Blame and Coercion: Together Again for the First Time

Citation for published version:
Siek, J, Thiemann, P & Wadler, P 2015, Blame and Coercion: Together Again for the First Time. in *36th annual ACM SIGPLAN conference on Programming Language Design and Implementation: Portland OR 13-17 June 2015.* ACM, pp. 425-435. https://doi.org/10.1145/2737924.2737968

Digital Object Identifier (DOI):
10.1145/2737924.2737968

Published In:
36th annual ACM SIGPLAN conference on Programming Language Design and Implementation

Blame and Coercion: Together Again for the First Time

Jeremy Siek
Indiana University, USA
jsiek@indiana.edu

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

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

Abstract

C#, Dart, Pyret, Racket, TypeScript, VB: many recent languages integrate dynamic and static types via gradual typing. We systematically develop three calculi for gradual typing and the relations between them, building on and strengthening previous work. The calculi are: $\lambda \mathsf{B}$, based on the blame calculus of Wadler and Findler (2009); $\lambda \mathsf{C}$, inspired by the coercion calculus of Henglein (1994); $\lambda \mathsf{S}$ inspired by the space-efficient calculus of Herman, Tomb, and Flanagan (2006) and the threesome calculus of Siek and Wadler (2010). While $\lambda \mathsf{B}$ is little changed from previous work, $\lambda \mathsf{C}$ and $\lambda \mathsf{S}$ are new. Together, $\lambda \mathsf{B}$, $\lambda \mathsf{C}$, and $\lambda \mathsf{S}$ provide a coherent foundation for design, implementation, and optimisation of gradual types.

We define translations from $\lambda \mathsf{B}$ to $\lambda \mathsf{C}$ and from $\lambda \mathsf{C}$ to $\lambda \mathsf{S}$. Much previous work lacked proofs of correctness or had weak correctness criteria; here we demonstrate the strongest correctness criterion one could hope for, that each of the translations is fully abstract. Each of the calculi reinforces the design of the others: $\lambda \mathsf{C}$ has a particularly simple definition, and the subtle definition of blame safety for $\lambda \mathsf{B}$ is justified by the simple definition of blame safety for $\lambda \mathsf{C}$. Our calculus $\lambda \mathsf{S}$ is implementation-ready: the first space-efficient calculus that is both straightforward to implement and easy to understand. We give two applications: first, using full abstraction from $\lambda \mathsf{C}$ to $\lambda \mathsf{S}$ to validate the challenging part of full abstraction between $\lambda \mathsf{B}$ and $\lambda \mathsf{C}$; and, second, using full abstraction from $\lambda \mathsf{B}$ to $\lambda \mathsf{S}$ to easily establish the Fundamental Property of Casts, which required a custom bisimulation and six lemmas in earlier work.

Categories and Subject Descriptors F.3.3 [*Logics and meaning of programs*]: Studies of Program Constructs—Type structure

Keywords Blame, lambda calculus, gradual typing

1. Introduction

Contracts and blame. Findler and Felleisen (2002) introduced two seminal ideas: higher-order *contracts* to monitor adherence to a specification, and *blame* to indicate which of two parties is at fault if the contract is violated. In particular, at higher-order a contract allocates blame to the environment if it supplies an incorrect argument or to the function if it supplies an incorrect result. Blame characterises correctness: one cannot guarantee that a contract interposed between typed and untyped code will not be violated, but one can guarantee that if it is violated then blame allocates to the untyped code, a result first established by Tobin-Hochstadt and Felleisen (2006).

Findler and Felleisen's innovation led to a bloom of others. Siek and Taha (2006) introduced gradual typing; Flanagan (2006) introduced hybrid typing, later implemented in Sage (Gronski et al. 2006); Ou et al. (2004) integrated simple and dependent types. These systems built crucially on contracts, and all used a similar translation from a source language to an intermediate language of explicit casts. Alas, they ignored blame. Wadler and Findler (2009) restored blame to this intermediate language and formalised it as as the *blame calculus*. They established *blame saftey*, a generalisation of the correctness criterion for contracts: given a cast between a less-precise and a more-precise type, blame always allocates to the less-precisely typed side of the cast—"Well-typed programs can't be blamed".

Space-efficient coercions. A naive implementation of contracts (or the blame calculus) suffers space leaks. 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 requires space proportional to the number of calls.

Herman et al. (2007, 2010) proposed a solution to this problem based on the coercion calculus of Henglein (1994). Alas, they also ignoring blame. Their calculus represents casts as coercions. When two coercions are applied in sequence, they are composed and normalised. The height of the composition of two coercions is bounded by the heights of the two original coercions; the size of a coercion in normalised form is bounded if its height is bounded, ensuring that computation proceeds in bounded space. However, normalising coercions requires that sequences of compositions are treated as equal up to associativity. While this is not a difficult problem in symbol manipulation, it does pose a challenge when implementing an efficient evaluator.

Siek and Wadler (2010) proposed an alternative solution. At first, they also ignored blame. They observed that any cast factors into a downcast from the source to a mediating type, followed by an upcast from the mediating type to the target—called a *threesome* because it involves three types. Two successive threesomes collapse to a single threesome, where the mediating type is the greatest lower bound of the two original mediating types. The height of the greatest lower bound of two types is bounded by their heights; and the size of a type is bounded if its height is bounded, again ensuring that computation proceeds in bounded space.

Siek and Wadler (2010) then restored blame by decorating the mediating type with labels that indicate how blame is to be allo-

PLDI'15, June 13–17, 2015, Portland, OR, USA.

http://dx.doi.org/10.1145/—

cated, and showed decorated types are in one-to-one correspondence with normalised coercions. A recursive definition computes the meet of the two decorated types (or equivalently the composition of the two corresponding coercions); it is straightforward to calculate, avoiding the associativity problem of coercions.

However, the notation for decorated types is far from transparent. Siek reports that Tanter attempted to implement Gradualtalk with threesomes, but found it too difficult. Wadler reports that while preparing a lecture on threesomes a few years after the paper was published, he required several hours to puzzle out the meaning of his own notation, $\bot^{mGp}$. Eventually, he could only understand it by relating it to the corresponding coercion—a hint that coercions may be clearer than threesomes once blame is involved.

Hence we have two approaches: Herman et al. (2007, 2010) is easy to understand, but hard to compute; Siek and Wadler (2010) is easy to compute, but hard to understand. Garcia (2013) attempted to ameliorate this tension by starting with the former and deriving the latter. However, the derivation necessarily contains all the confusing notation of Siek and Wadler while also introducing additional notations of its own, notably, a collection of ten supercoercions. By design, his derived definition of composition matches Siek and Wadler's original and so is no easier to read.

Much previous work lacked proofs of correctness or had weak correctness criteria. Herman et al. (2007, 2010) give no proof relating their calculus to others for gradual typing. Siek and Wadler (2010) establish that a term in the blame calculus converges if and only if its translation into the threesome calculus converges, but they do so only at the top level (Kleene equivalence: roughly, contextual equivalence without the context).

Our approach. We establish new foundations for gradual typing by considering a sequence of calculi and the relations between them: $\lambda\mathsf{B}$, based on the blame calculus of Wadler and Findler (2009); $\lambda\mathsf{C}$, inspired by the coercion calculus of Henglein (1994); $\lambda\mathsf{S}$, inspired by the space-efficient calculus of Herman et al. (2007, 2010) and the threesome calculus of Siek and Wadler (2010). While $\lambda\mathsf{B}$ is little changed from previous work, the other two are new.

The two new calculi are based on ideas so simple it is surprising no one thought of them years ago. For $\lambda\mathsf{C}$, the novel insight is to present a computational calculus as close as possible to the original coercion calculus of Henglein (1994). For $\lambda\mathsf{S}$, the novel insight is to restrict coercions to a canonical form and write out the algorithm that composes two canonical coercions to yield a canonical coercion.

Henglein (1994) explored optimisation of coercions, but remarkably neither he nor anyone else has written down the obvious reduction rules for evaluating a lambda calculus with coercions, as we have done here with $\lambda\mathsf{C}$. The result is a pleasingly simple calculus, close to correct by construction.

Our translation from $\lambda\mathsf{B}$ into $\lambda\mathsf{C}$ resembles many in the literature; it compiles casts into coercions. We show that this translation is a lockstep bisimulation, where a single reduction step in $\lambda\mathsf{B}$ corresponds to a single reduction step in $\lambda\mathsf{C}$, giving a close correspondence between the two calculi. There are several subtleties in the design of $\lambda\mathsf{B}$, but essentially none in the design of $\lambda\mathsf{C}$, and that the two run in lockstep suggests that both designs are correct.

A key property of the blame calculus is blame safety—"Well-typed programs can't be blamed". Surprisingly, no previous work considers whether translations preserve blame safety. Here we show that blame safety is preserved by translations between calculi, and, as a pleasant consequence, that the subtle definition of blame safety for $\lambda\mathsf{B}$ is justified by the straightforward definition of blame safety for $\lambda\mathsf{C}$.

Our reverse translation from $\lambda\mathsf{C}$ to $\lambda\mathsf{B}$ is novel. We observe that a single coercion must translate into a sequence of casts, because a coercion may contain many blame labels but a cast contains only one. The challenge is to show that translating from $\lambda\mathsf{C}$ to $\lambda\mathsf{B}$ and back again yields a term contextually equivalent to the original. This, together with the bisimulation, establishes the strongest correctness criterion one could hope for, full abstraction: translation from $\lambda\mathsf{B}$ to $\lambda\mathsf{C}$ preserves and reflects contextual equivalence.

For $\lambda\mathsf{S}$ we isolate a novel grammar corresponding to coercions in canonical form. Canonical forms are unique, and in one-to-one correspondence with normal forms. We present a simple recursive function that takes two coercions in canonical form, s and t, and returns their composition in canonical form, $s \mathbin{⨟} t$. Validating the correctness of this definition against Henglein's original rules is straightforward. As with threesomes, it avoids the problems of associativity previously attached to using coercions; but because it is based on coercions, it avoids the problems of decoding the meaning of the decorated types attached to threesomes.

