Blame, coercions, and threesomes, precisely

Jeremy Siek  
Indiana University  
jsiek@indiana.edu

Peter Thiemann  
Universität Freiburg  
thiemann@informatik.uni-freiburg.de

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

Abstract

We systematically present four calculi for gradual typing: the blame calculus of Wadler and Findler (2009); a novel calculus that pinpoints blame precisely; the coercion calculus of Henglein (1994); and the threesome calculus of Siek and Wadler (2010). Threesomes are given a syntax that directly exposes their origin as coercions in normal form, a more transparent presentation than that found in Siek and Wadler (2010) or Garcia (2013).

1. Introduction

C#, Dart, Pyret, Racket, TypeScript, VB: many recent languages integrate dynamic and static types. Blame, coercions, and threesomes provide a foundation for such integration. The blame calculus models the interaction of dynamic and static types; it compiles into a coercion calculus, which models how to implement casts but is not space-efficient; it in turn compiles into the threesome calculus, which models a space-efficient implementation. Here we strengthen this foundation in four ways.

Precise blame. The blame calculus reconciles static and dynamic types via casts. Previous work allocates blame to the whole of a cast. In practice, systems such as PLT Racket pinpoint the location of blame more precisely. For instance, blame may be allocated to the domain of a function contract, indicating that the problem did not lie with the range. No published work models precise allocation of blame. Here we present a novel variant of the blame calculus that models precise allocation of blame.

Blame safety. The blame calculus not only satisfies the usual type safety property, but also blame safety, which shows that if a cast fails, blame allocates to the dynamically typed side—“Well-typed programs can’t be blamed”. Surprisingly, no previous work considers whether the translations preserve blame safety. Here we show that blame safety is preserved by the translations between the calculi, and, as a pleasant consequence, that a somewhat subtle definition of blame safety for the blame calculus is justified by a straightforward definition of blame safety for coercions.

Transparent threesomes. The threesome calculus implements blame in a space-efficient way. Threesomes can be explained in relation to coercions, but the explanation is far from transparent. For instance, Wadler, on returning to the threesome paper after two years, struggled for hours to understand his own notation, \( \lambda^{\text{1to3}} \).

Here we present a novel version of the threesome calculus that renders the connection to coercions transparent.

Connect the dots. We systematically present four calculi that integrate dynamic and static types, and establish relationships between them. The four calculi are: \( \lambda B \), the blame calculus, based on Wadler and Findler (2009); \( \lambda P \), the precise blame calculus, a novel system that pinpoints blame precisely; \( \lambda C \), the coercion calculus, based on Henglein (1994); and \( \lambda T \), the threesome calculus, based on Siek and Wadler (2010). For each calculus, we provide a type system, reduction rules, and a definition of blame safety, and we prove type safety and blame safety. We provide translations between the four calculi, show that each translation preserves types and blame safety, and show that each translation is a simulation or bisimulation, or embeds in one. Our results are summarised in Figure 1: the three single-headed arrows stand for simulations, and two double-headed arrows stand for bisimulations.

The translation from \( \lambda B \) to \( \lambda C \) is well known, but the other translations are new. Our previous work related coercions in normal form to threesomes, but our proof that (unnormalised) coercions and threesomes are in bisimulation is novel. All proofs are straightforward and do not require new techniques.

Background. Findler and Felleisen (2002) introduced the notion of higher-order contracts and blame to enforce invariants in a dynamically-typed language at run time. Siek and Taha (2006) introduced gradual types as a way to integrate static and dynamic typing via casts. Wadler and Findler (2009) introduced the blame calculus and blame safety. Henglein (1994) introduced a coercion calculus to integrate statically and dynamically typed languages. Herman et al. (2010) exploited the coercion calculus to provide a version of casts and gradual typing that is space-efficient. Siek and Wadler (2010) introduced threesomes, a variant of the casts used in the blame calculus, to achieve space efficiency in a way that is amenable to implementation, and connected threesomes with the coercion calculus. Garcia (2013) observes that coercions are easier to understand while threesomes are easier to implement, and attempts to alleviate the tension by showing how to derive three-
somes from coercions. However, Garcia retains problematic notation introduced by Siek and Wadler (such as $\perp$), as well as introducing more notation (such as $\perp$). Our presentation builds on the previous work, exposing the connection to coercions in a way that we hope you will find easier to follow.

**Summary.** The paper is organised as follows. Section 2 provides an overview. Sections 3–6 describe the four calculi; each section presents type safety and blame safety for its calculus, and translations to and from calculi of the preceding sections. Section 7 surveys related work and Section 8 concludes.

## 2. Overview

This section demonstrates key features of the four calculi through a series of examples.

### 2.1 Blame calculus, $\lambda B$

The blame calculus supports integration of dynamically and statically typed code. For example, let $f : \text{Int}\to\text{Int}$ be the statically typed increment function, $\langle x : \text{Int} , x + 1 \rangle$. Here we use dynamically-typed code in a statically-typed context:

$$\langle [\lambda g. g \ 3] : \star \Rightarrow (\text{Int}\to\text{Int})\to\text{Int} \rangle \to \underbar{\ell} 4$$

Here $\star$ embeds a dynamically-typed term in the blame calculus, $\Rightarrow$ is a cast from a source type to a target type, while $\Rightarrow_{\ell}$ is a reduction relation from one term to another. The cast is labelled with a blame path, which (for now) is either of the form $\ell$, a blame label, or $\ähr$, the complement of a blame label. Our notation is chosen for clarity, not compactness: a practical language might infer the source or target type of a cast, or even an entire cast.

Blame paths are used to report the location of a cast failure. Here we are code with an error introduced:

$$\langle [\lambda g. \text{true}] : \star \Rightarrow (\text{Int}\to\text{Int})\to\text{Int} \rangle \to \underbar{\ell} \text{ blame } \ell$$

It returns blame $\ell$, indicating that the cast labelled $\ell$ failed, and, further, that it was the term contained in the cast that failed—we call this positive blame.

We may also use statically-typed code in a dynamically-typed context:

$$\langle \text{let } f' = (f : \text{Int}\to\text{Int} \Rightarrow *) \in [f' \ 3] \rangle \to \underbar{\ell} 4$$

Here is our code with an error introduced:

$$\langle \text{let } f' = (f : \text{Int}\to\text{Int} \Rightarrow *) \in [f' \text{ true}] \rangle \to \underbar{\ell} \text{ blame } \ähr$$

It returns blame $\ähr$ (note the complement!). indicating that the cast labelled $\ähr$ failed, and, further, that it was the context containing the cast that failed—we call this negative blame.

Blame safety lets us characterise under what circumstances a cast may produce positive or negative blame. In particular, we show that a cast from dynamic type only allocates positive blame; and that a cast to dynamic type only allocates negative blame. Hence, if a cast fails, blame allocates to the dynamically typed side—“Well-typed programs can’t blame”.

Details of the blame calculus $\lambda B$ appear in Section 3.

