleftarrow} S \to T \rangle u$
cast-free contexts	F	$::= \square \mid E[\square\ t] \mid E[v\ \square]$
evaluation contexts	E	$::= F \mid F[\langle T \overset{P}{\Longleftarrow} S \rangle \square]$

Naive subtyping $\boxed{P <:_n Q}$

$$B <:_n B \qquad P <:_n \texttt{Dyn} \qquad \bot <:_n Q \qquad \frac{P <:_n P' \quad Q <:_n Q'}{P \to Q <:_n P' \to Q'}$$

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

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

Algorithm to compute the greatest lower bound $\boxed{P \& Q}$

$$\begin{aligned} B \& B &= B \\ (P \to Q) \& (P' \to Q') &= (P' \& P) \to (Q \& Q') \\ \texttt{Dyn} \& P &= P \\ P \& \texttt{Dyn} &= P \end{aligned} \qquad \begin{aligned} \bot \& P &= \bot \\ P \& \bot &= \bot \\ P \& Q &= \bot \qquad \text{if } \mathrm{gnd}(P) \neq \mathrm{gnd}(Q) \end{aligned}$$

Compilation from the blame calculus $\boxed{\langle\!\langle t \rangle\!\rangle_3 = t}$

$$\langle\!\langle x \rangle\!\rangle_3 = x \qquad \langle\!\langle k \rangle\!\rangle_3 = k \qquad \langle\!\langle \lambda x : S.\ t \rangle\!\rangle_3 = \lambda x : S.\ \langle\!\langle t \rangle\!\rangle_3$$

$$\langle\!\langle t\ s \rangle\!\rangle_3 = \langle\!\langle t \rangle\!\rangle_3\ \langle\!\langle s \rangle\!\rangle_3 \qquad \langle\!\langle \langle T \overset{l}{\Leftarrow} S \rangle s \rangle\!\rangle_3 = \langle T \overset{T \& S}{\Longleftarrow} S \rangle \langle\!\langle s \rangle\!\rangle_3$$

New Reductions $\boxed{s \longrightarrow r}$

$$\begin{aligned} E[\langle B \overset{B}{\Longleftarrow} B \rangle u] &\longrightarrow E[u] \\ E[\langle \texttt{Dyn} \overset{\texttt{Dyn}}{\Longleftarrow} \texttt{Dyn} \rangle u] &\longrightarrow E[u] \\ F[\langle T \overset{Q}{\Longleftarrow} S' \rangle \langle S' \overset{P}{\Longleftarrow} S \rangle s] &\longrightarrow F[\langle T \overset{Q \& P}{\Longleftarrow} S \rangle s] \\ F[\langle T \overset{\bot}{\Longleftarrow} S \rangle u] &\longrightarrow \mathbf{blame} \\ E[(\langle S' \to T' \overset{P \to Q}{\Longleftarrow} S \to T \rangle u)\ w] &\longrightarrow E[\langle T' \overset{Q}{\Longleftarrow} T \rangle\ u(\langle S \overset{P}{\Longleftarrow} S' \rangle w)] \end{aligned}$$

Fig. 4. The threesome calculus without blame.

Equivalence to the coercion-based calculus The correspondence between the threesome cast calculus and the coercion-based calculus is very tight. We relate coercions to threesome casts with the mid function, which computes the middle type from a coercion.

$$mid(\iota_T) = T \qquad mid(G!) = G \qquad mid(G?^l) = G \qquad mid(\mathtt{Fail}^l_{H \Leftarrow G}) = \bot$$
$$mid(c \to d) = mid(c) \to mid(d) \qquad mid(d \circ c) = mid(d) \& mid(c)$$

The following is the bisimulation relation between the two.

$$\frac{}{x \approx x} \qquad \frac{}{k \approx k} \qquad \frac{s_1 \approx s_2 \quad t_1 \approx t_2}{s_1\ t_1 \approx s_2\ t_2} \qquad \frac{}{\mathbf{blame}\ l \approx \mathbf{blame}}$$
$$\frac{u_1 \approx u_2}{\lambda x : S.\ u_1 \approx \lambda x : S.\ u_2} \qquad \frac{s_1 \approx s_2 \quad R = mid(c)}{\langle c \rangle s_1 \approx \langle T \overset{R}{\Longleftarrow} S \rangle s_2}$$

Lemma 5 (Strong bisimulation between the coercion and threesome calculus). *If $t_c \approx t_3$ and both t_c and t_3 are well typed, then*

1. *if $t_c \longrightarrow r_c$, then $t_3 \longrightarrow r_3$ and $r_c \approx r_3$ for some r_3.*
2. *if $t_3 \longrightarrow r_3$, then $t_c \longrightarrow r_c$ and $r_c \approx r_3$ for some r_c.*

Theorem 2 (Equivalence of the coercion calculus and threesome calculus).