Translation from $\lambda\mathsf{C}$ to $\lambda\mathsf{S}$ is straightforward, but establishing its correctness is the most challenging result in the paper. The difficulty is that $\lambda\mathsf{C}$ breaks compositions into simpler components,

$$M\langle c\,;d\rangle \longrightarrow M\langle c\rangle\langle d\rangle,$$

while $\lambda\mathsf{S}$ assembles simpler components into compositions,

$$M\langle s\rangle\langle t\rangle \longrightarrow M\langle s \mathbin{⨟} t\rangle.$$

(As explained in Sections 3 and 4, c, d range over coercions and s, t over space-efficient coercions, $M\langle c\rangle$ denotes the application to term M of coercion c, and similarly for $M\langle s\rangle$.) We introduce a relation between terms of $\lambda\mathsf{C}$ and $\lambda\mathsf{S}$ and show it is a bisimulation. In this case the bisimulation is not lockstep: one step in $\lambda\mathsf{C}$ may correspond to many in $\lambda\mathsf{S}$, and vice-versa. Siek and Wadler (2010) establish a bisimulation similar to the one here, but our development is simpler because it uses coercions rather than decorated types, and because it uses $\lambda\mathsf{C}$ as an intermediate step. Because the mapping of λS back to λC is simply an inclusion, the bisimulation easily establishes full abstraction of the translation from $\lambda\mathsf{C}$ to $\lambda\mathsf{S}$.

Outline. Sections 2–4 systematically consider $\lambda\mathsf{B}$, $\lambda\mathsf{C}$, and $\lambda\mathsf{S}$. For each caclulus we introduce its syntax, type rules, and reduction rules; and we establish type safety and blame safety. In Sections 3–4, for each calculus we also consider translations to and from the previous calculus, show the translations preserve type and blame safety, and demonstrate a bisimulation and full abstraction.

In Section 5, we observe that full abstraction often makes it easy to establish equivalences in $\lambda\mathsf{B}$ or $\lambda\mathsf{C}$, because equivalent terms in those calculi translate into one and the same term in $\lambda\mathsf{S}$. In particular, we exploit full abstraction between $\lambda\mathsf{C}$ and $\lambda\mathsf{S}$ to establish the key lemma required to show full abstraction between $\lambda\mathsf{B}$ and $\lambda\mathsf{C}$. We also exploit full abstraction between $\lambda\mathsf{B}$ and $\lambda\mathsf{S}$ to establish The Fundamental Theorem of Casts, which required a custom bisimulation and six lemmas in Siek and Wadler (2010). Section 6 discusses related work, including a survey of how gradual typing is used in practice. Section 7 concludes.

2. Blame Calculus

Figure 1 defines the blame calculus, $\lambda\mathsf{B}$. This section reprises results from Wadler and Findler (2009), Siek and Wadler (2010), and Ahmed et al. (2011). Additional motivation and examples can be found in Wadler (2015).

Blame calculus is based on simply-typed lambda calculus, standard constructs of which are shown in gray. Let A, B, C range over types. A type is either a base type ι, a function type $A \rightarrow B$, or the dynamic type $\star$. Let G, H range over ground types. A ground type is either a base type ι or the function type $\star \rightarrow \star$. The dynamic type satisfies the domain equation

$$\star \cong \iota + (\star \rightarrow \star)$$

so each value of dynamic type belongs to one ground type.

Syntax

$$A, B, C ::= \iota \mid A \to B \mid \star$$

$$G, H ::= \iota \mid \star \to \star$$

$$L, M, N ::= k \mid op(\vec{M}) \mid x \mid \lambda x{:}A.\, N \mid L\ M \mid M : A \stackrel{p}{\Longrightarrow} B \mid \texttt{blame}\ p$$

$$V, W ::= k \mid \lambda x{:}A.\, N \mid V : A \to B \stackrel{p}{\Longrightarrow} A' \to B' \mid V : G \stackrel{p}{\Longrightarrow} \star$$

$$\mathcal{E} ::= \square \mid \mathcal{E}[op(\vec{V}, \square, \vec{M})] \mid \mathcal{E}[\square\ M] \mid \mathcal{E}[V\ \square] \mid \mathcal{E}[\square : A \stackrel{p}{\Longrightarrow} B]$$

Compatible $\boxed{A \sim B}$

$$\frac{}{\iota \sim \iota} \qquad \frac{A \sim A' \quad B \sim B'}{A \to B \sim A' \to B'} \qquad \frac{}{A \sim \star} \qquad \frac{}{\star \sim B}$$

Term typing $\boxed{\Gamma \vdash_{\mathsf{B}} M : A}$

$$\frac{\Gamma \vdash M : A \quad A \sim B}{\Gamma \vdash (M : A \stackrel{p}{\Longrightarrow} B) : B} \qquad \frac{}{\Gamma \vdash \texttt{blame}\ p : A}$$

Reduction $\boxed{M \longrightarrow_{\mathsf{B}} N}$

$$\mathcal{E}[op(\vec{V})] \longrightarrow \mathcal{E}[[\![op]\!](\vec{V})]$$

$$\mathcal{E}[(\lambda x{:}A.\, N)\ V] \longrightarrow \mathcal{E}[N[x{:=}V]]$$

$$\mathcal{E}[V : \iota \stackrel{p}{\Longrightarrow} \iota] \longrightarrow \mathcal{E}[V]$$

$$\mathcal{E}[(V : A \to B \stackrel{p}{\Longrightarrow} A' \to B')\ W] \longrightarrow \mathcal{E}[(V\ (W : A' \stackrel{\overline{p}}{\Longrightarrow} A)) : B \stackrel{p}{\Longrightarrow} B']$$

$$\mathcal{E}[V : \star \stackrel{p}{\Longrightarrow} \star] \longrightarrow \mathcal{E}[V]$$

$$\mathcal{E}[V : A \stackrel{p}{\Longrightarrow} \star] \longrightarrow \mathcal{E}[V : A \stackrel{p}{\Longrightarrow} G \stackrel{p}{\Longrightarrow} \star] \quad \text{if } A \neq \star, A \neq G, A \sim G$$

$$\mathcal{E}[V : \star \stackrel{p}{\Longrightarrow} A] \longrightarrow \mathcal{E}[V : \star \stackrel{p}{\Longrightarrow} G \stackrel{p}{\Longrightarrow} A] \quad \text{if } A \neq \star, A \neq G, A \sim G$$

$$\mathcal{E}[V : G \stackrel{p}{\Longrightarrow} \star \stackrel{q}{\Longrightarrow} G] \longrightarrow \mathcal{E}[V]$$

$$\mathcal{E}[V : G \stackrel{p}{\Longrightarrow} \star \stackrel{q}{\Longrightarrow} H] \longrightarrow \texttt{blame}\ q \quad \text{if } G \neq H$$

$$\mathcal{E}[\texttt{blame}\ p] \longrightarrow \texttt{blame}\ p \quad \text{if } \mathcal{E} \neq \square$$

Embedding dynamically typed λ-calculus $\boxed{\lceil M \rceil}$

$$\lceil k \rceil = k : \iota \stackrel{p}{\Longrightarrow} \star \quad \text{if } k : \iota$$

$$\lceil op(\vec{M}) \rceil = op(\lceil \vec{M} \rceil : \vec{\star} \stackrel{\vec{p}}{\Longrightarrow} \vec{\iota}) : \iota \stackrel{p}{\Longrightarrow} \star \quad \text{if } op : \vec{\iota} \to \iota$$

$$\lceil x \rceil = x$$

$$\lceil \lambda x.\, N \rceil = (\lambda x{:}\star.\, \lceil N \rceil) : \star \to \star \stackrel{p}{\Longrightarrow} \star$$

$$\lceil L\ M \rceil = (\lceil L \rceil : \star \stackrel{p}{\Longrightarrow} \star \to \star)\ \lceil M \rceil$$

Figure 1. Blame calculus (λB)

Types A and B are compatible, written $A \sim B$, if either is the dynamic type, if they are both the same base type, or they are both function types with compatible domains and ranges. Every type is either the dynamic type or compatible with a unique ground type. Two ground types are compatible if and only if they are equal.

Lemma 1 (Grounding)**.**

1. *If $A \neq \star$, there is a unique G such that $A \sim G$.*
2. *$G \sim H$ iff $G = H$.*

Incompatibility is the source of all blame: casting a type into the dynamic type and then casting out at an incompatible type allocates blame to the second cast.

Let p, q range over blame labels. To indicate on which side of a cast blame lays, each blame label p has a complement $\overline{p}$. Complement is involutive, $\overline{\overline{p}} = p$.

Let L, M, N range over terms. Terms are those of simply-typed lambda calculus, plus casts and blame. Each operator op on base types is specified by a total meaning function $[\![op]\!]$ that preserves types: if $op : \vec{\iota} \to \iota$ and $\vec{k} : \vec{\iota}$, then $[\![op]\!](\vec{k}) = k$ with $k : \iota$.

Typing, reduction, and safety judgments are written with subscripts indicating to which calculus they belong, except we omit subscripts in figures to avoid clutter. We write $\Gamma \vdash_{\mathsf{B}} M : A$ to indicate that in type environment Γ term M has type A. Type rules for simply-typed lambda calculus are standard and omitted. The type rule for casts is straightforward:

$$\frac{\Gamma \vdash_{\mathsf{B}} M \quad A \sim B}{\Gamma \vdash_{\mathsf{B}} (M : A \stackrel{p}{\Longrightarrow} B) : B}$$

If term M has type A and types A and B are compatible then a cast of M from A to B is a term of type B. The cast is decorated with a blame label p. We abbreviate a pair of casts

$$(M : A \stackrel{p}{\Longrightarrow} B) : B \stackrel{q}{\Longrightarrow} C \quad \text{as} \quad M : A \stackrel{p}{\Longrightarrow} B \stackrel{q}{\Longrightarrow} C.$$

A term $\texttt{blame}\ p$ has any type.

Every well-typed term not containing blame has a unique type: if $\Gamma \vdash M : A$ and $\Gamma \vdash M : A'$ and M does not contain a subterm of the form $\texttt{blame}\ p$, then $A = A'$.

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.

Let V, W range over values. A value is a constant, a lambda abstraction, a cast of a value from function type to function type, or a cast of a value from ground type to dynamic type. Let $\mathcal{E}$ range over evaluation contexts, which are standard, and include casts in the obvious way. We write $M \longrightarrow_{\mathsf{B}} N$ to indicate that term M steps to term N. For any reduction relation $\longrightarrow$, we write its reflexive and transitive closure as $\longrightarrow^*$.