### 2.2 Precise blame calculus, $\lambda P$

When a cast from a higher-order type fails, it can be for several reasons. Here is our code with three different errors:

$$\langle [\text{true}] : * \Rightarrow (\text{Int}\to\text{Int})\to\text{Int} \rangle \to \underbar{\ell} \text{ blame } \ell$$

$$\langle [\lambda p. \text{true}] : * \Rightarrow (\text{Int}\to\text{Int})\to\text{Int} \rangle \to \underbar{\ell} \text{ blame } \ell$$

$$\langle [\lambda g. \text{true}] : * \Rightarrow (\text{Int}\to\text{Int})\to\text{Int} \rangle \to \underbar{\ell} \text{ blame } \ell$$

The first fails because the term cannot be cast to a function. The second fails because the term can be cast to a function but the function returns a value of the wrong type. The third fails because the term can be cast to a function but that function passes its argument a value of the wrong type.

The precise blame calculus augments our notion of blame path to distinguish which part of the cast failed:

$$\langle [\text{true}] : * \Rightarrow (\text{Int}\to\text{Int})\to\text{Int} \rangle \rightarrow p \text{ blame } \ell$$

$$\langle [\lambda p. \text{true}] : * \Rightarrow (\text{Int}\to\text{Int})\to\text{Int} \rangle \rightarrow p \text{ blame } \ell$$

$$\langle [\lambda g. \text{true}] : * \Rightarrow (\text{Int}\to\text{Int})\to\text{Int} \rangle \rightarrow p \text{ blame } \ell$$

The first blames $\ell$, indicating that the top-level cast failed. The second blames $p/\text{prj}/\text{rng}$, indicating that the dynamic value projected to a function, but a value returned by the function did not match the declared range. The third blames $p/\text{prj}/\text{dom}/\text{inj}/\text{dom}$, indicating that the dynamic value projected to a function, the function was passed an argument matching its declared domain, that argument injected into the dynamic type, but the value passed to the argument did not match its declared domain.

Again, we can formulate a safety result for the precise blame calculus. We can also relate the two blame calculi, showing that a complemented blame label in the original calculus corresponds to a blame path with an odd number of dom tags in the precise calculus.

Details of the precise blame calculus $\lambda P$ appear in Section 4.

### 2.3 Coercion calculus, $\lambda C$

The blame and precise blame calculi compile into a coercion calculus based on Henglein (1994). Unlike the coercion calculus of Herman et al. (2010), the one we study is not space efficient; that is accomplished by the threesome calculus.

For example, the cast

$$\text{let } f' = (f : \text{Int}\to\text{Int} \Rightarrow *) \in [f' \ 3] \rangle \to \underbar{\ell} 4$$

compiles to the coercion

$$f?\ (\Rightarrow *) \Rightarrow (\text{Int}!) \Rightarrow (f?\ (\Rightarrow *))$$

while the cast

$$\text{let } f' = (f : \text{Int}\to\text{Int} \Rightarrow *) \in [f' \text{ true}] \rangle \to \underbar{\ell} \text{ blame } \ähr$$

compiles to the coercion

$$f?\ (\Rightarrow *) \Rightarrow (\text{Int}!) \Rightarrow (f?\ (\Rightarrow *))$$

Here coercions $\text{Int}!$ and $(*\Rightarrow *)$ are projections into the dynamic type; coercions $p?\text{Int}$ and $p?\ (\Rightarrow *)$ are projections from the dynamic type; arrow $\Rightarrow$ coeﬁces functions; and semicolon $;$ composes coercions—we discuss the details later. For now, observe that our first cast, which can allocate positive blame but not negative, compiles to a coercion that contains $\ell$ but not $\ähr$, while our second cast, which can allocate negative blame but not positive, compiles to a coercion that contains $\ähr$ but not $\ell$.

Safety for $\lambda B$ depends on a somewhat subtle deﬁnition of positive and negative subtyping, $\ll$ and $\lle$. In contrast, safety for $\lambda C$ has pleasingly simple definition: a coercion is safe for $p$ if it does not contain $p$. Consider a cast from $A$ to $B$ with path $p$; it turns out that $A \ll \lle B$ if and only if the cast compiles to a coercion not containing $p$; and $A \ll \lle B$ if and only if the cast compiles to a coercion not containing $\ähr$. The forward direction of this implication is only to be expected, but the backward direction is a surprise, and a pleasant one: it means the somewhat subtle deﬁnition of $\ll$ is justiﬁed by the correspondence to the coercion calculus.

Details of the coercion calculus $\lambda C$ appear in Section 5.

### 2.4 Threesome calculus, $\lambda T$

A naive implementation of the blame or coercion calculus will suffer space leaks. For instance, two mutually recursive procedures...
where the recursive calls are in tail position should run in constant space; but if one of them is statically typed and the other is dynamically typed, the mediating casts break the tail call property, and the program will require space proportional to the number of calls.

This problem was observed by Herman et al. (2010), who propose a solution based on the coercion calculus. If two coercions are applied in sequence then they are composed and normalised. Any sequence of coercions can be represented in bounded space, where the bound is the size of the largest coercion. (To be precise, the sequence of coercions can be represented in bounded space, where program will require space proportional to the number of calls. Always compresses to an equivalent sequence of two casts \( \perp \).

The notation for decorated types is opaque, and it is not immediately in one-to-one correspondence with normalised coercions. However, a type with a type decorated to indicate how blame is allocated.

Threesomes, like coercions, enable any sequence of casts to be read as a type yields the mediating type. Blame is restored by replacing the mediating type with a type decorated to indicate how blame is allocated.

Siek and Wadler (2010) demonstrate that the decorated types are in one-to-one correspondence with normalised coercions. However, the notation for decorated types is opaque, and it is not immediately obvious what notations such as \( \perp \), \( P \rightarrow^p Q \), or \( B^p \) mean. Garcia (2013) exploits the correspondence with coercions to calculate the form of decorated types, but retains the problematic notation of Siek and Wadler and adds other notations, such as \( ? \rightarrow \perp ? \).

Here we introduce a new notation that transparently corresponds to coercions in normal form. For instance, the coercion \( (\ell? (\times \rightarrow \times)) \rightarrow (\(((\ell? (\Int !)) \rightarrow (\Int !)) \rightarrow (\ell? (\Int !))) \) corresponds to the threesome \( \ell? (((\ell? (\Int !)) \rightarrow (\Int !)) \rightarrow (\ell? (\Int !))) \).

Notation is simplified rather than multiplied. In the new syntax, it is trivial to read a threesome as a coercion in normal form (by expanding some occurrences of ! and \( \ell ? \)) or to read it as a type (by dropping all occurrences of ! and \( \ell ? \), in this case yielding \( (\Int !) \rightarrow (\Int !) \)). The reading as a type yields the mediating type of a threesome for the form of threesomes that ignores blame.

Details of the threesome calculus \( \Lambda T \) appear in Section 6.

3. Blame calculus

Figure 2 defines the blame calculus, \( \lambda B \). Results in this section appear in Wadler and Findler (2009) and Ahmed et al. (2011).

Let \( A, B, C \) range over types. A type is either a base type \( K \), a function type \( A \rightarrow B \), or the dynamic type \( * \). Let \( G, H \) range over ground types. A ground type is either a base type \( K \) or the function type \( * \rightarrow * \). The dynamic type satisfies the domain equation \( * \cong K + (K \rightarrow K) \) so each value of type dynamic is an injection from a ground type.

Types \( A \) and \( B \) are compatible, written as \( A \sim B \), if a cast from \( A \) to \( B \) may succeed. Type \( * \) is compatible with every type, base types with themselves, and functions are compatible if their domains and ranges are compatible. Every type other than the dynamic type is compatible with exactly one ground type, and ground types are compatible if and only if they are identical.