1. *$\langle\!\langle t_2 \rangle\!\rangle_c \longrightarrow^*_c v_c$ implies $\langle\!\langle t_2 \rangle\!\rangle_3 \longrightarrow^*_3 v_3$ and $v_c \approx v_3$ for some v_3.*
2. *$\langle\!\langle t_2 \rangle\!\rangle_3 \longrightarrow^*_3 v_3$ implies $\langle\!\langle t_2 \rangle\!\rangle_c \longrightarrow^*_c v_c$ and $v_c \approx v_3$ for some v_c.*
3. *$\langle\!\langle t_2 \rangle\!\rangle_c \longrightarrow^*_c$ $\mathbf{blame}\ l$ for some l iff $\langle\!\langle t_2 \rangle\!\rangle_3 \longrightarrow^*_3$ $\mathbf{blame}$.*

Space Efficiency The main task in putting bounds on the size of casts during execution is to put a bound on the result of composing 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 coercions are trees with limited branching, it follows that the overall size of the composed coercion is bounded by roughly 2^h where h is the height.

Here 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 middle types, and therefore the resulting size of the new middle type is not much bigger. Recall the reduction rule for composing casts:

$$F[\langle T \overset{R}{\Longleftarrow} S' \rangle \langle S' \overset{U}{\Longleftarrow} S \rangle u] \longrightarrow F[\langle T \overset{R\&U}{\Longleftarrow} S \rangle u]$$

A straw man for the bound is the size of the greatest lower bound of all the types that occur in the program. The problem with this straw man is that the greatest lower bound of two types can sometimes be smaller, thereby not providing an upper bound on size. For example, $\mathtt{Int} \& (\mathtt{Dyn} \to \mathtt{Dyn}) = \bot$. Instead we need to take the maximum of the structure of the two types. This can be accomplished by mapping types to their *shadow*, written $\lceil T \rceil$, and then computing the greatest lower bound.

$$\lceil B \rceil = \mathtt{Dyn} \qquad \lceil S \to T \rceil = \lceil S \rceil \to \lceil T \rceil \qquad \lceil \mathtt{Dyn} \rceil = \mathtt{Dyn} \qquad \lceil \bot \rceil = \mathtt{Dyn}$$

We then have the properties that 1) the size of the shadow is the same as the size of the type, 2) a lower bound of the shadow of two types is also a lower bound of the shadow of their greatest lower bound, and 3) the size of a shadow is bounded above by any lower type with respect to naive subtyping.

Lemma 6.

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 $R <:_n \lceil S \rceil$ and $\bot \notin R$ then $size(\lceil S \rceil) \leq size(R)$.*

Lemma 7 (Preservation of lower bounds). *If T is a lower bound (with respect to naive subtyping) of the shadows of every type that occurs in s, and $s \longrightarrow s'$, then T is also a lower bound of the shadows of every type that occurs in s'.*

Theorem 3. *The size of any type that appears in the reduction sequence of a program is bounded by the size of the greatest lower bound of the shadows of all the types in the program.*

We conjecture that the size of the greatest lower bound of the shadows of all the types in the program is typically much less than 2^h, where h is the maximum height of any type in the program.

5 Threesomes with blame

We add blame tracking to the threesome cast calculus by augmenting the middle type P in $\langle T \overset{P}{\Longleftarrow} S \rangle s$ with blame labels. Fig. 5 shows the syntax of the threesome cast calculus, and most importantly, the syntax of labeled types. The erasure of a labeled type P to an unlabeled type is written $|P|$. Fig. 5 also shows the compilation of the blame calculus to the threesome calculus.