The first two rules are standard (and not repeated in subsequent figures). A cast from a base type to itself leaves the value unchanged. 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; to allocate blame correctly, the blame label 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). A cast from type $\star$ to itself leaves the value unchanged. Assume A is neither the dynamic type $\star$ nor a ground type, and G is the unique ground type compatible with A; then a cast from A to $\star$ factors into a cast from A to G followed by a cast from G to $\star$, and a cast from $\star$ to A factors into a cast from $\star$ to G followed by a cast from G to A. A cast from a ground type G to type $\star$ and back to the same ground type G leaves the value unchanged. A cast from a ground type G to type $\star$ and back to an incompatible ground type H allocates blame to the label of the outer cast. (Why the outer cast? This choice traces back to Findler and Felleisen (2002), and reflects the idea that we always hold an injection from ground type to dynamic type blameless, but may allocate blame to a projection from dynamic type to ground type.)

Two rules have side conditions $A \neq \star, A \neq G, A \sim G$. The condition implies that $G = \star \to \star$, so we could rewrite the rules replacing G by $\star \to \star$. We use the given form because it is more

Subtype $\boxed{A <: B}$

$$\frac{}{\iota <: \iota} \qquad \frac{A' <: A \quad B <: B'}{A \to B <: A' \to B'} \qquad \frac{A <: G}{A <: \star}$$

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

$$\frac{}{\iota <:^+ \iota} \qquad \frac{A' <:^- A \quad B <:^+ B'}{A \to B <:^+ A' \to B'} \qquad \frac{}{A <:^+ \star}$$

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

$$\frac{}{\iota <:^- \iota} \qquad \frac{A' <:^+ A \quad B <:^- B'}{A \to B <:^- A' \to B'} \qquad \frac{}{\star <:^- B}$$

$$\frac{A <:^- G}{A <:^- \star}$$

Naive subtype $\boxed{A <:_n B}$

$$\frac{}{\iota <:_n \iota} \qquad \frac{A <:_n A' \quad B <:_n B'}{A \to B <:_n A' \to B'} \qquad \frac{}{A <:_n \star}$$

Safe cast $\boxed{(A \stackrel{p}{\Longrightarrow} B)\ \mathsf{safe}_{\mathsf{B}}\ q}$

$$\frac{A <:^+ B}{(A \stackrel{p}{\Longrightarrow} B)\ \mathsf{safe}\ p} \qquad \frac{A <:^- B}{(A \stackrel{p}{\Longrightarrow} B)\ \mathsf{safe}\ \overline{p}}$$

$$\frac{p \neq q \quad \overline{p} \neq q}{(A \stackrel{p}{\Longrightarrow} B)\ \mathsf{safe}\ q}$$

Figure 2. Subtyping and blame safety

compact, and it adapts if we permit other ground types, such as product $G = \star \times \star$.

The following lemma will prove useful later.

Lemma 2 (Failure). *If $A \neq \star$, $A \sim G$, and $G \neq H$, then*

$$V : A \stackrel{p_1}{\Longrightarrow} G \stackrel{p_2}{\Longrightarrow} \star \stackrel{p_3}{\Longrightarrow} H \stackrel{p_4}{\Longrightarrow} B \longrightarrow^* \mathtt{blame}\ p_3$$

Embedding $\lceil M \rceil$ takes terms of dynamically-typed lambda calculus into the blame calculus. The embedding introduces a fresh label p for each cast.

Type safety is established via preservation and progress.

Proposition 3 (Type safety, Wadler and Findler (2009)).

1. *If $\vdash_{\mathsf{B}} M : A$ and $M \longrightarrow_{\mathsf{B}} N$ then $\vdash_{\mathsf{B}} N : A$.*
2. *If $\vdash_{\mathsf{B}} M : A$ then either*
   (a) *there exists a term N such that $M \longrightarrow_{\mathsf{B}} N$, or*
   (b) *there exists a value V such that $M = V$, or*
   (c) *there exists a label p such that $M = \mathtt{blame}\ p$.*

The same will hold, mutatis mutandis, for $\lambda\mathsf{C}$ and $\lambda\mathsf{S}$.

Type safety does not rule out blame as a result. How to guarantee blame cannot arise in certain circumstances is the subject of the next section.

2.1 Blame Safety

Figure 2 presents four different subtyping relations and defines safety for blame calculus.

Why do we need *four* different subtyping relations? Each has a different purpose. Relation $A <: B$ characterizes when a cast $A \Longrightarrow B$ *never* yields blame; relations $A <:^+ B$ and $A <:^- B$ characterize when a cast $A \Longrightarrow B$ cannot yield *positive* or *negative* blame, respectively; and relation $A <:_n B$ characterizes when type A is more *precise* than type B. All four relations are reflexive and transitive, and subtyping, positive subtyping, and naive subtyping are antisymmetric.

The first three subtyping relations are characterised by *contravariance*. A cast from a base type to itself never yields blame. A cast from a function type to a function type never yields positive blame if the cast of the arguments never yields negative blame and if the cast of the results never yields positive blame; and ditto with positive and negative reversed; as with casts, each rule is contravariant in the function domain and covariant in the function range. A cast from ground type to dynamic type never yields blame. A cast to dynamic type never yields positive blame, while a cast from dynamic type never yields negative blame.

Naive subtyping is characterised by *covariance*. A base type is as precise as itself, precision of function types is covariant in *both* the domain and range of functions, and the dynamic type is the least precise type.

These four relations are closely connected: ordinary subtyping decomposes into positive and negative subtyping, which can be reassembled to yield naive subtyping, almost like a tangram.

Lemma 4 (Tangram, Wadler and Findler (2009)).

1. *$A <: B$ iff $A <:^+ B$ and $A <:^- B$.*
2. *$A <:_n B$ iff $A <:^+ B$ and $B <:^- A$.*

A cast from A to B decorated with p is *safe* for blame label q,

$$(A \stackrel{p}{\Longrightarrow} B)\ \mathsf{safe}_{\mathsf{B}}\ q,$$

if evaluation of the cast can never allocates blame to 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 label p never allocates blame other than to p or $\overline{p}$. Safety extends to terms in the obvious way: $M\ \mathsf{safe}_{\mathsf{B}}\ q$ if every cast in M is safe for q. Blame safety is established via a variant of preservation and progress.

Proposition 5 (Blame safety, Wadler and Findler (2009)).

1. *If $M\ \mathsf{safe}_{\mathsf{B}}\ q$ and $M \longrightarrow_{\mathsf{B}} N$ then $N\ \mathsf{safe}_{\mathsf{B}}\ q$.*
2. *If $M\ \mathsf{safe}_{\mathsf{B}}\ q$ then $M \not\longrightarrow_{\mathsf{B}} \mathtt{blame}\ q$.*

The same will hold, mutatis mutandis, for $\lambda\mathsf{C}$ and $\lambda\mathsf{S}$.

2.2 Contextual Equivalence

Contextual equivalence is defined as usual. Evaluating a term may have three outcomes: converge, allocate blame to p, or diverge. Two terms are contextually equivalent if they have the same outcome in any context.

Let $\mathcal{C}$ range over contexts. A context is an expression with a single hole in any position. Write $M{\uparrow}_{\mathsf{B}}$ if M diverges; coinductively, $M{\uparrow}_{\mathsf{B}}$ if $M \longrightarrow_{\mathsf{B}} N$ and $N{\uparrow}_{\mathsf{B}}$.

Definition 6 (Contextual equivalence). *Two terms are contextually equivalent, $M \stackrel{\mathsf{ctx}}{=}_{\mathsf{B}} N$, if for any context $\mathcal{C}$, either*

1. *both converge, $\mathcal{C}[M] \longrightarrow^*_{\mathsf{B}} V$ and $\mathcal{C}[N] \longrightarrow^*_{\mathsf{B}} W$, for some values V and W.*
2. *both blame the same label, $\mathcal{C}[M] \longrightarrow^*_{\mathsf{B}} \mathtt{blame} p$ and $\mathcal{C}[N] \longrightarrow^*_{B} \mathtt{blame}\ p$, for some label p, or*
3. *both diverge, $\mathcal{C}[M]{\uparrow}_{\mathsf{B}}$ and $\mathcal{C}[N]{\uparrow}_{\mathsf{B}}$.*

The same will apply, mutatis mutandis, for $\lambda\mathsf{C}$ and $\lambda\mathsf{S}$.

3. Coercion Calculus

Figure 3 defines the coercion calculus, $\lambda\mathsf{C}$. Our coercions correspond to those of Henglein (1994), except that a coercion from dynamic type to ground type is decorated with a blame label, as in Siek and Wadler (2010), and we add a coercion $\perp^{GpH}$, similar to

Syntax

$$
\begin{aligned}
c, d &::= \mathtt{id}_A \mid G\,\mathtt{!} \mid G\mathtt{?}^p \mid c \to d \mid c \mathbin{;} d \mid \bot^{GpH} \\
L, M, N &::= k \mid op(\vec{M}) \mid x \mid \lambda x{:}A.\, N \mid L\ M \mid M\langle c\rangle \mid \mathtt{blame}\ p \\
V, W &::= k \mid \lambda x{:}A.\, N \mid V\langle c \to d\rangle \mid V\langle G\,\mathtt{!}\rangle \\
\mathcal{E} &::= \square \mid \mathcal{E}[op(\vec{V}, \square, \vec{M})] \mid \mathcal{E}[\square\ M] \mid \mathcal{E}[V\ \square] \mid \mathcal{E}[\square\langle c\rangle]
\end{aligned}
$$

Coercion typing $\boxed{c : A \Longrightarrow B}$

$$\frac{}{\mathtt{id}_A : A \Longrightarrow A}$$

$$\frac{}{G\,\mathtt{!} : G \Longrightarrow \star} \qquad \frac{}{G\mathtt{?}^p : \star \Longrightarrow G}$$

$$\frac{c : A' \Longrightarrow A \qquad d : B \Longrightarrow B'}{(c \to d) : A \to B \Longrightarrow A' \to B'}$$

$$\frac{c : A \Longrightarrow B \qquad d : B \Longrightarrow C}{(c \mathbin{;} d) : A \Longrightarrow C}$$

$$\frac{A \neq \star \qquad A \sim G \qquad G \neq H}{\bot^{GpH} : A \Longrightarrow B}$$

Term typing $\boxed{\Gamma \vdash_{\mathsf{C}} M : A}$

$$\frac{\Gamma \vdash M : A \qquad c : A \Longrightarrow B}{\Gamma \vdash M\langle c\rangle : B} \qquad \frac{}{\Gamma \vdash \mathtt{blame}\ p : A}$$