**Lemma 1** (Ground types).

- If \( A \neq * \) there exists a unique ground type \( G \) such that \( A \sim G \).
- If \( G \) and \( H \) are ground types, then \( G \sim H \) iff \( G = H \).

Compatibility is reflexive and symmetric but not transitive. In particular, if \( G \neq H \), then \( G \sim * \) and \( * \sim H \) but \( G \not\sim H \). This is the source of all blame: casting from one type into the dynamic type, and then attempting to cast out at an incompatible type.

Let \( \ell \) range over blame labels, and \( p, q \) range over blame paths, which are of the form \( \ell \circ \ell \). To indicate on which side of a cast blame lays, each blame path \( p \) has a complement \( \overline{p} \). Complement is involutive, so \( \overline{\overline{p}} = p \).

Let \( L, M, N \) range over terms. Terms are those of the simply-typed lambda calculus, plus casts. Type judgments take the form \( \Gamma \vdash B : A \), where \( \Gamma \) is a type environment pairing variables with types. Each operator \( op \) has a type system specific to that type.

The typing rules for constants, operators, variables, abstraction, and application are standard (and not repeated in subsequent figures). Typing, reduction, and safety judgments are written with subscripts indicating to which calculus they belong, except we omit subscripts in figures to avoid clutter.

The typing rule for casts is straightforward:

\[
\begin{align*}
\Gamma \vdash B & : A \sim B \\
\Gamma \vdash (M : A \overrightarrow{B} B) & : B
\end{align*}
\]

If term \( M \) has type \( A \), and type \( A \sim B \), then the cast of \( M \) from \( A \) to \( B \) is a term of \( B \). The cast is decorated with a blame path \( p \). If a cast from \( A \) to \( B \) decorated with \( p \) allocates blame to \( p \) we say it has positive blame, meaning the fault lies with the term contained in the cast; and if it allocates blame to \( \overline{p} \) we say it has negative blame, meaning the fault lies with the context containing the cast.

Every value of dynamic type is constructed by a cast from ground type, written \( M : G : \Rightarrow * \). Such casts cannot fail, so they are not decorated with a blame path. It is convenient to distinguish such undecorated casts for a technical reason, explained below.

Let \( V, W \) range over values. A value is a constant, a lambda term, a decorated cast between values of function type, or an undecorated cast of a value from ground type to dynamic type. Let \( E \) range over evaluation contexts, which are standard, and include casts in the obvious way.

We write \( M \overrightarrow{N} p \) to indicate that term \( M \) steps to term \( N \), and \( M \overrightarrow{\text{blame}} p \) to indicate that term \( M \) allocates blame to path \( p \). We treat \( \text{blame} p \) as distinct from a term. Rules (BETA), (DELTA) are standard (and not repeated in subsequent figures). Rule (BASE): a cast from a base type to itself leaves the value unchanged. Rule (WRAP): a cast of a function applied to a value reduces to a term that casts on the domain, applies the function, and casts on the range; in order to allocate blame correctly, the blame path on the cast of the domain is complemented, corresponding to the fact that function types are contravariant in the domain and covariant in the range (Findler and Felleisen 2002; Wadler and Findler 2009).

Rule (GROUND): a cast from type \( A \) to type \( * \) factors into a cast from \( A \) to the unique ground type \( G \) that is compatible with \( A \), decorated with the original blame path, followed by an undecorated cast from \( G \) to \( * \). This reduction motivates the distinction between casts with and without decoration: if they were conflated this rule would lead to infinite regress. Rules (COLLAPSE), (CONFLICT): a cast from type \( * \) to type \( A \) examines the ground type \( G \) of the value of type \( * \). If \( G \) is compatible with \( A \), then the two casts collapse to a direct cast from \( G \) to \( A \), otherwise the offending cast is blamed.

Embedding \( [M] \) takes terms of dynamically-typed lambda calculus into the blame calculus. The embedding introduces a fresh label \( \ell \) for each cast from the dynamic type.
Syntax

Variables $x, y$
Constants $k ::= 0 | 1 | ⋯ | true | false | ⋯$
Operators $op ::= + | \leq | ⋯$
Base types $K ::= \text{Int} | \text{Bool}$
Terms $L, M, N ::= k \mid \text{op}(\tilde{M}) \mid x \mid \lambda x. A. N \mid L M \mid M : A \xrightarrow{P} B \mid M : G \Rightarrow ⋯$
Values $V, W ::= k \mid \lambda x. A. N \mid V : A \Rightarrow B \xrightarrow{P} A' \Rightarrow B' \mid V : G \Rightarrow ⋯$
Evaluation context $E ::= \square \mid E[\text{op}(\tilde{V}, \square, \tilde{M})] \mid E[\square M] \mid E[V \square] \mid E[\square : A \Rightarrow P] \mid E[\square : G \Rightarrow ⋯]$

Compatibility

$$A \sim B$$

Term typing

$$k : K \quad \Gamma \vdash k : K$$

$$\Gamma \vdash M : K \quad \Gamma \vdash \text{op}(\tilde{M}) : K$$

$$\Gamma \vdash x : A$$

$$\Gamma, x : A \vdash N : B$$

$$\Gamma \vdash \lambda x. A. N : A \Rightarrow B$$

$$\Gamma \vdash L : A \Rightarrow B$$

$$\Gamma \vdash M : A$$

Reduction

$$E[\text{op}(\tilde{V})] \rightarrow E[(\text{op})(\tilde{V})]$$

$$E[(\lambda x. A. N) V] \rightarrow E[N[x:=V]]$$

$$E[V : K \xrightarrow{P} K] \rightarrow E[V]$$

$$E[(V : A \Rightarrow B \xrightarrow{P} A' \Rightarrow B')] W] \rightarrow E[V (W : A' \Rightarrow B) : B \xrightarrow{P} B']$$

$$E[V : A \xrightarrow{P} \star] \rightarrow E[(V : A \xrightarrow{P} G) : G \Rightarrow \star]$$

if $\star \neq A$ and $A \sim G$

$$E[(V : G \Rightarrow \star) : \star \xrightarrow{P} A] \rightarrow E[V : G \xrightarrow{P} A]$$

if $G \sim A$

$$E[(V : G \Rightarrow \star) : \star \xrightarrow{P} A] \rightarrow \text{blame } p$$

if $G \not\sim A$

Positive and negative subtype

$$A <:+ \quad A <:-$$

$$k <:+ K \quad A' <:+ A \quad B <:+ B' \quad A <:+ \star$$

$$A \rightarrow B <:+ A' \rightarrow B'$$

$$K <:- K \quad \star <:- B \quad A <:- G \quad A <:- \star$$

Safe cast

$$(A \xrightarrow{P} B) \text{ safe } p$$

Embedding of dynamically-typed lambda calculus

$$[k] = k : K \Rightarrow \star$$

if $k : K$

$$[\text{op}(\tilde{M})] = \text{op}([\tilde{M}] : \star \xrightarrow{P} \tilde{K}) : K \Rightarrow \star$$

if $\text{op} : \tilde{K} \rightarrow K$

$$[x] = x$$

$$[\lambda x. M] = (\lambda x. \star. [M]) : \star \rightarrow \star \Rightarrow \star$$

$$[M. N] = ([M] : \star \xrightarrow{P} \star \rightarrow \star) [N]$$

Figure 2. Blame calculus (LAB)
3.1 Type safety

Type safety is established via preservation and progress.