With the addition of blame labels, we must replace the use of greatest lower bound in the semantics with a composition operator that takes into account the blame labels. Also, with the addition of blame labels, the order of cast failure becomes observable so composition is no longer symmetric. Before giving the definition of the composition operator, we establish some auxiliary notation.

$$gnd(B^p) = B^p \qquad gnd(P \to^p Q) = (\mathtt{Dyn} \to \mathtt{Dyn})^p \qquad P^{G^p} \text{ iff } gnd(P) = G^p$$

We give the definitions for the composition of labeled types and the dynamic semantics of the threesome cast calculus in Fig. 6.

Our treatment of the labeled bottom type $\bot^{lG^p}$ 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.

$$\langle \mathtt{Int} \overset{l}{\Leftarrow} \mathtt{Dyn} \rangle \langle \mathtt{Dyn} \overset{m}{\Leftarrow} \mathtt{Bool} \rangle \langle \mathtt{Bool} \overset{n}{\Leftarrow} \mathtt{Dyn} \rangle \langle \mathtt{Dyn} \overset{o}{\Leftarrow} \mathtt{Int} \rangle 1 \longrightarrow \mathbf{blame}\, n$$

$$\langle \mathtt{Int} \overset{l}{\Leftarrow} \mathtt{Dyn} \rangle \langle \mathtt{Dyn} \overset{m}{\Leftarrow} \mathtt{Bool} \rangle \langle \mathtt{Bool} \overset{n}{\Leftarrow} \mathtt{Dyn} \rangle \langle \mathtt{Dyn} \overset{o}{\Leftarrow} \mathtt{Bool} \rangle \mathtt{true} \longrightarrow \mathbf{blame}\, l$$

New Syntax

$$\begin{array}{lll} \text{optional label} & p, q & ::= l \mid \epsilon \\ \text{labeled types} & P, Q & ::= B^p \mid P \to^p Q \mid \texttt{Dyn} \mid \bot^{lG^p} \\ \text{terms} & s, t & ::= k \mid x \mid \lambda x.t \mid s\ t \mid \langle T \xLeftarrow{P} S \rangle s \end{array}$$

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

$$\frac{\Gamma \vdash s : S \quad T <:_n |P| \quad S <:_n |P|}{\Gamma \vdash \langle T \xLeftarrow{P} S \rangle s : T}$$

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

$$\{\!\!\{ B \overset{l}{\Leftarrow} B \}\!\!\} = B^\epsilon \qquad \{\!\!\{ S' \to T' \overset{l}{\Leftarrow} S \to T \}\!\!\} = \{\!\!\{ S \overset{\bar{l}}{\Leftarrow} S' \}\!\!\} \to^\epsilon \{\!\!\{ T' \overset{l}{\Leftarrow} T \}\!\!\}$$

$$\{\!\!\{ \texttt{Dyn} \overset{l}{\Leftarrow} \texttt{Dyn} \}\!\!\} = \texttt{Dyn} \qquad \{\!\!\{ B \overset{l}{\Leftarrow} \texttt{Dyn} \}\!\!\} = B^l \qquad \{\!\!\{ \texttt{Dyn} \overset{l}{\Leftarrow} B \}\!\!\} = B^\epsilon$$

$$\{\!\!\{ S \to T \overset{l}{\Leftarrow} \texttt{Dyn} \}\!\!\} = \{\!\!\{ \texttt{Dyn} \overset{\bar{l}}{\Leftarrow} S \}\!\!\} \to^l \{\!\!\{ T \overset{l}{\Leftarrow} \texttt{Dyn} \}\!\!\}$$

$$\{\!\!\{ \texttt{Dyn} \overset{l}{\Leftarrow} S \to T \}\!\!\} = \{\!\!\{ S \overset{\bar{l}}{\Leftarrow} \texttt{Dyn} \}\!\!\} \to^l \{\!\!\{ \texttt{Dyn} \overset{l}{\Leftarrow} T \}\!\!\}$$

Compile blame terms to threesome terms $\boxed{\langle\!\langle t \rangle\!\rangle_3 = t}$

$$\langle\!\langle x \rangle\!\rangle_3 = x \qquad \langle\!\langle k \rangle\!\rangle_3 = k \qquad \langle\!\langle \lambda x : S.\ t \rangle\!\rangle_3 = \lambda x : S.\ \langle\!\langle t \rangle\!\rangle_3$$

$$\langle\!\langle t\ s \rangle\!\rangle_3 = \langle\!\langle t \rangle\!\rangle_3\ \langle\!\langle s \rangle\!\rangle_3 \qquad \langle\!\langle \langle T \overset{l}{\Leftarrow} S \rangle s \rangle\!\rangle_3 = \langle T \xLeftarrow{\{\!\!\{ T \overset{l}{\Leftarrow} S \}\!\!\}} S \rangle \langle\!\langle s \rangle\!\rangle_3$$

Fig. 5. Static semantics of the threesome calculus and compilation from blame calculus.

