A Complement to Blame

Philip Wadler

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

Abstract

Contracts, gradual typing, and hybrid typing all permit less-precisely typed and more-precisely typed code to interact. Blame calculus encompasses these, and guarantees blame safety: blame for type errors always lays with less-precisely typed code. This paper serves as a complement to the literature on blame calculus: it elaborates on motivation, comments on the reception of the work, critiques some work for not properly attending to blame, and looks forward to applications. No knowledge of contracts, gradual typing, hybrid typing, or blame calculus is assumed.

1998 ACM Subject Classification D.3.2 Multiparadigm Languages

Keywords and phrases contracts, gradual typing, hybrid typing, blame calculus

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.309

1 Introduction

Findler and Felleisen [5] introduced two seminal ideas: higher-order *contracts* to dynamically monitor adherence to a type discipline, and *blame* to indicate which of two parties is at fault if the contract is violated. Siek and Taha [17] introduced gradual types to integrate untyped and typed code, while Flanagan [6] introduced hybrid types to integrate simple types with refinement types. Both used similar source languages, similar target languages with explicit casts, and similar translations between source and target. Both depended upon contracts, but neither used blame.

Motivated by the similarities between gradual and hybrid types, Wadler and Findler [25] introduced blame calculus, which unifies the two by encompassing untyped, simply-typed, and refinement-typed code. As the name indicates, it also restores blame, which enables a proof of *blame safety*: blame for type errors always lays with less-precisely typed code – "well-typed programs can't be blamed". Blame safety may seem obvious (of course the blame lies with the less-precisely typed code!), so why bother with a proof? Because programming language researchers have learned that formal statement and proof help guide sound language design.

Findler and Felleisen [5] introduce blame, but give no general results on when blame can or cannot arise. Tobin-Hochstadt and Felleisen [22] and Matthews and Findler [14] both offer proofs that when integrating untyped and typed code, blame must always lay with the untyped code; each requires a sophisticated proof based on operational equivalence. Siek and Taha [17] and Flanagan [6] say nothing about blame safety, since they ignore blame. Wadler and Findler [25] generalise blame safety to levels of precision (untyped, simply-typed, and refinement typed), and introduce a new proof technique that avoids operational equivalence, instead using the standard progress and preservation approach to type soundness of Wright and Felleisen [27].

Arguably, gradual and hybrid types can have a stronger foundation if based on blame calculus (or some other formulation of blame). Subsequent work on gradual typing does

1st Summit on Advances in Programming Languages (SNAPL'15).
Eds.: Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett; pp. 309–320

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

exactly this; Siek is a coauthor of several of the blame papers cited below. However, further work on hybrid types continues to ignore blame.

Section 2 summarises blame calculus. Section 3 critiques two studies that relate contracts to hybrid types. Section 4 makes some observations about notation. Section 5 discusses related work, notably in industry. Section 6 speculates on who might benefit from blame.

2 Blame calculus in a nutshell

Blame calculus is developed in a series of papers. Wadler and Findler [25] introduces the calculus, which is based on casts, and applies it to gradual and refinement types. Siek and Wadler [18] observes that every cast can be decomposed into an upcast and a downcast, and applies this to derive a new space-efficient implementation. Ahmed et al. [1] extends blame calculus to polymorphic types; remarkably, an untyped program cast to a polymorphic type is guaranteed to satisfy relational parametricity – "theorems for free" [15, 23]. Siek et al. [16] relates the blame calculus to two new calculi, one inspired by the coercion calculus of Henglein [11] and one inspired by the space-efficient calculus of Herman et al. [12], and shows that translations between the three are fully abstract.

From untyped to typed. The blame calculus provides a framework for integrating less-precisely and more-precisely typed programs. One scenario is that we begin with a program in an untyped language that we wish to convert to a typed language.

Here is an untyped program.

$$
\begin{array}{l}
\lceil \texttt{let}\ x = 2 \\
\ \texttt{let}\ f = \lambda y.\, y + 1\ \texttt{in} \\
\ \texttt{let}\ h = \lambda g.\, g\,(g\,x)\ \texttt{in} \\
\ h\ f \rceil
\end{array}
$$

The term evaluates to $\lceil 4 \rceil : \star$.

By default, our programming language is typed, so we indicate untyped code by surrounding it with ceiling brackets, $\lceil \cdot \rceil$. Untyped code is typed code where every term has the dynamic type, $\star$. Dana Scott [19] and Robert Harper [9] summarise this characterisation with the slogan "untyped is uni-typed".