**Proposition 2** (Type safety for blame calculus).

- If \( \vdash_B M : A \) and \( M \rightarrow_B N \) then \( \vdash_B N : A \).
- If \( \vdash_B M : A \) then either
  - there exists a value \( V \) such that \( M = V \), or
  - there exists a term \( N \) such that \( M \rightarrow_B N \), or
  - there exists a path \( p \) such that \( M \rightarrow_{blame} p \).

Preservation and progress for the blame calculus do not rule out blame as a result. How to guarantee that blame cannot arise in certain circumstances is the subject of the next section.

3.2 Blame safety

We write \( A <^+ B \) or \( A <^− B \) if a cast from type \( A \) to type \( B \) never allocates positive or negative blame, respectively. A cast from a base type to itself never allocates blame (as per (BASE)). A cast from a function to a function never allocates positive blame if the cast of the domain never allocates negative blame and if the cast of the range never allocates positive blame; and ditto with positive and negative reversed; each rule is contravariant in the function domain and covariant in the function range (as per (WRAP)). A cast to dynamic never allocates positive blame, while a cast from dynamic never allocates negative blame. A cast from ground type to dynamic never allocates blame (as per (GROUND)).

A cast from \( A \) to \( B \) decorated with \( p \) is safe for blame \( q \), written

\[
(A \xrightarrow{p} B) \text{ safe}_{blame} q
\]

if evaluation of the cast can never yield blame \( q \). The three rules reflect that if \( A <^+ B \) the cast never allocates positive blame, if \( A <^− B \) the cast never allocates negative blame, and a cast with path \( p \) never allocates blame other than to \( p \) or \( \overline{p} \). Safety extends to terms in the obvious way: we write \( M \text{ safe}_{blame} q \) if every cast in \( M \) is safe for \( q \). Blame safety is established via a variant of preservation and progress.

**Proposition 3** (Blame safety for the blame calculus).

- If \( M \text{ safe}_{blame} q \) and \( M \rightarrow_B N \) then \( N \text{ safe}_{blame} q \).
- If \( M \text{ safe}_{blame} q \) then \( M \rightarrow_{\overline{p}} \text{ blame} \).

4. Precise blame calculus

Figure 3 defines the precise blame calculus, \( \lambda P \).

Types are the same as those of the blame calculus, as are terms, except for the grammar of blame paths. A blame path begins with a label \( \ell \) and is followed by a sequence of tags, standing for domain, range, injection, and projection. There is no longer an operation of blame complementation.

The typing rules are unchanged. The definition of values and evaluation context are as before, except for the change in blame paths. The reduction rules are as before, except that each of the rules (WRAP), (GROUND), and (COLLAPSE) adds tags to the blame path as appropriate.

4.1 Type safety

**Proposition 4** (Type safety for precise blame calculus).

- If \( \vdash_B M : A \) and \( M \rightarrow_B N \) then \( \vdash_B N : A \).
- If \( \vdash_B M : A \) then either
  - there exists a value \( V \) such that \( M = V \), or
  - there exists a term \( N \) such that \( M \rightarrow_B N \), or
  - there exists a path \( p \) such that \( M \rightarrow_{\overline{p}} \text{ blame} \).
Paths to blame labels ($\lambda P$ to $\lambda B$)

\[
|p|_{\lambda B}^P = \ell \\
|p/\text{dom}|_{\lambda B}^P = |p|_{\lambda B}^P \\
|p/\text{rng}|_{\lambda B}^P = |p|_{\lambda B}^P \\
|p/\text{inj}|_{\lambda B}^P = |p|_{\lambda B}^P \\
|p/\text{prj}|_{\lambda B}^P = |p|_{\lambda B}^P
\]

Figure 4. Relating blame and precise blame calculi ($\lambda B$ and $\lambda P$)

4.2 Blame safety

We write $p \subseteq q$ if path $p$ is contained within path $q$. For example, $\ell/\text{prj}/\text{dom}$ is contained in $\ell/\text{prj}/\text{dom}/\text{inj}/\text{dom}$. A cast from $A$ to $B$ decorated with $p$ is safe for blame $q$, written

\[
(A \xrightarrow{p} B) \text{ safe } q
\]

if evaluation of the cast can never allocate a path that contains $q$, or a path that contains, but not exactly $q$. The rules for safety are closely related to the corresponding reduction rules. Safety extends to terms in the obvious way. As with $\lambda B$, blame safety for $\lambda P$ is established via a variant of preservation and progress.

Proposition 5 (Blame safety for precise blame calculus),

- If $M \text{ safe}_P p$ and $M \rightarrow_P N$ then $N \text{ safe}_P p$
- If $M \text{ safe}_P p$ then $M \xrightarrow{\text{prj}} \text{ blame } p$

4.3 Translation to blame calculus

This section considers how $\lambda P$ relates to $\lambda B$. In this section, we let $M$, $N$ and $p$ range over terms and paths of $\lambda B$, and let $M'$, $N'$ and $p'$ range over terms and paths of $\lambda P$.

We write $|p'|_{\lambda B}^P = p$ to indicate that the precise path on the left translates to the path on the right (Figure 4). The translation drops all tags and takes the complement of any path underneath a domain tag. The translation extends to terms in the obvious way: we write $|M'|_{\lambda B}^P = M$ when $M'$ is a term of the precise blame calculus and $M$ is the term of the blame calculus obtained by replacing each precise path $p'$ by its corresponding path $p = |p'|_{\lambda B}^P$.

The translation from $\lambda P$ to $\lambda B$ preserves and reflects type and blame safety.

Proposition 6 (Precise blame preserves and reflects blame),

- If $M = |M'|_{\lambda B}^P$ then
  - $\Gamma \vdash_{\lambda B} M : A \iff \Gamma \vdash_{\lambda P} M' : A$,
  - $M \text{ safe}_B p \iff M' \text{ safe}_P p'$ for all $p'$ such that $p = |p'|_{\lambda B}^P$.

The translation from $\lambda P$ to $\lambda B$ is a bisimulation.

Proposition 7 (Precise blame bisimulates blame),

- If $M' \rightarrow_{\lambda P} N'$ then $|M'|_{\lambda B}^P \rightarrow_{\lambda B} |N'|_{\lambda B}^P$.
- If $M' \rightarrow_{\lambda P} \text{ blame } p'$ then $|M'|_{\lambda B}^P \rightarrow_{\lambda B} \text{ blame } |p'|_{\lambda B}^P$.
- If $M \rightarrow_{\lambda B} N$ and $M = |M'|_{\lambda B}^P$ then there exists an $N'$ such that $M' \rightarrow_{\lambda P} N'$ and $N = |N'|_{\lambda B}^P$.
- If $M \rightarrow_{\lambda B} \text{ blame } p$ and $M = |M'|_{\lambda B}^P$ then there exists a $p'$ such that $M' \rightarrow_{\lambda P} \text{ blame } p'$ and $p = |p'|_{\lambda B}^P$.

The bisimulation is on the nose: a single step in one system corresponds to a single step in the other system.