Syntax

$$\begin{array}{lll} \text{values} & v, w & ::= u \mid \langle \texttt{Dyn} \xLeftarrow{P} S \rangle u \mid \langle S' \to T' \xLeftarrow{P \to^p Q} S \to T \rangle u \\ \text{cast-free context} & F & ::= \Box \mid E[\Box\ t] \mid E[v\ \Box] \\ \text{evaluation contexts} & E & ::= F \mid F[\langle T \xLeftarrow{P} S \rangle \Box] \end{array}$$

Composition $\boxed{Q \circ P}$

$$\begin{array}{rl} B^q \circ B^p &= B^p \\ P \circ \texttt{Dyn} &= P \\ \texttt{Dyn} \circ P &= P \end{array} \qquad \begin{array}{rl} Q^{H^m} \circ P^{G^p} &= \bot^{mG^p} \text{ if } G \neq H \\ Q \circ \bot^{mG^p} &= \bot^{mG^p} \\ \bot^{mG^q} \circ P^{G^p} &= \bot^{mG^p} \\ \bot^{mH^l} \circ P^{G^p} &= \bot^{lG^p} \end{array}$$

$$(P' \to^q Q') \circ (P \to^p Q) = (P \circ P') \to^p (Q' \circ Q)$$

New Reductions $\boxed{s \longrightarrow r}$

$$\begin{array}{rcl} E[\langle B \xLeftarrow{B^\epsilon} B \rangle u] &\longrightarrow& E[u] \\ E[\langle \texttt{Dyn} \xLeftarrow{\texttt{Dyn}} \texttt{Dyn} \rangle u] &\longrightarrow& E[u] \\ F[\langle U \xLeftarrow{Q} T \rangle \langle T \xLeftarrow{P} S \rangle s] &\longrightarrow& F[\langle U \xLeftarrow{Q \circ P} S \rangle s] \\ F[\langle T \xLeftarrow{\bot^{lG^\epsilon}} S \rangle u] &\longrightarrow& \mathbf{blame}\ l \\ E[(\langle S' \to T' \xLeftarrow{P \to^p Q} S \to T \rangle u)\ w] &\longrightarrow& E[\langle T' \xLeftarrow{Q} T \rangle\ u(\langle S \xLeftarrow{P} S' \rangle w)] \end{array}$$

Fig. 6. Dynamic semantics of the threesome cast calculus.

In the threesome calculus the 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 have the calculus 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 $\bot^{l\mathtt{Bool}^n}$.

Equivalence to the (twosome) blame calculus The bisimulation relation is similar to the one for the threesome calculus without blame. The main difference is that uses of the greatest lower bound are replaced with the compilation and composition functions.

$$\frac{}{x \approx x} \qquad \frac{s_2 \approx s_3 \quad t_2 \approx t_3}{s_2\ t_2 \approx s_3\ t_3} \qquad \frac{s_2 \approx s_3}{\lambda x : S.\ s_2 \approx \lambda x : S.\ s_3} \qquad \frac{s_2 \approx s_3}{s_2 \approx s_3}$$

$$\frac{s_2 \approx s_3 \quad P = \{\!\{T \overset{l}{\Leftarrow} S\}\!\}}{\langle T \overset{l}{\Leftarrow} S\rangle s_2 \approx \langle T \overset{P}{\Longleftarrow} S\rangle s_3} \qquad \frac{s_2 \approx \langle T \overset{P}{\Longleftarrow} S\rangle s_3 \quad Q = \{\!\{U \overset{l}{\Leftarrow} T\}\!\}}{\langle U \overset{l}{\Leftarrow} T\rangle s_2 \approx \langle U \overset{Q\circ P}{\Longleftarrow} S\rangle s_3} \qquad \mathbf{blame}\ l \approx \mathbf{blame}\ l$$