As a matter of software engineering, when we add types to our code we may not want to do so all at once. Of course, it is trivial to rewrite a four-line program. However, the technique described here is intended to apply also when each one-line definition is replaced by a thousand-line module. We manage the transition between untyped and typed code by a new construct with an old name, "cast".

Here is our program, mostly typed, with one untyped component cast to a type.

$$
\begin{array}{l}
\texttt{let}\ x = 2 \\
\texttt{let}\ f = \lceil \lambda y.\, y + 1 \rceil : \star \Longrightarrow^{p} \texttt{Int} \to \texttt{Int} \\
\texttt{let}\ h = \lambda g{:}\texttt{Int} \to \texttt{Int}.\, g\,(g\,x) \\
\texttt{in}\ h\ f
\end{array}
$$

The term evaluates to $4 : \texttt{Int}$.

In general, a cast has the form $M : A \Longrightarrow^{p} B$, where M is a term, A and B are types, and p is a blame label. Term M has type A, while the type of the cast as a whole is B. The blame label p is used to indicate where blame lies in event of a failure. For instance, if we

replace

$$\lceil \lambda y.\, y > 0 \rceil \qquad \text{by} \qquad \lceil \lambda y.\, y + 1 \rceil$$

then the term evaluates to `blame` p, indicating that the party at fault is the term contained in the cast labeled p. Cast failures may not be immediate: in this case, the cast fails only when the function contained in the cast is applied and returns a boolean rather than an integer.

Here is our program, mostly untyped, but with one typed component cast to untyped.

$$\begin{array}{l}\texttt{let}\ x = \lceil 2 \rceil \\ \texttt{let}\ f = (\lambda y{:}\texttt{Int}.\, y + 1) : \texttt{Int} \to \texttt{Int} \Longrightarrow^{p} \star \\ \texttt{let}\ h = \lceil \lambda g.\, g\ (g\ x) \rceil \\ \texttt{in}\ \lceil h\ f \rceil\end{array}$$

Variables bound to untyped code or appearing in an untyped region must have type $\star$. The term evaluates to $\lceil 4 \rceil : \star$.

Blame in a cast may lie either with the *term contained* in the cast or the *context containing* the cast. For instance, if we replace

$$\lceil \lambda g.\, g\ 2 \rceil \qquad \text{by} \qquad \lceil \lambda g.\, g\ \texttt{true} \rceil$$

then the term evaluates to `blame` $\overline{p}$, indicating that the party at fault is the context containing the cast labeled p. The complement of blame label p is written $\overline{p}$. Complementation is involutive, $\overline{\overline{p}} = p$.

When blame lies with the *term contained* in the cast (that is, a term with a cast labeled p evaluates to `blame` p), we say we have *positive blame*, and when blame lies with the *context containing* the cast (that is, a term with a cast labeled p evaluates to `blame` $\overline{p}$), we say we have *negative blame*.

In our examples, positive blame arose when casting from untyped to typed, and negative blame arose when casting from typed to untyped. In both cases blame lies on the less precisely typed side of the cast. This is no coincidence, as we will see.

From simply typed to refinement typed. Refinement types are base types that additionally satisfy a predicate. For instance, $(x : \texttt{Int})\{x \geq 0\}$ is the type of integers that are greater than or equal to zero, which we abbreviate `Nat`. In general, refinement types have the form $(x : \iota)\{M\}$ where ι is a base type, and M is a term of type `Bool`.

Just as casts take untyped code to typed, they also can take simply-typed code to refinement-typed. For example

$$\begin{array}{l}\texttt{let}\ x = 2 : \texttt{Int} \Longrightarrow^{q} \texttt{Nat} \\ \texttt{let}\ f = (\lambda y{:}\texttt{Int}.\, y + 1) : \texttt{Int} \to \texttt{Int} \Longrightarrow^{p} \texttt{Nat} \to \texttt{Nat} \\ \texttt{let}\ h = \lambda g{:}\texttt{Nat} \to \texttt{Nat}.\, g\ (g\ x) \\ \texttt{in}\ h\ f\end{array}$$

returns $4 : \texttt{Nat}$.

As before, casts can fail. For instance, if we replace $\lambda y{:}\texttt{Int}.\, y + 1$ by $\lambda y{:}\texttt{Int}.\, y - 2$ then the term evaluates to `blame` p, indicating that the party at fault is the term contained in the cast labeled p. In this case, the cast fails when the function contained in the cast returns -2, which fails the dynamic test applied when casting `Int` to `Nat`.