5. Coercion calculus

Figure 5 defines the coercion calculus, $\lambda C$. The coercions of this calculus correspond to those of Henglein (1994), except that the coercion from dynamic type to ground type is decorated with a blame path, as in Siek and Wadler (2010). The blame paths used in this section may be taken to be either those of $\lambda B$ or $\lambda P$, and types are the same as those in $\lambda B$ and $\lambda P$.

Let $c, d$ range over coercions. We write $c : A \Rightarrow B$ to indicate that $c$ coerces values of type $A$ to type $B$. The identity coercion at type $A$ is written $\text{id}(A)$. The injection from ground type $G$ to dynamic type is written $G!$, and projection from dynamic type to ground type $G$ is written $p? G$. The latter is decorated with a blame path $p$, to which blame is allocated if the projection fails. A function coercion $c \rightarrow d$ coerces a function $A \rightarrow B$ to a function $A' \rightarrow B'$, where $c$ coerces $A'$ to $A$, and $d$ coerces $B$ to $B'$. This construct is contravariant in the domain coercion $c$ and covariant in the range coercion $d$. Finally, the composition $c ; d$ coerces $A$ to $C$, where $c$ coerces $A$ to $B$, and $d$ coerces $B$ to $C$.

Terms of the calculus are as before, except that we replace casts (both decorated and undecorated) by application of a coercion. The typing rule for coercion application is straightforward:

\[
\Gamma \vdash_{\lambda C} M : A \quad c : A \Rightarrow B \\
\Gamma \vdash_{\lambda C} (M : A \xrightarrow{c} B)
\]

If term $M$ has type $A$, and $c$ coerces $A$ to $B$, then the application of $c$ to $M$ is a term of type $B$. For ease of comparison, we write casts and coercion application with a similar notation. The types in a coercion application are not required, and can be omitted at run time in an implementation.

Each coercion uniquely determines its source and target types, but the converse is not true. Unlike a cast, the source and target type of a coercion are not necessarily compatible, due to coercion composition.

Values and evaluation contexts are as in the blame calculus, except that casts are replaced by corresponding coercions. The identity coercion leaves a value unchanged ($\text{Id}$). Application of a coerced function applies the domain coercion to the argument and the range coercion to the result ($\text{WRAP}$). If an injection meets a matching projection, the coercion behaves as the identity ($\text{COLLAPSE}$). In case of a mismatch, the coercion allocates blame to the path in the projection ($\text{CONFLICT}$). Application of a composed coercion applies each of the coercions in turn ($\text{DECOMPOSE}$).

5.1 Type safety

Proposition 8 (Type safety for coercion calculus),

- If $\Gamma \vdash_{\lambda C} M : A$ and $M \rightarrow_{\lambda C} N$ then $\Gamma \vdash_{\lambda C} N : A$.
- If $\Gamma \vdash_{\lambda C} M : A$ then either
  - there exists a value $V$ such that $M = V$, or
  - there exists a term $N$ such that $M \rightarrow_{\lambda C} N$, or
  - there exists a path $p$ such that $M \rightarrow_{\lambda C} \text{ blame } p$.

5.2 Blame safety

A coercion $c$ is safe for blame path $q$, written $c \text{ safe}_C q$, if application of the coercion never allocates blame $q$. Unlike previously, the definition does not depend on source and target types, so they are omitted. The definition is pleasingly simple: a coercion is safe for $q$ if it does not mention the path $q$. Again, blame safety for $\lambda C$ is established via a variant of preservation and progress.

Proposition 9 (Blame safety for coercion calculus),

- If $M \text{ safe}_C p$ and $M \rightarrow_{\lambda C} N$ then $N \text{ safe}_C p$.
- If $M \text{ safe}_C p$ then $M \xrightarrow{\text{prj}} \text{ blame } p$. 

5.3 Canonical coercions

Canonical coercions help us to relate coercions to thresomes. The definition is similar to that of Henglein (1994, Section 2.5).

For any A and B there is a canonical coercion \( c : A \leadsto B \), unique up to blame labels. The next section uses the following fact.

**Lemma 10 (Failure).** Consider coercion \( f : A \leadsto B \) defined by

\[
f = c \circ (G!); (p?H); d
\]

with \( * \neq A \) and \( A \sim G \), where \( c : A \leadsto G \) is a canonical coercion and \( d : H \leadsto B \) is any coercion. Then

\[
V : A \leadsto B \leadsto \text{blame } p
\]

for any V such that \( \vdash_c V : A \).

5.4 Translation from and to blame calculus

This section considers the relation of \( \lambda C \) to \( \lambda B \), and the next section considers the relation to \( \lambda P \). Figure 6 presents the translations. In this section, we let \( M, N \) range over terms of \( \lambda B \) and \( M', N' \) range over terms of \( \lambda C \). The translation from blame to coercions is taken from Siek and Wadler (2010). The reverse translation is novel and yields a sequence of casts instead of a single cast.

We write

\[
[A \xrightarrow{P} B]^{bc} = c
\]

to indicate that the cast on the left translates to the coercion on the right. We write

\[
[c]^{cb} = [A_1 \xrightarrow{P_1} A_2, \ldots, A_m \xrightarrow{P_m} A_{m+1}]
\]
Blame to coercion ($\lambda B$ to $\lambda C$)

\[ |K \xrightarrow{P} K^{BC} = \text{id}(K) | \]
\[ |A \rightarrow B \xrightarrow{P} A' \rightarrow B'|^{BC} = |A' \xrightarrow{P} A|^{BC} \rightarrow |B \xrightarrow{P} B'|^{BC} | \]
\[ |\ast \xrightarrow{P} *|^{BC} = \text{id}(\ast) | \]
\[ |A \xrightarrow{P} *|^{BC} = |A \xrightarrow{P} G|^{BC}; (G!) \text{ if } G \sim A \text{ and } A \neq \ast | \]
\[ |\ast \xrightarrow{P} A|^{BC} = (p \ ? \ G); |G \xrightarrow{P} A|^{PC} \text{ if } G \sim A \text{ and } A \neq \ast | \]

Coercion to blame ($\lambda C$ to $\lambda B$)

\[ |\text{id}(A)|^{CB} = [ ] | \]
\[ |G|^{|CB} = |G \xrightarrow{\ast} \ast | \]
\[ |p \ ? \ G|^{|CB} = [ \ast \xrightarrow{P} G | \]
\[ |c; d|^{|CB} = |c|^{|CB} + |d|^{|CB} | \]
\[ |c|^{CB} = [A_1 \xrightarrow{P_1} A_2, \ldots, A_n \xrightarrow{P_n} A_{n+1}] | \]
\[ |c \rightarrow d|^{|CB} = [A_{n+1} \rightarrow B_1 \xrightarrow{P_{n+1}} A_m \rightarrow B_1, \ldots, A_2 \rightarrow B_1 \xrightarrow{P_2} A_1 \rightarrow B_1, A_1 \rightarrow B_1 \xrightarrow{P_1} A_1 \rightarrow B_{n+1}] | \]

Precise blame to coercion ($\lambda P$ to $\lambda C$)

\[ |K \xrightarrow{P} K^{PC} = \text{id}(K) | \]
\[ |A \rightarrow B \xrightarrow{P} A' \rightarrow B'|^{PC} = |A' \xrightarrow{P/\text{dom}} A|^{PC} \rightarrow |B \xrightarrow{P/\text{rng}} B'|^{PC} | \]
\[ |\ast \xrightarrow{P} *|^{PC} = \text{id}(\ast) | \]
\[ |A \xrightarrow{P} *|^{PC} = |A \xrightarrow{P/\text{inj}} G|^{PC}; (G!) \text{ if } G \sim A \text{ and } A \neq \ast | \]
\[ |\ast \xrightarrow{P} A|^{PC} = (p \ ? \ G); |G \xrightarrow{P/\text{prj}} A|^{PC} \text{ if } G \sim A \text{ and } A \neq \ast | \]

Figure 6. Relating blame, precise blame, and coercion calculi ($\lambda B$, $\lambda P$, and $\lambda C$)

to indicate that the coercion on the left translates to the sequence of zero or more casts on the right. Both translations extend to terms in the obvious way.