Reduction $\boxed{M \longrightarrow_{\mathsf{C}} N}$

$$
\begin{array}{rcll}
\mathcal{E}[V\langle \mathtt{id}_A\rangle] & \longrightarrow & \mathcal{E}[V] & \\
\mathcal{E}[(V\langle c \to d\rangle)\ W] & \longrightarrow & \mathcal{E}[(V\ (W\langle c\rangle))\langle d\rangle] & \\
\mathcal{E}[V\langle G\,\mathtt{!}\rangle\langle G\mathtt{?}^p\rangle] & \longrightarrow & \mathcal{E}[V] & \\
\mathcal{E}[V\langle G\,\mathtt{!}\rangle\langle H\mathtt{?}^p\rangle] & \longrightarrow & \mathtt{blame}\ p & \text{if } G \neq H \\
\mathcal{E}[V\langle c \mathbin{;} d\rangle] & \longrightarrow & \mathcal{E}[V\langle c\rangle\langle d\rangle] & \\
\mathcal{E}[V\langle \bot^{GpH}\rangle] & \longrightarrow & \mathtt{blame}\ p & \\
\mathcal{E}[\mathtt{blame}\ p] & \longrightarrow & \mathtt{blame}\ p & \text{if } \mathcal{E} \neq \square
\end{array}
$$

Safe coercion $\boxed{c \ \mathsf{safe}_{\mathsf{C}}\ q}$

$$\frac{}{\mathtt{id}_A \ \mathsf{safe}\ q} \qquad \frac{}{G\,\mathtt{!}\ \mathsf{safe}\ q} \qquad \frac{p \neq q}{G\mathtt{?}^p\ \mathsf{safe}\ q}$$

$$\frac{c\ \mathsf{safe}\ q \quad d\ \mathsf{safe}\ q}{c \to d\ \mathsf{safe}\ q} \qquad \frac{c\ \mathsf{safe}\ q \quad d\ \mathsf{safe}\ q}{c \mathbin{;} d\ \mathsf{safe}\ q} \qquad \frac{p \neq q}{\bot^{GpH}\ \mathsf{safe}\ q}$$

Height $\boxed{||c||}$

$$
\begin{array}{ll}
||\mathtt{id}_A|| = 1 & ||c \to d|| = \max(||c||, ||d||) + 1 \\
||G\,\mathtt{!}|| = 1 & ||c \mathbin{;} d|| = \max(||c||, ||d||) \\
||G\mathtt{?}^p|| = 1 & ||\bot^{GpH}|| = 1
\end{array}
$$

Figure 3. Coercion calculus (λC)

`Fail` in Herman et al. (2007, 2010). Our type rules and definition of height are well-known; our reduction rules and all results in this section are new.

Blame labels and types are as in λB. Let c, d range over coercions. We write $c : A \Longrightarrow B$ to indicate that c coerces values of type A to type B. Our type rules follow Henglein (1994). The identity coercion at type A is written $\mathtt{id}_A$. Injection from ground type G to dynamic type is written $G\,\mathtt{!}$, and projection from dynamic type to ground type G is written $G\mathtt{?}^p$. The latter is decorated with a label p, to which blame is allocated if the projection fails. A function coercion $c \to d$ coerces a function $A \to B$ to a function $A' \to B'$, 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. The composition $c \mathbin{;} d$ coerces A to C, where c coerces A to B, and d coerces B to C. The fail coercion $\bot^{GpH}$ represents the result of a failed coercion from ground type G to ground type H, and is introduced because it is essential to the space-efficient representation described in the following section.

Terms of the calculus are as before, except that we replace casts by application of a coercion, $M\langle c\rangle$. The typing rule is straightforward:

$$\frac{\Gamma \vdash_{\mathsf{C}} M : A \qquad c : A \Longrightarrow B}{\Gamma \vdash_{\mathsf{C}} M\langle c\rangle : B}$$

If term M has type A, and c coerces A to B, then application to M of c is a term of type B.

Every well-typed coercion not containing failure has a unique type: if $c : A \Longrightarrow B$ and $c : A' \Longrightarrow B'$ and c does not contain a coercion of the form $\bot^{GpH}$ then $A = A'$ and $B = B'$. Conversely, distinct coercions may have the same type: for example, $\mathtt{id}_\star$ and $G\mathtt{?}^p \mathbin{;} G\,\mathtt{!}$ both have type $\star \Longrightarrow \star$.

Values and evaluation contexts are as in the blame calculus, with casts replaced by corresponding coercions. We write $M \longrightarrow_{\mathsf{C}} N$ to indicate that term M steps to term N. The identity coercion leaves a value unchanged. A coercion of a function applied to a value reduces to a term that coerces on the domain, applies the function, and coerces on the range. If an injection meets a matching projection, the coercion leaves the value unchanged. If an injection meets an incompatible projection, the coercion fails and allocates blame to the label in the projection. (Here it is clear why blame falls on the outer coercion: the inner coercion is an injection and has no blame label, while the outer is a projection with a blame label.) Application of a composed coercion applies each of the coercions in turn.

A coercion c is *safe* for blame label q, written $c\ \mathsf{safe}_{\mathsf{C}}\ q$, if application of the coercion never allocates blame to q. The definition is pleasingly simple: a coercion is safe for q if it does not mention label q.

Height of a coercion is as in Herman et al. (2007, 2010), and will be used in Section 4.

Type and blame safety and contextual equivalence for λC are as in λB. Propositions 3 and 5 and Definition 6 apply mutatis mutandis.

3.1 Relating λB to λC

The relation between λB and λC is presented in Figure 4. In this section, we let M, N range over terms of λB and M', N' range over terms of λC.

We write

$$|A \overset{p}{\Longrightarrow} B|^{\mathsf{BC}} = c$$

to indicate that the cast on the left translates to the coercion on the right. The translation is designed to ensure there is a lockstep bisimulation between λB and λC. The translation extends to terms in the obvious way, replacing each cast by the corresponding coercion.

We write

$$|c|^{\mathsf{CB}} = Z$$

to indicate that the coercion on the left translates to the sequence of casts on the right. Here Z ranges over sequences of casts. As defined in Figure 4, we write $Z \to B$ (respectively $B \to Z$) to replace in Z each source or target type A by $A \to B$ (respectively $B \to A$), we write $\overline{Z}$ to reverse the sequence Z and complement all the blame labels, and we write $Z \mathbin{+\!\!+} Z'$ to concatenate two sequences Z and Z', where the last type of one sequence must match the first of the other. In the clause for $c \to d$, the right-hand

Blame to coercion (λB to λC) $\boxed{|A \stackrel{p}{\Longrightarrow} B|^{\mathsf{BC}} = c}$

$$|\iota \stackrel{p}{\Longrightarrow} \iota|^{\mathsf{BC}} = \mathtt{id}_\iota$$
$$|A \to B \stackrel{p}{\Longrightarrow} A' \to B'|^{\mathsf{BC}} = |A' \stackrel{\overline{p}}{\Longrightarrow} A|^{\mathsf{BC}} \to |B \stackrel{p}{\Longrightarrow} B'|^{\mathsf{BC}}$$
$$|\star \stackrel{p}{\Longrightarrow} \star|^{\mathsf{BC}} = \mathtt{id}_\star$$
$$|G \stackrel{p}{\Longrightarrow} \star|^{\mathsf{BC}} = G!$$
$$|A \stackrel{p}{\Longrightarrow} \star|^{\mathsf{BC}} = |A \stackrel{p}{\Longrightarrow} G|^{\mathsf{BC}} ; G! \qquad \text{if } A \neq \star, A \neq G, A \sim G$$
$$|\star \stackrel{p}{\Longrightarrow} G|^{\mathsf{BC}} = G?^p$$
$$|\star \stackrel{p}{\Longrightarrow} A|^{\mathsf{BC}} = G?^p ; |G \stackrel{p}{\Longrightarrow} A|^{\mathsf{BC}} \qquad \text{if } A \neq \star, A \neq G, A \sim G$$

Coercion to blame (λC to λB) $\boxed{|c|^{\mathsf{CB}} = Z}$

$$|\mathtt{id}_A|^{\mathsf{CB}} = [\,]$$
$$|G!|^{\mathsf{CB}} = [G \stackrel{\bullet}{\Longrightarrow} \star]$$
$$|G?^p|^{\mathsf{CB}} = [\star \stackrel{p}{\Longrightarrow} G]$$
$$|c \to d|^{\mathsf{CB}} = \overline{(|c|^{\mathsf{CB}} \to B)} +\!\!+ (A' \to |d|^{\mathsf{CB}})$$
$$\text{where } c \to d : A \to B \Longrightarrow A' \to B'$$
$$|c\,;d|^{\mathsf{CB}} = |c|^{\mathsf{CB}} +\!\!+ |d|^{\mathsf{CB}}$$
$$|\perp^{GpH}_{A \Longrightarrow B}|^{\mathsf{CB}} = [A \stackrel{\bullet}{\Longrightarrow} G, G \stackrel{\bullet}{\Longrightarrow} \star, \star \stackrel{p}{\Longrightarrow} H, H \stackrel{\bullet}{\Longrightarrow} B]$$

where if

$$Z = [A_1 \stackrel{p_1}{\Longrightarrow} A_2, \cdots, A_m \stackrel{p_m}{\Longrightarrow} A_{m+1}]$$
$$Z' = [A_{m+1} \stackrel{p_{m+1}}{\Longrightarrow} A_{m+2}, \cdots, A_{m+n} \stackrel{p_{m+n}}{\Longrightarrow} A_{m+n+1}]$$

then

$$Z \to B = [A_1{\to}B \stackrel{p_1}{\Longrightarrow} A_2{\to}B, \cdots, A_m{\to}B \stackrel{p_m}{\Longrightarrow} A_{m+1}{\to}B]$$
$$B \to Z = [B{\to}A_1 \stackrel{p_1}{\Longrightarrow} B{\to}A_2, \cdots, B{\to}A_m \stackrel{p_m}{\Longrightarrow} B{\to}A_{m+1}]$$
$$\overline{Z} = [A_{m+1} \stackrel{\overline{p_m}}{\Longrightarrow} A_m, \cdots, A_2 \stackrel{\overline{p_1}}{\Longrightarrow} A_1]$$
$$Z +\!\!+ Z' = [A_1 \stackrel{p_1}{\Longrightarrow} A_2, \cdots, A_{m+n} \stackrel{p_{m+n}}{\Longrightarrow} A_{m+n+1}]$$

Figure 4. Relating λB to λC

side can be taken as either

$$\overline{(|c|^{\mathsf{CB}}{\to}B)} +\!\!+ (A'{\to}|d|^{\mathsf{CB}}) \;\text{ or }\; (A{\to}|d|^{\mathsf{CB}}) +\!\!+ \overline{(|c|^{\mathsf{CB}}{\to}B')},$$

equivalently. We write $\perp^{GpH}_{A\Longrightarrow B}$ to indicate that $\perp^{GpH}$ is used as a cast from A to B. This is an informal notation, with the extra information easily recovered by type inference. We choose not to use $\perp^{GpH}_{A\Longrightarrow B}$ as a formal notation throughout, since it would complicate the definition of $\fatsemi$ in Section 4. We write $\bullet$ as a blame label in casts where the label is irrelevant because the cast cannot allocate blame. The translation extends to terms in the obvious way, replacing each coercion by the corresponding sequence of casts.