$$\frac{t_2 \approx ([\![T_1 \to T_2 \overset{P}{\Longleftarrow} S]\!] s_3)\ (\langle T_1 \overset{\{\!\{T_1 \overset{l}{\Leftarrow} U_1\}\!\}}{\Longleftarrow} U_1\rangle t_3) \quad Q = \{\!\{U_1 \to U_2 \overset{l}{\Leftarrow} T_1 \to T_2\}\!\}}{\langle U_2 \overset{l}{\Leftarrow} T_2\rangle t_2 \approx (\langle U \overset{Q\circ P}{\Longleftarrow} S\rangle s_3)\ t_3}$$

$$\text{where } [\![T \overset{P}{\Longleftarrow} S]\!] s = \begin{cases} s & \text{if } S = |P| = T \\ \langle T \overset{P}{\Longleftarrow} S\rangle s & \text{otherwise} \end{cases}$$

Lemma 8 (Bisimulation between the blame calculus and the threesome calculus). *If $t_2 \approx t_3$ and both t_2 and t_3 are well typed, then*

1. *if $t_2 \longrightarrow r_2$, then $t_3 \longrightarrow_3^* r_3$ and $r_2 \approx r_3$ for some r_3.*
2. *if $t_3 \longrightarrow r_3$, then $t_2 \longrightarrow^* r_2$ and $r_2 \approx r_3$ for some r_2.*

Theorem 4 (Equivalence of blame calculus and the threesome calculus).

1. *$t_2 \longrightarrow^* v_2$ implies $\langle\!\langle t_2\rangle\!\rangle_3 \longrightarrow_3^* v_3$ and $v_2 \approx v_3$ for some v_3.*
2. *$\langle\!\langle t_2\rangle\!\rangle_3 \longrightarrow_3^* v_3$ implies $t_2 \longrightarrow^* v_2$ and $v_2 \approx v_3$ for some v_2.*
3. *$t_2 \longrightarrow^* \mathbf{blame}\ l$ iff $\langle\!\langle t_2\rangle\!\rangle_3 \longrightarrow_3^* \mathbf{blame}\ l$.*

Equivalence to the coercion calculus The bisimulation between the coercion calculus and the threesome calculus is similar to the bisimulation without blame, just the mid function must be updated to account for blame labels. The bisimulation leads to the equivalence of the threesome calculus and the coercion-based calculus.

Theorem 5 (Equivalence of the coercion calculus and threesome calculus).

1. *$\langle\!\langle t_2\rangle\!\rangle_c \longrightarrow_c^* v_c$ implies $\langle\!\langle t_2\rangle\!\rangle_3 \longrightarrow_3^* v_3$ and $v_c \approx v_3$ for some v_3.*
2. *$\langle\!\langle t_2\rangle\!\rangle_3 \longrightarrow_3^* v_3$ implies $\langle\!\langle t_2\rangle\!\rangle_c \longrightarrow_c^* v_c$ and $v_c \approx v_3$ for some v_c.*
3. *$\langle\!\langle t_2\rangle\!\rangle_c \longrightarrow_c^* \mathbf{blame}\ l$ iff $\langle\!\langle t_2\rangle\!\rangle_3 \longrightarrow_3^* \mathbf{blame}\ l$.*

Using the transitivity of bisimulation, it is then straightforward to connect, via the threesome calculus, the blame calculus with the coercion-based calculus .

Corollary 1 (Equivalence of the blame and coercion calculus).

1. *$t_2 \longrightarrow^* v_2$ implies $\langle\!\langle t_2\rangle\!\rangle_c \longrightarrow_c^* v_c$ and $v_2 \approx v_c$ for some v_c.*
2. *$\langle\!\langle t_2\rangle\!\rangle_c \longrightarrow_c^* v_c$ implies $t_2 \longrightarrow^* v_2$ and $v_2 \approx v_c$ for some v.*
3. *$t_2 \longrightarrow^* \mathbf{blame}\ l$ iff $\langle\!\langle t_2\rangle\!\rangle_c \longrightarrow_c^* \mathbf{blame}\ l$.*

6 Conclusion

In this paper we presented the first space-efficient, high-level semantics for casts: the threesome cast calculus. We show that the threesome cast calculus is equivalent to the blame calculus and the $\mathrm{L} \cup \mathrm{UD}$ coercion-based calculus, thereby providing a concrete connection between prior high-level and low-level semantics for casts.

Bibliography

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

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.

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.

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

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