Analogous to previously, positive blame arises when casting from simply-typed to refinement-typed, and negative blame arises when casting from refinement-typed to simply-typed. As before, in both cases blame lies on the less precisely typed side of the cast.

How blame works. The blame calculus consists of typed lambda calculus plus casts $M : A \Longrightarrow^p B$ and failure $\texttt{blame}\ p$. Let L, M, N range over terms, V, W range over values, A, B range over types, and p, q range over blame labels.

We used the construct $\lceil \cdot \rceil$ to indicate untyped code, but it is easily defined in terms of the other constructs. For instance, $\lceil \lambda g.\, g\ 2 \rceil$ translates to

$$(\lambda g{:}\star\,.\,(g : \star \Longrightarrow^{p_1} \star \to \star)\ (2 : \texttt{Int} \Longrightarrow^{p_2} \star)) : \star \to \star \Longrightarrow^{p_3} \star$$

where p_1, p_2, p_3 are fresh blame labels introduced by the translation.

The most important reduction rule of the blame calculus is the (Wrap) rule, which descends directly from the handling of higher-order contracts in Findler and Felleisen [5].

$$(V : A \to B \Longrightarrow^p A' \to B')\ W \quad \longrightarrow \quad (V\ (W : A' \Longrightarrow^{\overline{p}} A)) : B \Longrightarrow^p B' \qquad \text{(Wrap)}$$

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. To allocate blame correctly, the blame label on the cast of the domain is complemented, corresponding to the fact that the argument W is supplied by the context. In the cast for the domain the types are swapped (A' to A) and the blame label is complemented ($\overline{p}$), while in the cast for the range the order is retained (B to B') and the blame label is unchanged (p), corresponding to the fact that function types are contravariant in the domain and covariant in the range.

It is straightforward to establish type safety, using preservation and progress as usual.

▶ **Theorem 1** (Type safety)**.**
1. *If* $\vdash M : A$ *and* $M \longrightarrow N$ *then* $\vdash N : A$*.*
2. *If* $\vdash M : A$ *then either*
   a. *there exists a term* N *such that* $M \longrightarrow N$*, or*
   b. *there exists a value* V *such that* $M = V$*, or*
   c. *there exists a label* p *such that* $M = \texttt{blame}\ p$*.*

Here type safety is a weak result, as it cannot rule out the possibility that a term reduces to the form $\texttt{blame}\ p$, which is akin to a type error. How to guarantee blame cannot arise in certain circumstances is the subject of the next section.

Blame safety. The statement and proof of blame safety is relatively straightforward. However, it requires four different subtyping relations: $<:$, $<:^+$, $<:^-$, and $<:_n$, called ordinary, positive, negative, and naive subtyping respectively.

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

We omit the details of the definitions here; see Wadler and Findler [25]. As is standard, $<:$, $<:^+$, and $<:^-$ are contravariant in the domain and covariant in the range of function types, while $<:_n$ is (perhaps surprisingly) covariant in both the domain and the range.

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

▶ **Lemma 2** (Tangram)**.**
1. $A <: B$ *iff* $A <:^+ B$ *and* $A <:^- B$*.*
2. $A <:_n B$ *iff* $A <:^+ B$ *and* $B <:^- A$*.*

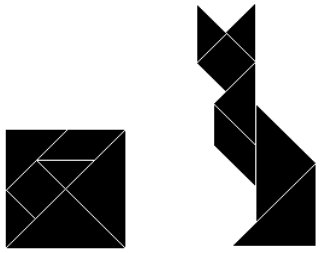

Figure 1 Tangram as metaphor.

I named the result by analogy to the well-known tangram puzzle, where a square decomposes into parts that recombine into a different shape (Figure 1).

We can now characterise under what circumstances reduction of a term can never blame p.

▶ **Definition 3.** Term M is *safe* for label p, written M safe p, if every cast in M of the form $A \Longrightarrow^p B$ satisfies $A <:^+ B$, and every cast of the form $A \Longrightarrow^q B$ where $q = \overline{p}$ satisfies $A <:^- B$.

If M safe p then $M \not\longrightarrow^* \texttt{blame}\ p$. This is established via a variant of preservation and progress.

▶ **Theorem 4** (Blame Safety)**.**
1. *If* M safe p *and* $M \longrightarrow N$ *then* N safe p.
2. *If* M safe p *then* $M \not\longrightarrow \texttt{blame}\ p$.