As a first step justifying this definition, we observe several contextual equivalences for λC.

Lemma 7 (Equivalences)**.** *The following hold in* λC*.*

1. $M\langle \mathtt{id}\rangle \stackrel{\mathsf{ctx}}{=}_{\mathsf{C}} M$
2. $M\langle c\,;d\rangle \stackrel{\mathsf{ctx}}{=}_{\mathsf{C}} M\langle c\rangle\langle d\rangle$
3. $M\langle c \to d\rangle \stackrel{\mathsf{ctx}}{=}_{\mathsf{C}} M\langle (c \to \mathtt{id})\,;(\mathtt{id} \to d)\rangle$
4. $M\langle c \to d\rangle \stackrel{\mathsf{ctx}}{=}_{\mathsf{C}} M\langle (\mathtt{id} \to c)\,;(d \to \mathtt{id})\rangle$

Proof of this lemma is deferred to Section 5.1, where we apply a new technique that makes the proof straightforward.

Translating from λC to λB and back again is the identity, up to contextual equivalence.

Lemma 8 (Coercions to blame)**.** *If* M' *is a term of* λC *then* $||M'|^{\mathsf{CB}}|^{\mathsf{BC}} \stackrel{\mathsf{ctx}}{=}_{\mathsf{C}} M'$*.*

The subtle definition of positive and negative subtyping is justified by the correspondence to the coercion calculus. It is not too surprising that the definition is sound (safety in B implies safety in C), but it is surprising that the definition is also complete (safety in C implies safety in B).

Lemma 9 (Positive and negative subtyping)**.**

1. $A <:^+ B$ *iff* $|A \stackrel{p}{\Longrightarrow} B|^{\mathsf{BC}}$ $\mathsf{safe}_{\mathsf{C}}$ p*.*
2. $A <:^- B$ *iff* $|A \stackrel{p}{\Longrightarrow} B|^{\mathsf{BC}}$ $\mathsf{safe}_{\mathsf{C}}$ $\overline{p}$*.*

(The full proof is in the supplementary material.)

It follows immediately that translation from λB to λC preserves type and blame safety.

Proposition 10 (Preservation, λB to λC)**.**

1. *If* $\Gamma \vdash_{\mathsf{B}} M : A$ *then* $\Gamma \vdash_{\mathsf{C}} |M|^{\mathsf{BC}} : A$*.*
2. *If* M $\mathsf{safe}_{\mathsf{B}}$ q *then* $|M|^{\mathsf{BC}}$ $\mathsf{safe}_{\mathsf{C}}$ q*.*

The translation from λB to λC is a bisimulation. The bisimulation is lockstep: a single step in λB corresponds to a single step in λC, and vice versa.

Proposition 11 (Bisimulation, λB to λC)**.**
Assume $\vdash_{\mathsf{B}} M : A$ *and* $\vdash_{\mathsf{C}} M' : A$ *and* $|M|^{\mathsf{BC}} = M'$*.*

1. *If* $M \longrightarrow_{\mathsf{B}} N$ *then* $M' \longrightarrow_{\mathsf{C}} N'$ *and* $|N|^{\mathsf{BC}} = N'$ *for some* N'*.*
2. *If* $M' \longrightarrow_{\mathsf{C}} N'$ *then* $M \longrightarrow_{\mathsf{B}} N$ *and* $|N|^{\mathsf{BC}} = N'$ *for some* N*.*
3. *If* $M = V$ *then* $M' = V'$ *and* $|V|^{\mathsf{BC}} = V'$ *for some* V'*.*
4. *If* $M' = V'$ *then* $M = V$ *and* $|V|^{\mathsf{BC}} = V'$ *for some* V*.*
5. *If* $M = \mathtt{blame}\, p$ *then* $M' = \mathtt{blame}\, p$*.*
6. *If* $M' = \mathtt{blame}\, p$ *then* $M = \mathtt{blame}\, p$*.*

Translation from λB to λC is fully abstract.

Proposition 12 (Fully abstract, λB to λC)**.** *If* M *and* N *are terms of* λB *then* $M \stackrel{\mathsf{ctx}}{=}_{\mathsf{B}} N$ *iff* $|M|^{\mathsf{BC}} \stackrel{\mathsf{ctx}}{=}_{\mathsf{C}} |N|^{\mathsf{BC}}$*.*

4. Space-efficient Coercion Calculus

Figure 5 defines the space-efficient coercion calculus, λS. Space-efficient coercions correspond to coercions in a canonical form. All results in this section are new.

Blame labels and types are as in λB and λC. Space-efficient coercions follow a specific, three-part grammar. There is one space-efficient coercion for each equivalence class of coercions with respect to the equational theory of Henglein (1994). The grammar has been chosen to facilitate the definition of a recursive composition operator, that takes two canonical coercions and computes the canonical coercion corresponding to their composition.

Let s, t range over space-efficient coercions, i range over intermediate coercions, and g, h range over ground coercions. Space-efficient coercions are either the identity coercion at dynamic type $\mathtt{id}_\star$, a projection followed by an intermediate coercion $(G?^p; i)$, or just an intermediate coercion i. An intermediate coercion is either a ground coercion followed by an injection $(g\,; G!)$, just a ground coercion g, or the failure coercion $\perp^{GpH}$. A ground coercion is an identity coercion of base type $\mathtt{id}_\iota$ or a function coercion $s \to t$. Let f range over identity-free coercions, which play a role in reduction.

Syntax

$$
\begin{aligned}
s, t &::= \mathtt{id}_\star \mid (G?^p \,;\, i) \mid i \\
i &::= (g \,;\, G!) \mid g \mid \perp^{GpH} \\
g, h &::= \mathtt{id}_\iota \mid (s \to t) \\
f &::= (G?^p \,;\, i) \mid (g \,;\, G!) \mid \perp^{GpH} \mid (s \to t) \\
L, M, N &::= k \mid op(\vec{M}) \mid x \mid \lambda x{:}A.\, N \mid L\ M \mid M\langle t\rangle \mid \mathtt{blame}\ p \\
U &::= k \mid \lambda x{:}A.\, N \\
V, W &::= U \mid U\langle s \to t\rangle \mid U\langle g \,;\, G!\rangle \\
\mathcal{E} &::= \mathcal{F} \mid \mathcal{F}[\Box\langle f\rangle] \\
\mathcal{F} &::= \Box \mid \mathcal{E}[op(\vec{V}, \Box, \vec{M})] \mid \mathcal{E}[\Box\ M] \mid \mathcal{E}[V\ \Box]
\end{aligned}
$$

Composition $\boxed{s \mathbin{⨟} t = r}$

$$
\begin{aligned}
\mathtt{id}_\iota \mathbin{⨟} \mathtt{id}_\iota &= \mathtt{id}_\iota \\
(s \to t) \mathbin{⨟} (s' \to t') &= (s' \mathbin{⨟} s) \to (t \mathbin{⨟} t') \\
\mathtt{id}_\star \mathbin{⨟} t &= t \\
(g \,;\, G!) \mathbin{⨟} \mathtt{id}_\star &= g \,;\, G! \\
(G?^p \,;\, i) \mathbin{⨟} t &= G?^p \,;\, (i \mathbin{⨟} t) \\
g \mathbin{⨟} (h \,;\, H!) &= (g \mathbin{⨟} h) \,;\, H! \\
(g \,;\, G!) \mathbin{⨟} (G?^p \,;\, i) &= g \mathbin{⨟} i \\
(g \,;\, G!) \mathbin{⨟} (H?^p \,;\, i) &= \perp^{GpH} \qquad \text{if } G \neq H \\
\perp^{GpH} \mathbin{⨟} s &= \perp^{GpH} \\
g \mathbin{⨟} \perp^{GpH} &= \perp^{GpH}
\end{aligned}
$$

Reduction $\boxed{M \longrightarrow_{\mathsf{S}} N}$

$$
\begin{aligned}
\mathcal{E}[(U\langle s \to t\rangle)\ V] &\longrightarrow \mathcal{E}[(U\ (V\langle s\rangle))\langle t\rangle] \\
\mathcal{F}[U\langle \mathtt{id}_\iota\rangle] &\longrightarrow \mathcal{F}[U] \\
\mathcal{F}[U\langle \mathtt{id}_\star\rangle] &\longrightarrow \mathcal{F}[U] \\
\mathcal{F}[M\langle s\rangle\langle t\rangle] &\longrightarrow \mathcal{F}[M\langle s \mathbin{⨟} t\rangle] \\
\mathcal{F}[U\langle \perp^{GpH}\rangle] &\longrightarrow \mathtt{blame}\ p \\
\mathcal{E}[\mathtt{blame}\ p] &\longrightarrow \mathtt{blame}\ p \qquad \text{if } \mathcal{E} \neq \Box
\end{aligned}
$$

Figure 5. Space-efficient coercion calculus (λS)

The source of an intermediate coercion is never the dynamic type. Source and target of a ground coercion are never the dynamic type, and both are compatible with the same unique ground type.

Lemma 13 (Source and Target)**.**

1. *If $i : A \Longrightarrow B$ then $A \neq \star$.*
2. *If $g : A \Longrightarrow B$ then $A \neq \star$ and $B \neq \star$ and there exists a unique G such that $A \sim G$ and $G \sim B$.*

Terms of the calculus are as in λC, except that we restrict coercions to space-efficient coercions. The key idea of the dynamics, as in Herman et al. (2007, 2010) and Siek and Wadler (2010), is to combine and normalize adjacent coercions, which ensures space efficiency. Ensuring adjacent coercions are combined requires we adjust the notion of value and evaluation context. 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 top-level coercion. Let $\mathcal{E}$ range over contexts and $\mathcal{F}$ range over coercion-free contexts, where no context applies two coercions in succession, each applied coercion is identity free, and a coercion-free context does not have a coercion application innermost. Reduction of a term that is a cast must occur in a cast-free context. These adjustments ensure that if a term contains two coercions in succession in an evaluation context, then those coercions are composed before other reductions occur. The other reduction rules are straightforward.