A coercion translated from a cast contains at most a single blame label and its complement. However, an arbitrary coercion may contain many blame labels, which is why the inverse translation must yield a sequence of casts, not a single cast. The identity coercion translates to an empty sequence, while the composition of two coercions appends the corresponding sequences. Injection translates to a cast decorated with an irrelevant blame label (written $\ast$). One might think it could translate to an undecorated cast, but further translation of such a cast in the domain or range of a function (see below) requires a decorated cast. Projection translates straightforwardly. Translation of functions is the most interesting case. Casts in the domain are complemented, but this complementation is undone when rule (WRAP) of $\lambda B$ is applied, yielding the original label.

Neither translation is inverse to the other. One might hope that if one starts with a coercion, translates it to a sequence of casts, and translates each cast back to a coercion, then the composition of the sequence of coercions yields original coercion, but this is not the case. We conjecture that the two coercions are equivalent (in that they translate to the same threesome), but we have not proved it.

The somewhat subtle definition of positive and negative subtyping is justified by the correspondence to the coercion calculus.

Lemma 11 (Positive and negative subtyping).

- $A <:^+$ $B$ iff $|A \xrightarrow{P} B|^{BC} \text{ safe}_C \ p$.
- $A <:^-$ $B$ iff $|A \xrightarrow{P} B|^{BC} \text{ safe}_C \ p$.

The translation from $\lambda B$ to $\lambda C$ preserves type and blame safety, as does the reverse translation.

Proposition 12 (Blame preserves coercions, and vice versa).

- If $\Gamma \vdash_B M : A$ then $\Gamma \vdash_C |M|^{BC} : A$.
- If $\Gamma \vdash_C M' : A$ then $\Gamma \vdash_B |M'|^{CB} : A$.
- If $M \text{ safe}_B$ $p$ then $|M|^{BC} \text{ safe}_C$ $p$.
- If $M' \text{ safe}_C$ $p$ then $|M'|^{CB} \text{ safe}_B$ $p$.

The translation from $\lambda B$ to $\lambda C$ is a simulation, as is the reverse translation. We write $\rightarrow^*$ for the reflexive and transitive closure of the reduction relation $\rightarrow$, and we write $M \rightarrow^* \text{ blame} \ p$ if there exists $N$ such that $M \rightarrow^* N$ and $N \rightarrow \text{ blame} \ p$.

Proposition 13 (Blame simulates coercions, and vice versa).

- If $M \rightarrow^* N$ then $|M|^{BC} \rightarrow^*_C |N|^{BC}$.
\begin{itemize}
\item If \( M \rightarrow_B \text{blame } p \) then \( |M|_{CB} \rightarrow_T^\blame \text{blame } p \).
\item If \( M' \rightarrow_T N' \) then \( |M'|_{CB} \rightarrow_B |N'|_{CB} \).
\item If \( M' \rightarrow_T \text{blame } p \) then \( |M|_{CB} \rightarrow_B^p |N|_{CB} \rightarrow_B \text{blame } p \).
\end{itemize}

The simulations are not on the nose: a single step in one system corresponds to zero or more steps in the other system.

### 5.5 Translation from precise blame calculus

In this section, let \( M, N \) range over terms of \( \mathcal{X} \) and \( M', N' \) range over terms of \( \mathcal{C} \). The translation from \( \mathcal{X} \) to \( \mathcal{C} \) closely resembles that from \( \lambda \mathcal{B} \) to \( \lambda \mathcal{C} \), except for the addition of precise blame labels.

We write
\[
|A|_{\text{PC}} = c
\]
to say that the cast on the left translates to the coercion on the right (Figure 6). The translation extends to terms in the obvious way.

The inverse translation is not possible. The trick used to translate functions does not work here: instead of complementing the blame path we would need operations that yield the original path after a domain tag is added; but this is possible only when the path itself happens to begin with a domain tag, which may not be the case. Range, projection, and injection tags raise similar issues.

The translation from \( \lambda \mathcal{P} \) to \( \lambda \mathcal{C} \) preserves type and blame safety and the translation is a simulation.

**Proposition 14** (Precise blame preserves coercions).

- If \( \Gamma \vdash \mu M : A \) then \( \Gamma \vdash^\mu |M|_{\text{PC}} : A \).
- If \( M \text{ safe } p \) then \( |M|_{\text{PC}} \text{ safe } p \).

**Proposition 15** (Coercions simulate precise blame).

- If \( M \rightarrow^p N \) then \( |M|_{\text{PC}} \rightarrow^\mu |N|_{\text{BC}} \).
- If \( M \rightarrow_T \text{blame } p \) then \( |M|_{\text{PC}} \rightarrow_B^p |N|_{\text{BC}} \rightarrow_B \text{blame } p \).

The simulations are not on the nose: a single step in one system corresponds to zero or more steps in the other system. We conjecture that this simulation may in fact be a bisimulation, but we have not proved it.

### 6. Threesome calculus

Figure 7 defines the threesome calculus, \( \lambda \mathcal{T} \). Threesomes correspond to coercions in a particular normal form. Threesomes are introduced in Siek and Walder (2010), and their relation to coercions is discussed there and in Garcia (2013); but the presentation here is more transparent. The blame paths used in this section may be taken to be either those of \( \lambda \mathcal{B} \) or \( \lambda \mathcal{P} \), and types are as in \( \lambda \mathcal{B} \).

Let \( s, t \) range over threesome coercions, \( i \) range over injection coercions, and \( g, h \) range over ground coercions. Every ground coercion is an injection coercion, and every injection coercion is a threesome coercion. We write \( t : A \Rightarrow B \) to indicate that \( t \) coerces values of type \( A \) to type \( B \).

Instead of permitting arbitrary composition, threesome coercions follow a specific grammar that incorporates a restricted form of composition. A ground coercion is either \( K \) (corresponding to the coercion \( \text{id}(K) \)) or \( s \Rightarrow t \) (corresponding to the coercion \( s \Rightarrow t \)). A threesome coercion either is of the form \( \ast \) (corresponding to the coercion \( \text{id}(\ast) \)), or is of the form \( p ? i \) (corresponding to the coercion \( \text{id}(p ? i) \), \( i \in G \Rightarrow A \)), or is an injection coercion. An injection coercion either is of the form \( g ! \) (corresponding to the coercion \( g \Rightarrow G \), \( g \in G \Rightarrow A \)), or is a ground coercion, or is of the form \( G \Rightarrow H \) (corresponding to a coercion \( c : (G) \Rightarrow (H) \) \( d \), where \( c : A \Rightarrow G \) and \( d : H \Rightarrow B \) are canonical coercions, with \( \ast \neq A \) and \( A \sim G \) and \( G \neq H \); by Lemma 10, such a coercion always fails yielding blame \( p \)). By design, any type \( A \) also happens to be a threesome coercion. The correspondence between threesomes and coercions is spelled out in the translation from \( \lambda \mathcal{T} \) to \( \lambda \mathcal{C} \) given in Figure 8.