If space-efficient coercions s and t are the canonical form of coercions c and d, then $s \mathbin{⨟} t$ is the canonical form of $c \,;\, d$. We establish the termination of composition by observing that the sum of the sizes of the arguments gets smaller at each recursive call. Further the correctness of each equation in the definition is easily justified by the equational theory of Henglein (1994).

Height is preserved by composition.

Proposition 14 (Height)**.** $||s \mathbin{⨟} t|| \leq \max(||s||, ||t||)$.

A space-efficient coercion contains at most two compositions (check the grammar), so a space-efficient coercion bounded in height is also bounded in size.

Type and blame safety and contextual equivalence are as in λB. The definition of blame safety from Figure 3, Propositions 3 and 5, and Definition 6 apply mutatis mutandis.

4.1 Relating λC to λS

The translation from λC to λS is presented in Figure 6. In this section, we let M, N range over terms of λC and let M', N' range over terms of λS.

We write

$$|c|^{\mathsf{CS}} = s$$

to indicate that the coercion on the left translates to the space-efficient coercion on the right. The translation extends to terms in the obvious way, replacing each coercion by the corresponding space-efficient coercion.

The inverse translation

$$|s|^{\mathsf{SC}} = c$$

is trivial, since each space-efficient coercion is a coercion.

Translating λC to λS preserves type and blame safety.

Proposition 15 (Preservation, λC to λS)**.**

1. *If $\Gamma \vdash_{\mathsf{C}} M : A$ then $\Gamma \vdash_{\mathsf{S}} |M|^{\mathsf{CS}} : A$.*
2. *If M $\mathsf{safe}_{\mathsf{C}}$ q then $|M|^{\mathsf{CS}}$ $\mathsf{safe}_{\mathsf{S}}$ q.*

Dynamics of λC and λS differ in that the former breaks up compositions, while the latter combines them. In Figure 6, we define a bisimulation $\approx$ that relates λC to λS. Rules in grey make the relation a congruence; rules (i), (ii), (iii) relate a sequence of zero or more coercion applications to a single space-efficient coercion application. Consider the sequence of reductions in λC.

$$
\begin{aligned}
& (V\langle c_1 \to d_1\rangle\langle c_2 \to d_2\rangle)\ W && \text{(a)} \\
\longrightarrow_{\mathsf{C}}\ & ((V\langle c_1 \to d_1\rangle)\ (W\langle c_2\rangle))\langle d_2\rangle && \text{(b)} \\
\longrightarrow_{\mathsf{C}}\ & (V\ (W\langle c_2\rangle\langle c_1\rangle))\langle d_1\rangle\langle d_2\rangle && \text{(c)}
\end{aligned}
$$

If $V \approx V'$, $W \approx W'$, $|c_i|^{\mathsf{CS}} = s_i$, and $|d_i|^{\mathsf{CS}} = t_i$, these two reductions relate to a single reduction in λS.

$$
\begin{aligned}
& (V\langle (s_2 \mathbin{⨟} s_1) \to (t_1 \mathbin{⨟} t_2)\rangle)\ W && \text{(d)} \\
\longrightarrow_{\mathsf{S}}\ & (V\ (W\langle s_2 \mathbin{⨟} s_1\rangle))\langle t_1 \mathbin{⨟} t_2\rangle && \text{(e)}
\end{aligned}
$$

Here (a) $\approx$ (d) via (i) once and (ii) twice; and (b) $\approx$ (d) via (i) once, (ii) once, and (iii) once; and (c) $\approx$ (e) via (i) once and (ii) twice in both the domain and the range.

Relation $\approx$ is a bisimulation. It is not lockstep: a single step in λC corresponds to zero or more steps in λS, and vice versa.

Coercions to space-efficient ($\lambda\mathsf{C}$ to $\lambda\mathsf{S}$) $\boxed{|c|^{\mathsf{CS}} = s}$

$$
\begin{aligned}
|\mathtt{id}_\star|^{\mathsf{CS}} &= \mathtt{id}_\star \\
|\mathtt{id}_\iota|^{\mathsf{CS}} &= \mathtt{id}_\iota \\
|\mathtt{id}_{A\to B}|^{\mathsf{CS}} &= |\mathtt{id}_A|^{\mathsf{CS}} \to |\mathtt{id}_B|^{\mathsf{CS}} \\
|G?^p|^{\mathsf{CS}} &= G?^p \,;\, |\mathtt{id}_G|^{\mathsf{CS}} \\
|G!|^{\mathsf{CS}} &= |\mathtt{id}_G|^{\mathsf{CS}} \,;\, G! \\
|c \to d|^{\mathsf{CS}} &= |c|^{\mathsf{CS}} \to |d|^{\mathsf{CS}} \\
|c\,;d|^{\mathsf{CS}} &= |c|^{\mathsf{CS}} \fatsemi |d|^{\mathsf{CS}} \\
|\bot^{GpH}|^{\mathsf{CS}} &= \bot^{GpH}
\end{aligned}
$$

Bisimulation between $\lambda\mathsf{C}$ and $\lambda\mathsf{S}$ $\boxed{M \approx_{\mathsf{CS}} M'}$

$$
\frac{}{k \approx k} \qquad \frac{\vec{M} \approx \vec{M}'}{op(\vec{M}) \approx op(\vec{M}')} \qquad \frac{}{x \approx x}
$$

$$
\frac{M \approx M'}{\lambda x{:}A.\, M \approx \lambda x{:}A.\, M'} \qquad \frac{L \approx L' \quad M \approx M'}{L\ M \approx L'\ M'}
$$

$$
\frac{}{\mathtt{blame}\ p \approx \mathtt{blame}\ p}
$$

$$
\frac{M \approx M' \quad \vdash M : A \quad |\mathtt{id}_A|^{\mathsf{CS}} = s}{M \approx M'\langle s\rangle} \tag{i}
$$

$$
\frac{M \approx M'\langle s\rangle \quad |c|^{\mathsf{CS}} = t}{M\langle c\rangle \approx M'\langle s \fatsemi t\rangle} \tag{ii}
$$

$$
\frac{M \approx (L'\langle r\rangle)\ (M'\langle s\rangle) \quad |d|^{\mathsf{CS}} = t}{M\langle d\rangle \approx (L'\langle r \fatsemi (s \to t)\rangle)\ M'} \tag{iii}
$$

Figure 6. Relating $\lambda\mathsf{C}$ to $\lambda\mathsf{S}$

Proposition 16 (Bisimulation, $\lambda\mathsf{C}$ to $\lambda\mathsf{S}$)**.** *Assume $\vdash_{\mathsf{C}} M : A$ and $\vdash_{\mathsf{S}} M' : A$ and $M \approx M'$.*

1. *If $M \longrightarrow_{\mathsf{C}} N$ then $M' \longrightarrow^*_{\mathsf{S}} N'$ and $N \approx N'$ for some N'.*
2. *If $M' \longrightarrow_{\mathsf{S}} N'$ then $M \longrightarrow^*_{\mathsf{C}} N$ and $N \approx N'$ for some N.*
3. *If $M = V$ then $M' \longrightarrow^*_{\mathsf{S}} V'$ and $V \approx V'$ for some V'.*
4. *If $M' = V'$ then $M \longrightarrow^*_{\mathsf{C}} V$ and $V \approx V'$ for some V.*
5. *If $M = \mathtt{blame}\ p$ then $M' = \mathtt{blame}\ p$.*
6. *If $M' = \mathtt{blame}\ p$ then $M = \mathtt{blame}\ p$.*

(The full proof is in the supplementary material.)

Terms relate to their translations by $\approx$.

Proposition 17. $M \approx |M|^{\mathsf{CS}}$.

Translation from $\lambda\mathsf{C}$ to $\lambda\mathsf{S}$ is fully abstract.

Proposition 18 (Fully abstract, $\lambda\mathsf{C}$ to $\lambda\mathsf{S}$)**.** *If M and N are terms of $\lambda\mathsf{C}$ then $M \stackrel{\text{ctx}}{\cong}_{\mathsf{C}} N$ iff $|M|^{\mathsf{CS}} \stackrel{\text{ctx}}{\cong}_{\mathsf{S}} |N|^{\mathsf{CS}}$.*

5. Applications

Full abstraction considerably eases some proofs. In this section, we use it to demonstrate two useful results, Lemma 7 from Section 3.1, which justifies the translation $|\cdot|^{\mathsf{CB}}$, and the Fundamental Law of Casts from Siek and Wadler (2010).

5.1 Lemma 7

Lemma 7 from Section 3.1 is used to justify the design of $|\cdot|^{\mathsf{CB}}$, the mapping from $\lambda\mathsf{C}$ back to $\lambda\mathsf{B}$. We repeat the lemma here, with some additional clauses.

Lemma 19 (Equivalences)**.** *The following hold in $\lambda\mathsf{C}$.*

1. $M\langle \mathtt{id}\rangle \stackrel{\text{ctx}}{\cong}_{\mathsf{C}} M$
2. $M\langle c\,;d\rangle \stackrel{\text{ctx}}{\cong}_{\mathsf{C}} M\langle c\rangle\langle d\rangle$
3. $M\langle c\,;\mathtt{id}\rangle \stackrel{\text{ctx}}{\cong}_{\mathsf{C}} M\langle c\rangle \stackrel{\text{ctx}}{\cong}_{\mathsf{C}} M\langle \mathtt{id}\,;c\rangle$
4. $M\langle (c \to d)\,;(c' \to d')\rangle \stackrel{\text{ctx}}{\cong}_{\mathsf{C}} M\langle (c'\,;c) \to (d\,;d')\rangle$
5. $M\langle c \to d\rangle \stackrel{\text{ctx}}{\cong}_{\mathsf{C}} M\langle (c \to \mathtt{id})\,;(\mathtt{id} \to d)\rangle$
6. $M\langle c \to d\rangle \stackrel{\text{ctx}}{\cong}_{\mathsf{C}} M\langle (\mathtt{id} \to c)\,;(d \to \mathtt{id})\rangle$

Proof. Part 1 follows from $M\langle \mathtt{id}\rangle \longrightarrow_{\mathsf{C}} M$, part 2 is similar, and part 3 follows from parts 1 and 2. Part 4 is more interesting. Let $|c|^{\mathsf{CS}} = s$, $|d|^{\mathsf{CS}} = t$, $|c'|^{\mathsf{CS}} = s'$ and $|d'|^{\mathsf{CS}} = t'$. Applying $|\cdot|^{\mathsf{CS}}$ to each side of the equation gives

$$(s \to t) \fatsemi (s' \to t') \stackrel{\text{ctx}}{\cong}_{\mathsf{S}} (s' \fatsemi s) \to (t \fatsemi t')$$

which holds immediately from the definition of $\fatsemi$. Then part 4 follows because $|\cdot|^{\mathsf{CS}}$ reflects contextual equivalence, the backward part of Proposition 18. Part 5 follows from

$$c \to d \stackrel{\text{ctx}}{\cong}_{\mathsf{C}} (\mathtt{id}\,;c) \to (\mathtt{id}\,;d) \stackrel{\text{ctx}}{\cong}_{\mathsf{C}} (c \to \mathtt{id})\,;(\mathtt{id} \to d)$$

which follows from parts 3 and 4. Part 6 is similar. □

Typically, one might be tempted to prove a result such as Lemma 7 by introducing a custom bisimulation relation—indeed, that is how we first attempted to demonstrate it. Eventually we realised that we could show terms equivalent in $\lambda\mathsf{C}$ by mapping them into $\lambda\mathsf{S}$ and exploiting full abstraction. Instead of introducing a custom bisimulation relation, all of the “heavy lifting” is done by bisimulation $\approx$ from Figure 6 and by Proposition 16.

Full abstraction from $\lambda\mathsf{C}$ to $\lambda\mathsf{S}$ does not depend of full abstraction from $\lambda\mathsf{B}$ to $\lambda\mathsf{C}$, so there is no circularity.

5.2 Fundamental Property of Casts

As a second application, we show how to establish the Fundamental Property of Casts, Lemma 2 of Siek and Wadler (2010), which asserts that a single cast is contextually equivalent to a pair of casts. We will do so by mapping two terms of $\lambda\mathsf{B}$ to contextually equivalent terms of $\lambda\mathsf{S}$.

First, we define a notion of pointed type.

$$S, T ::= \iota \mid S \to T \mid \star \mid \bot$$

We extend naive subtyping to include pointed types by setting $\bot <:_n T$, for all T. Meet of two types is a pointed type, $A \mathbin{\&} B = T$. The meet of two types is their greatest lower bound with respect to naive subtyping, $<:_n$.

Take $|-|^{\mathsf{BS}}$ to be the composition of $|-|^{\mathsf{BC}}$ and $|-|^{\mathsf{CS}}$. We first establish one simple lemma, which follows immediately by case analysis on A, B, and C.

Lemma 20. *If $A \mathbin{\&} B <:_n C$ then*

$$|A \stackrel{p}{\Longrightarrow} B|^{\mathsf{BS}} = |A \stackrel{p}{\Longrightarrow} C|^{\mathsf{BS}} \fatsemi |C \stackrel{p}{\Longrightarrow} B|^{\mathsf{BS}}$$

The fundamental property follows immediately by full abstraction from $\lambda\mathsf{B}$ to $\lambda\mathsf{C}$ and $\lambda\mathsf{C}$ to $\lambda\mathsf{S}$.

Lemma 21 (Fundamental Property of Casts)**.** *Let M be a term of $\lambda\mathsf{B}$. If $A \mathbin{\&} B <:_n C$ then*

$$M : A \stackrel{p}{\Longrightarrow} B \stackrel{\text{ctx}}{\cong}_{\mathsf{B}} M : A \stackrel{p}{\Longrightarrow} C \stackrel{p}{\Longrightarrow} B$$

Siek and Wadler (2010) establish the same result with more difficulty: they require a custom bisimulation and six lemmas.

6. Related Work

This section provides an in-depth comparison to the work of Siek and Wadler (2010), Greenberg (2013), and Garcia (2013), then summarizes systems that use gradual typing and other relevant work.

6.1 Relation to Siek and Wadler (2010)

Siek and Wadler (2010) use threesomes of the form

$$\langle T \overset{P}{\Longleftarrow} S\rangle\, s$$

where s is a term, S, T are types, and P is a labeled type that indicates how blame is allocated if the cast fails. Here is the grammar for labeled types:

$$\begin{aligned} p, q &::= l \mid \epsilon \\ P, Q &::= B^p \mid P \to^p Q \mid \star \mid \bot^{lGp} \end{aligned}$$

Their l, m range over blame labels (our p, q), their p, q range over optional blame labels, their P, Q range over labeled types, their B ranges over base types (our ι), and their G, H range over ground types (our G, H). The meaning of a labeled type is subtle as it depends on whether each label is present or not. For example, their $\bot^{lG\epsilon}$ corresponds to our $\bot^{GpH}$, while their $\bot^{lGm}$ correspond to our $G?^q ; \bot^{GpH}$ (taking their l, m to correspond to our p, q, respectively). Their paper includes a translation $(\!|-|\!)$ from threesomes to coercions.

If our space-efficient coercions s, t correspond to their labeled types P, Q, then $s \,⨟\, t$ corresponds to $Q \circ P$ (note the reversal!), defined as follows.

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

Here P^{Gp} means that labelled type P is compatible with ground type G and that p is the topmost optional blame label in P. The correctness of these equations is not immediate. For instance, in the penultimate line why do P^{Gp} and $\bot^{mHl}$ compose to yield $\bot^{lGp}$? Perhaps the easiest way to validate the equations is to translate to coercions using $(\!|-|\!)$, then check that the left-hand side normalises to the right-hand side. In contrast, our definition of $⨟$ (Figure 5) is easily justified by the equational theory of Henglein (1994).

6.2 Relation to Greenberg (2013)

Greenberg (2013) considers a sequence of calculi CAST, NAIVE, and EFFICIENT, roughly corresponding to our λB, λC, and λS. Unlike us, he includes refinement types, but omits blame; and he formulates correctness in terms of logical relations rather than full abstraction.

His EFFICIENT resembles our λS, in that it defines a composition operator that serves the same purpose as our $⨟$. He writes $c_1 * c_2 \Rightarrow c_3$ to indicate that the composition of c_1 and c_2 is equivalent to c_3. The rules to compute $c_1 * c_2$ compose the right-most primitive coercion of c_1 with the left-most primitive coerection of c_2, then recursivley compose the result with what is left of c_1 and c_2. For example, here is the rule for composing function coercions.

$$\frac{\begin{array}{c} c_{21} * c_{11} \Rightarrow c_{31} \\ c_{12} * c_{22} \Rightarrow c_{32} \\ c_1 * (c_{31} \to c_{32}) \,;\, c_2 \Rightarrow c \end{array}}{c_1 \,;\, (c_{11} \to c_{12}) * (c_{21} \to c_{22}) \,;\, c_2 \Rightarrow c}$$

His definition is recursive but not a structural recursion, and proving it total is challenging, requiring four pages. In contrast, our definition is a structural recursion, and totality is straightforward.

6.3 Relation to Garcia (2013)

Garcia (2013) observes that coercions are easier to understand while threesomes are easier to implement, and shows how to derive threesomes from coercions through a series of correctness-preserving transformations. To accomplish this, he defines super-coercions and gives their meaning in terms of a translation $\mathcal{N}(-)$ to coercions.

$$\begin{aligned} \mathcal{N}(\iota_P) &= \iota_P \\ \mathcal{N}(\mathtt{Fail}^l) &= \mathtt{Fail}^l \\ \mathcal{N}(\mathtt{Fail}^{l_1 G l_2}) &= \mathtt{Fail}^{l_1} \circ G?^{l_2} \\ \mathcal{N}(G!) &= G! \\ \mathcal{N}(G?^l) &= G?^l \\ \mathcal{N}(G?^l!) &= G! \circ G?^l \\ \mathcal{N}(\ddot{c}_1 \to \ddot{c}_2) &= \mathcal{N}(\ddot{c}_1) \to \mathcal{N}(\ddot{c}_2) \\ \mathcal{N}(\ddot{c}_1 \,!\!\to \ddot{c}_2) &= (\star \to \star)! \circ (\mathcal{N}(\ddot{c}_1) \to \mathcal{N}(\ddot{c}_2)) \\ \mathcal{N}(\ddot{c}_1 \to ?^l\, \ddot{c}_2) &= (\mathcal{N}(\ddot{c}_1) \to \mathcal{N}(\ddot{c}_2)) \circ (\star \to \star)?^l \\ \mathcal{N}(\ddot{c}_1 \,!\!\to ?^l \ddot{c}_2) &= (\star \to \star)! \circ (\mathcal{N}(\ddot{c}_1) \to \mathcal{N}(\ddot{c}_2)) \circ (\star \to \star)?^l \end{aligned}$$

His l ranges over blame labels (our p, q), his ι is the identity coercion (our id), his P ranges over atomic types (either a base type or the dynamic type), his $\mathtt{Fail}^l$ is a failure coercions (our $\bot^{GpH}$), and his $\ddot{c}$ ranges over supercoercions. Garcia (2013) derives a recursive composition function for supercoercions but the definition was too large to publish as there are sixty pairs of compatible supercoercions. In contrast, our definition fits in ten lines.

6.4 Systems that use Gradual Typing

Racket (formerly Scheme) supports dynamic and static typing and higher-order contracts with blame (Flatt and PLT 2014). Racket permits contracts to be written directly. Typed Racket inserts contracts that allocate blame when dynamically typed code fails to conform to the static types declared for it (Tobin-Hochstadt and Felleisen 2008). Racket has an extensive and well-tested implementation of contracts, but does not support space-efficient contracts. Racket is the source, via Findler and Felleisen (2002), of the rule for casting functions in λB (the fourth reduction rule in Figure 1).

Pyret has limited support for gradual typing (Patterson et al. 2014). Pyret checks that a first-order value (such as integer) conforms to its declaration, but only checks that a higer-order value is a function, not that it conforms to its declared parameter and result types. Pyret does not implement any equivalent of the rule for casting functions in λB.

Dart provides support for gradual typing with implicit casts to and from type `dynamic` (Bracha and Bak 2011; ECMA 2014). Dart does not provide full static type checking; its type checker aims to warn of likely errors rather than to ensure lack of failures. In checked mode, Dart performs a test at every place that a value can be assigned to a variable and raises an exception if the value's type is not a subtype of the variable's declared type. Dart does not implement any equivalent of the rule for casting functions in λB.

C# type `dynamic` and VB type `Object` play a role similar to our type $\star$, with the compiler introducing first-order casts as needed (Bierman et al. 2010; Feigenbaum 2008). These languages do not have higher-order structural types, only nominal types, so the programmer must manually construct explicit wrappers to accomplish what would amount to a higher-order cast. C# and VB do not implement any equivalent of the rule for casting functions in λB.