Unlike in \( \lambda \mathcal{C} \), a threesome coercion does not uniquely determine its source or target type, because the source and target type of a failure coercion \( G \Rightarrow H \) are not specified. An alternative design might add source and target types to the form \( G \Rightarrow H \), but this choice would complicate the definition of \( \cdot \Rightarrow \cdot \) considered below.

The source of every injection coercion is compatible with a unique ground type, as is the target of every ground coercion, as given by \( \text{src}(i) \) and \( \text{trg}(g) \). Further, the source and target of a ground coercion are always compatible with the same ground type.

**Lemma 16** (Source and Target).

- \( \text{src}(i) = G \) if \( i : A \Rightarrow B \) and \( A \sim G \).
- \( \text{trg}(g) = G \) if \( g : A \Rightarrow B \) and \( B \sim G \).
- If \( g : A \Rightarrow B \) then \( \text{src}(g) = \text{trg}(g) \).

Terms of the calculus are as in \( \lambda \mathcal{C} \), except that we replace coercions by threesome coercions. As with the coercion calculus, the types in a threesome coercion application are not required, and can be omitted at run time in an implementation.

The key idea, as in Herman et al. (2010) and Siek and Walder (2010), is to collapse adjacent coercions, which ensures space efficiency. Collapsing requires some adjustments to values and evaluation contexts. Let \( U \) range over uncoerced values and \( V, W \) range over values, where an uncoerced value contains no top-level coercion, and a value at most one. Let \( F \) range over cast-free contexts and \( E \) range over contexts, where the innermost term of a cast-free context is not a cast. Reduction of a term that is a cast must occur in a cast-free context. These adjustments ensure that if a term contains two casts in succession in an evaluation context that those casts are reduced by rule (COMPOSE) before other reductions occur. All the other reduction rules are straightforward.

Composition of threesome coercions is computed by \( \cdot \Rightarrow \cdot \). If threesomes \( s \) and \( t \) correspond to the normal forms of coercions \( c \) and \( d \), then the threesome \( s \cdot t \) corresponds to the normal form of the coercion \( c ; d \). The grammar of threesome coercions makes the definition of threesome composition easy to express.

#### 6.1 Type safety

**Proposition 17** (Type safety for threesome calculus).

- If \( \cdot \Rightarrow_T M : A \Rightarrow M \Rightarrow N \Rightarrow A \).
- If \( \cdot \Rightarrow_T M : A \) then either
  - there exists a value \( V \) such that \( M \Rightarrow V \), or
  - there exists a term \( N \) such that \( M \Rightarrow N \), or
  - there exists a path \( p \) such that \( M \Rightarrow \text{blame } p \).

#### 6.2 Blame safety

A threesome \( t \) is safe for path \( q \), written \( t \text{ safe } q \), if application of the threesome never blames \( p \). Again, the definition is simple: a threesome is safe for \( q \) if it does not mention the path \( q \). Blame safety for \( \lambda \mathcal{T} \) is established via a variant of preservation and progress.

**Proposition 18** (Blame safety for threesome calculus).

- If \( M \text{ safe } p \) and \( M \Rightarrow N \Rightarrow N \text{ safe } p \).
- If \( M \text{ safe } p \) then \( M \Rightarrow \text{blame } p \).

#### 6.3 Translation from and to coercion calculus

This section considers how \( \lambda \mathcal{T} \) relates to \( \lambda \mathcal{C} \). Figure 8 presents the translations. In this section, we let \( M, N \) range over terms of \( \lambda \mathcal{C} \) and let \( M', N' \) range over terms of \( \lambda \mathcal{T} \).

We write
\[
[e] = t
\]
Syntax

<table>
<thead>
<tr>
<th>Category</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threesome coercions</td>
<td>( s, t \ ::= \star</td>
</tr>
<tr>
<td>Injection coercions</td>
<td>( i \ ::= g , !</td>
</tr>
<tr>
<td>Ground coercions</td>
<td>( g, h \ ::= K ,</td>
</tr>
<tr>
<td>Terms</td>
<td>( L, M, N \ ::= k ,</td>
</tr>
<tr>
<td>Uncoerced values</td>
<td>( U \ ::= k ,</td>
</tr>
<tr>
<td>Values</td>
<td>( V, W \ ::= U ,</td>
</tr>
<tr>
<td>Cast-free context</td>
<td>( F \ ::= \square ,</td>
</tr>
<tr>
<td>Evaluation context</td>
<td>( E \ ::= F ,</td>
</tr>
</tbody>
</table>

Threesome typing

\[
\begin{align*}
K : K & \Longrightarrow K \\
(s \to t) : A & \Longrightarrow B \\
\ast \to \ast & \Longrightarrow \ast \\
(g!) : A & \Longrightarrow \ast \\
(p \, ? \, i) : \ast & \Longrightarrow A \\
\ast \, \not\equiv A & \Longrightarrow \ast \\
(G \, p \, H) : A & \Longrightarrow B
\end{align*}
\]

Evaluation context \( E \) and Threesome coercions \( s, t \)

\[
\begin{align*}
E[\Gamma] & : t : A \Rightarrow B \\
G[\Gamma] & : M : A \\
\end{align*}
\]

Threesome composition

\[
\begin{align*}
K \triangleright K & = K \\
(s \to t) \triangleright (s' \to t') & = (s' \triangleright s) \to (t \triangleright t') \\
\ast \triangleright t & = t \\
(g!) \triangleright \ast & = g! \\
(p \, ? \, i) \triangleright t & = p \, ? \, (i \triangleright t) \\
g \triangleright (h \, !) & = (g \triangleright h) ! \\
(g!) \triangleright (p \, ? \, i) & = \begin{cases} 
\{ g \triangleright i \} & \text{if } G = H \\
\{ G \, p \, H \} & \text{if } G \neq H
\end{cases}
\text{where } G = \text{trg}(g) \text{ and } H = \text{src}(i) \\
g \triangleright (G \, p \, H) & = G \, p \, H
\end{align*}
\]

Evaluation context \( E \) and Threesome coercions \( s, t \)

\[
\begin{align*}
E[\Gamma] & : t : A \Rightarrow B \\
G[\Gamma] & : M : A \\
\end{align*}
\]

Reduction

\[
\begin{align*}
E[(U : A \to B \, \triangleright \to t \, A' \to B') \, V] & \longrightarrow E[U \, (V : A' \, \triangleright \to s \, A) : B \, \triangleright \to B'] \quad \text{WRAP} \\
F[U : K \Longrightarrow K] & \longrightarrow F[U] \quad \text{BASE} \\
F[U : \ast \Longrightarrow \ast] & \longrightarrow F[U] \quad \text{STAR} \\
F[U : A \, G \, p \, H & \, \triangleright \to B] \longrightarrow \text{blame } p \quad \text{CONFLICT} \\
F[(M : A \, \triangleright \to \, B) : B \, \triangleright \to C] & \longrightarrow F[(M : A \, \triangleright \to \, C)] \quad \text{COMPOSE} \\
\end{align*}
\]