TypeScript provides `interface` declarations that allow users to specify types for an imported JavaScript module or library (Hejlsberg 2012). The DefinitelyTyped repository contains over 150 such declarations for a variety of popular JavaScript libraries (Yankov 2013). TypeScript is not concerned with type soundness, which it does not provide (Bierman et al. 2014), but instead exploits types to provide better prompting in Visual Studio, for instance to to populate a pulldown menu with well-typed methods that might be invoked at a given point. The information supplied by `interface` declarations is taken on faith; failures to conform to the declaration are not reported. Typescript does not implement any equivalent of the rule for casting functions in λB.

Several systems explore how to modify TypeScript to restore various forms of type safety.

Safe TypeScript is a refinement of TypeScript that guarantees type safety by adding run-time type information (RTTI) to values of dynamic type `any` (Rastogi et al. 2015). It introduces the notion of erased types that cannot be coerced to `any`. Erased types are used to communicate with external libraries that are unaware of RTTI. Furthermore, subtyping of function types is restricted to never manipulate RTTI, avoiding the need for wrappers that may change the object identity. Safe Typescript does not implement any equivalent of the rule for casting functions in λB.

StrongScript (Richards et al. 2015) extends TypeScript's optional types with concrete types. A concrete type is a (nominal) class type which is statically checked and which is protected by compiler-generated casts against its less strictly typed context. The main goals of this work are compatibility with TypeScript and enabling the generation of efficient code for concretely typed parts of a program. Blame tracking is an optional feature that may be disabled to avoid run-time overhead. StrongScript relies upon an equivalent of the rule for casting functions in λB.

Microsoft has funded Wadler and a PhD student to build a tool, TypeScript TNG, that uses blame calculus to generate wrappers from TypeScript `interface` declarations. The wrappers monitor interactions between a library and a client, and if a failure occurs then blame will indicate whether it is the library or the client that has failed to conform to the declared types. TypeScript TNG relies upon an equivalent of the rule for casting functions in λB.

Initial results on TypeScript TNG appear promising, but there is much to do. We need to assess how many and what sort of errors are revealed by wrappers, and measure the overhead wrappers introduce. It would be desirable to ensure that generated wrappers never change the semantics of programs (save to detect more errors) but aspects of JavaScript (notably, that wrappers affect pointer equality) make it difficult to guarantee noninterference; we need to determine to what extent these cases are an issue in practice. The current design of TypeScript TNG is not space-efficient, and implementing a space-efficient version and measuring its effect would be interesting future work.

6.5 Other Relevant Work

Abadi et al. (1991) study an early notion of type `Dynamic`. Floyd (1967) and Hoare (1969) introduce reasoning about programs with pre- and post-conditions and Meyer (1988) popularises checking them at runtime under the name *contracts*. Findler and Felleisen (2002) introduce higher-order contracts for functional languages.

Tobin-Hochstadt and Felleisen (2006) formalize the interaction between static and dynamic typing at the granularity of modules and prove a precursor to blame safety. Matthews and Findler (2007) define an operational semantics for multi-language programs with static (ML) and dynamic (Scheme) components. Gronski et al. (2006) present Sage, a gradually-typed language with refinement types. Dimoulas et al. (2011, 2012) develop criteria for judging blame tracking strategies. Disney et al. (2011) extend contracts with temporal properties. Strickland et al. (2012) study contracts for mutable objects. Thiemann (2014) takes first steps towards gradual typing for session types.

Hinze et al. (2006) design an embedded DSL for contracts with blame assignment in Haskell. Chitil (2012) develops a lazy version of contracts for Haskell. Greenberg et al. (2010) study dependent contracts and the translation between latent and manifest systems. 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 language TS* into JavaScript.

Siek et al. (2009) explore design choices for cast checking and blame tracking in the setting of the coercion calculus. Ahmed et al. (2011) extend the blame calculus to include parametric polymorphism. Siek and Garcia (2012) define a space-efficient abstract machine for the gradually-typed lambda calculus based on coercions. Siek et al. (2015) propose the *gradual guarantee* as a new criteria for gradual typing, characterizing how changes in the precision of type annotations may change a program's static and dynamic semantics. Wadler (2015) surveys work on the blame calculus.

7. Conclusion

Findler and Felleisen (2002) introduced higher-order contracts, setting up a foundation for gradual typing; but they observed a problem with space efficiency. Herman et al. (2007, 2010) restored space efficiency; but required an evaluator to reassociate parentheses. Siek and Wadler (2010) gave a recursive definition of composition that is easy to compute; but the correctness of their definition is not transparent. Here we provide composition that is easy to compute and transparent. At last, we are in a position to implement space-efficient contracts and test them in practice.

When Siek and Wadler (2010) was published we thought we had discovered a solution that was easy to implement and easy to understand. Only later did we realise that it was not quite so easy as we thought! We believe this next step is a significant improvement. For us, the lesson is clear: no matter how simple your theory, strive to make it simpler still!

Acknowledgments

Thanks to Shayan Najd, Michael Greenberg, and the PLDI referees for comments. Siek acknowledges NSF Grant 1360694. Wadler acknowledges EPSRC Programme Grant EP/K034413/1 and a Microsoft Research PhD Scholarship.

References

M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Prog. Lang. Syst.*, 13(2):237–268, April 1991.

A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In *Principles of Programming Languages (POPL)*, pages 201–214, 2011.

N. Benton. Undoing dynamic typing (declarative pearl). In J. Garrigue and M. Hermenegildo, editors, *Functional and Logic Programming*, volume 4989 of *Lecture Notes in Computer Science*, pages 224–238. Springer Berlin Heidelberg, 2008.

G. Bierman, E. Meijer, and M. Torgersen. Adding dynamic types to C#. In *European Conference on Object-Oriented Programming*, ECOOP'10. Springer-Verlag, 2010.

G. M. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 257–281, 2014.

G. Bracha and L. Bak. Dart, a new programming language for structured web programming. Presentation at GOTO conference, Oct. 2011.

O. Chitil. Practical typed lazy contracts. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 67–76, New York, NY, USA, 2012. ACM.

C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 215–226, New York, NY, USA, 2011. ACM.

C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Complete monitors for behavioral contracts. In *ESOP*, 2012.

T. Disney, C. Flanagan, and J. McCarthy. Temporal higher-order contracts. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 176–188, New York, NY, USA, 2011. ACM.

ECMA. *Dart Programming Language Specification*, 2nd edition, December 2014.

L. Feigenbaum. Walkthrough: Dynamic programming in Visual Basic 10.0 and C# 4.0, Dec. 2008.
`http://blogs.msdn.com/b/vbteam/archive/2008/12/17/walkthrough-dynamic-programming-in-visual-basic-10-0-and-c-4-0-lisa-feigenbaum.aspx`.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, pages 48–59, Oct. 2002.

C. Flanagan. Hybrid type checking. In *Principles of Programming Languages (POPL)*, Jan. 2006.

M. Flatt and PLT. The Racket reference 6.0. Technical report, PLT Inc., 2014. `http://docs.racket-lang.org/reference/index.html`.

R. W. Floyd. Assigning meanings to programs. In *Symposium in Applied Mathematics*, volume 19, pages 19–32, 1967.

R. Garcia. Calculating threesomes, with blame. In *International Conference on Functional Programming (ICFP)*, pages 417–428, 2013.

M. Greenberg. *Manifest Contracts*. PhD thesis, University of Pennsylvania, Nov. 2013.

M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *Principles of Programming Languages (POPL) 2010*, 2010.

J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop (Scheme)*, pages 93–104, Sept. 2006.

A. Hejlsberg. Introducing TypeScript. Microsoft Channel 9 Blog, Oct. 2012.

F. Henglein. Dynamic typing: Syntax and proof theory. *Sci. Comput. Programming*, 22(3):197–230, 1994.

D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *Trends in Functional Programming (TFP)*, Apr. 2007.

D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23:167–189, 2010.

R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In M. Hagiya and P. Wadler, editors, *Proceedings of the Eighth International Symposium on Functional and Logic Programming (FLOPS 2006)*, volume 3945 of *Lecture Notes in Computer Science*, pages 208–225. Springer Berlin / Heidelberg, Apr. 2006.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *Principles of Programming Languages (POPL)*, pages 3–10, Jan. 2007.

B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *IFIP International Conference on Theoretical Computer Science*, pages 437–450, Aug. 2004.

D. Patterson, J. G. Politz, and S. Krishnamurthi. *Pyret Language Reference*. PLT, Brown University, 5.3.6 edition, 2014.
`http://www.pyret.org/docs/`.

A. Rastogi, N. Swamy, C. Fournet, G. M. Bierman, and P. Vekris. Safe & efficient gradual typing for TypeScript. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 167–180. ACM, 2015.

G. Richards, F. Z. Nardelli, and J. Vitek. Concrete types for TypeScript. In *European Conference on Object-Oriented Programming*, ECOOP'15. Springer-Verlag, 2015.

J. G. Siek and R. Garcia. Interpretations of the gradually-typed lambda calculus. In *Scheme and Functional Programming Workshop*, 2012.

J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop (Scheme)*, pages 81–92, Sept. 2006.

J. G. Siek and P. Wadler. Threesomes, with and without blame. In *Principles of Programming Languages (POPL)*, pages 365–376, 2010.

J. G. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In *European Symposium on Programming*, ESOP, pages 17–31, Mar. 2009.

J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland. Refined criteria for gradual typing. In *Summit on Advances in Programming Languages (SNAPL)*, May 2015.

T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In *Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, 2012.

N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. Bierman. Gradual typing embedded securely in javascript. In *ACM Conference on Principles of Programming Languages (POPL)*, Jan. 2014.

P. Thiemann. Session types with gradual typing. In M. Maffei and E. Tuosto, editors, *Trustworthy Global Computing - 9th International Symposium, TGC 2014, Rome, Italy, September 5-6, 2014. Revised Selected Papers*, volume 8902, pages 144–158. Springer, 2014.

S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *Principles of Programming Languages (POPL)*, pages 395–406, 2008. . URL `http://doi.acm.org/10.1145/1328438.1328486`.

S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium (DLS)*, pages 964–974, Oct. 2006.

P. Wadler. A complement to blame. In *Summit on Advances in Programming Languages (SNAPL)*, May 2015.

P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*, pages 1–16, Mar. 2009.

B. Yankov. Definitely typed repository, 2013.
`https://github.com/borisyankov/DefinitelyTyped`.