Safe threesome

\[
\begin{align*}
K \, \triangleright \, q & \, | \, s \, \text{safe } q & \, | \, t \, \text{safe } q & \, | \, \ast \, \text{safe } q \\
(g!) \, \triangleright \, q & \, | \, s \, \text{safe } q & \, | \, i \, \text{safe } q & \, | \, p \neq q & \, | \, p \neq q \\
\end{align*}
\]

Figure 7. Threesome calculus (\( \lambda T \))

To indicate that the coercion on the left translates to the threesome coercion on the right. We write

\[
| A \, \triangleright \to B | \triangleright \to C = c
\]

to indicate that the threesome on the left translates to the coercion on the right. The translation requires source and target types, as well as \( G \) and \( H \), in order to correctly generate the coercion corresponding to a threesome of the form \( G \, p \, H \). Both translations extend to terms in the obvious way.

The translation from \( \lambda C \) to \( \lambda T \) preserves type and blame safety, as does the reverse translation.

Proposition 19 (Coercions preserve threesomes, and vice versa).
Coercions to threesome (λC to λT)
\[ \text{|c|}^{\text{CT}} = t \]
\[ |\text{id}(A)|^{\text{CT}} = A \]
\[ |p ? G|^{\text{CT}} = p ? G \]
\[ |G!|^{\text{CT}} = G! \]
\[ |s \rightarrow t|^{\text{CT}} = |s|^{\text{CT}} \rightarrow |t|^{\text{CT}} \]
\[ |s ; t|^{\text{CT}} = |s|^{\text{CT}} ; |t|^{\text{CT}} \]

Threesome to coercions (λT to λC)
\[ M \approx \text{|c|}^{\text{CT}} = c \]
\[ |K \overset{K}{\rightarrow} K|^{\text{TC}} = \text{id}(K) \]
\[ |s \overset{x}{\rightarrow} s|^{\text{TC}} = \text{id}(x) \]
\[ |A \rightarrow B \overset{s}{\rightarrow} A' \rightarrow B'|^{\text{TC}} = |A|^{\overset{s}{\rightarrow}} A^{\text{TC}} \rightarrow |B|^{\overset{t}{\rightarrow}} B'^{\text{TC}} \]
\[ |A|^{\overset{g}{\rightarrow}} |s|^{\text{TC}} = |A|^{\overset{g}{\rightarrow}} |G|^{\text{TC}} ; (G!) \]
\[ |p \overset{p}{\rightarrow} A|^{\text{TC}} = (p ? G) ; |G|^{\overset{t}{\rightarrow}} A^{\text{TC}} \]
\[ |G \overset{p H}{\rightarrow} B|^{\text{TC}} = c ; (G!) ; (p ? H) ; d \]
where \( c : A \overset{=}{} G \) and \( d : H \overset{=}{} B \)

Bisimulation between λC and λT
\[ M \cong M' \]

Figure 8. Relating coercion and threesome calculi (λC and λT)

- If \( \Gamma \vdash_C M : A \) then \( \Gamma \vdash_T |M|^{\text{CT}} : A \).
- If \( \Gamma \vdash_T M' : A \) then \( \Gamma \vdash_C |M'|^{\text{CT}} : A \).
- If \( M \overset{\text{safe}_C}{\rightarrow} p \) then \( |M|^{\text{CT}} \overset{\text{safe}_T}{\rightarrow} p \).
- If \( M' \overset{\text{safe}_T}{\rightarrow} p \) then \( |M'|^{\text{CT}} \overset{\text{safe}_C}{\rightarrow} p \).

The dynamics of the threesome calculus differ significantly from that of the other three calculi because it compresses sequences of casts into a single threesome. We define a bisimulation \( \approx \) that relates the coercion calculus and the threesome calculus. The rules relate a sequence of coercions to a single threesome. Rule (1) relates an empty sequence of coercions to the identity threesome. Rule (2) relates multiple coercions to a single threesome which is their composition. Rule (3) relates function coercions to function threesome. On the coercion side, there is one reduction via (WRAP) for each cast wrapped around the function, while on the threesome side, there is just a single reduction via (WRAP).

The translation from λC to λT is a subset of the relation \( \cong \), as is the reverse translation.

Proposition 20 (Translations are a subset of \( \cong \)).
\[ M \cong |M|^{\text{CT}} \text{ and } |M'|^{\text{TC}} \cong M' \]

The \( \cong \) relation is a bisimulation.

Proposition 21 (Threesome bisimulate coercions).
- If \( M \cong M' \) and both \( M \) and \( M' \) are well typed, then
  - If \( M \overset{\cong}{\rightarrow} N \text{ then } M' \overset{\cong}{\rightarrow} N' \text{ and } N \cong N' \text{ for some } N' \).
  - If \( M \overset{\cong}{\rightarrow} \text{blame } p \text{ then } M' \overset{\cong}{\rightarrow} \text{blame } p \).
  - If \( M' \overset{\cong}{\rightarrow} N' \text{ then } M \overset{\cong}{\rightarrow} \text{blame } p \text{ and } N \cong N' \text{ for some } N' \).
  - If \( M' \overset{\cong}{\rightarrow} \text{blame } p \text{ then } M \overset{\cong}{\rightarrow} \text{blame } p \).

The bisimulation is not on the nose: a single step in threesome corresponds to zero or more steps in coercions. The proof follows the same lines as that of Lemma 8 of Siek and Wadler (2010).

7. Related Work

Languages that integrate dynamic and static types include C# (Bieneman et al. 2010), Dart (Bracha and Bak 2011), Pyret (Patterson et al. 2014), Racket (Flatt and PLT 2014), TypeScript (Heiltsburg 2012), and VB (Meijer and Drayton 2004).

Barron et al. (1963) include a notion of dynamic type, called general, in the otherwise statically-typed CPL. Liskov et al. (1979) include an any type in CLU. Abadi et al. (1991) discuss languages with typecase and formalize its semantics.


Greenberg et al. (2010) study dependent contracts and the translation between latent and manifest systems. Greenberg (2013) considers calculi CAST, NAIVE, and EFFICIENT, similar to our λB, λC, and λT; unlike our work, he includes refinement types, but omits blame.

Benton (2008) introduces ‘undoable’ cast operators, to enable a failed cast to report an error at a more convenient location. Swamy et al. (2014) present a secure embedding of the gradually typed TS’ language into JavaScript.


8. Conclusion

We have defined translations between foundational calculi for blame, coercions, and threesome, and established tight connections between them: type and blame safety are preserved, and there are mutual simulations between the calculi. We introduced the precise blame calculus, a novel calculus that allocates blame to specific
parts of a cast; and we presented a revision of the threesome calculus, where the connection to coercions is rendered transparent. We look forward to designs that exploit these foundations in a practical setting, and experiments to evaluate their utility.

Acknowledgments Thanks to Shayan Najd and Michael Greenberg for comments on an earlier draft. Siek is supported by NSF Grant 1360694.

References