The proof is by straightfoward case analysis of each of the reduction rules, and involves checking that the reduction rules preserve the desired invariants. For instance, for (Wrap),

$$(V : A \to B \Longrightarrow^p A' \to B')\, W \longrightarrow (V\ (W : A' \Longrightarrow^{\overline{p}} A)) : B \Longrightarrow^p B'$$

we must check that if $A \to B <:^+ A' \to B'$ then $A' <:^- A$ and $B <:^+ B'$; and similarly with $<:^+$ and $<:^-$ interchanged. In fact, this easily follows from the definitions of $<:^+$ and $<:^-$. While the basic proof of blame safety is mine, the formulation in terms of preservation and progress is due to Robby Findler.

Let $\mathcal{C}$ range over contexts, terms containing a hole. Assume p and $\overline{p}$ do not appear in context $\mathcal{C}$ or term M. Combining the Tangram Lemma with Blame Safety we can characterise casts that never fail, and show that when casting between a less precise type and a more precise type that the more-precisely-typed side of the cast is never to blame.

▶ **Corollary 5** (Blame Theorem)**.**
1. *If* $A <: B$ *then* $\mathcal{C}[M : A \Longrightarrow^p B] \not\longrightarrow^* \texttt{blame}\ p$ *and* $\mathcal{C}[M : A \Longrightarrow^p B] \not\longrightarrow^* \texttt{blame}\ \overline{p}$.
2. *If* $A <:_n B$ *then* $\mathcal{C}[M : A \Longrightarrow^p B] \not\longrightarrow^* \texttt{blame}\ p$.
3. *If* $B <:_n A$ *then* $\mathcal{C}[M : A \Longrightarrow^p B] \not\longrightarrow^* \texttt{blame}\ \overline{p}$.

The last two points are summarised by the motto "well-typed programs can't be blamed".

For instance, in the earlier examples, the untyped term in a typed context can never blame the context because $\star <:^- \texttt{Int} \to \texttt{Int}$, while the typed term in an untyped context can never blame the term because $\texttt{Int} \to \texttt{Int} <:^+ \star$, and both of these follow from (and indeed are equivalent to) $\texttt{Int} \to \texttt{Int} <:_n \star$.

The result for ordinary subtyping is not nearly so useful as the results for naive subtyping. Cast between ordinary subtypes, such as $\texttt{Int} \to \texttt{Nat} <: \texttt{Nat} \to \texttt{Int}$, rarely arise, while

casts between naive subtypes, such as $\texttt{Int} \to \texttt{Int} <:_n \texttt{Nat} \to \texttt{Nat}$ arise often. Despite this, the literature often presents the result for ordinary subtyping, while the results for naive subtyping are ignored.

An encouraging discovery and a disappointing reception. I was amazed and delighted to discover the Tangram Lemma. When defining the four relations, I had in mind that $<:$ should imply $<:^+$ and $<:^-$, and that $<:_n$ should imply $<:^+$ and the converse of $<:^-$. But it was a complete surprise to find that the implications I had in mind could be replaced by equivalences.

The Tangram Lemma convinced me that there must be mathematical substance to the Blame Calculus. I felt less like I was inventing, and more like I was discovering. Interestingly, I know of no analogues of the Tangram Lemma in other areas of mathematics, where two relations refactor into two other relations.

So far as I can tell, I am alone in my enthusiasm for the Tangram Lemma and the Blame Theorem. Others don't find them as exciting as I do. Perhaps one reason why is because the Tangram Lemma is stated in terms of four subtyping relations, which may be three too many to easily absorb.

I'm particularly disturbed that researchers tend to formulate systems without complement, which renders the Blame Theorem impossible to formulate or prove, and which discards useful debugging information, as described in the next section.

3 To complement or not to complement?

Gronski and Flanagan [8] relate the contracts of Findler and Felleisen [5] to the hybrid types of Flanagan [6]; and Greenberg et al. [7] build on these results.

Interestingly, the cited works relate contracts where blame is complemented to casts where blame is not complemented. This section begins by presenting a variation of their results, using our casts with complement; and then compares it to their version, where casts are not complemented.

I've translated the notation of the cited papers to the notation used here. Among other things, "casts with complement" here corresponds to "casts with two blame labels" in those papers; and "casts without complement" corresponds to "casts with one blame label"; see "Two against one" in Section 4.

We write contracts of Findler and Felleisen [5] as $M \mathbin{@^p} A$, indicating term M is subject to contract A with blame label p. The equivalent of the (Wrap) rule is

$$(V \mathbin{@^p} A \to B)\, W \quad \longrightarrow \quad (V\, (W \mathbin{@^{\overline{p}}} A)) \mathbin{@^p} B.$$

Observe the blame label p is complemented for the domain but not the range, just as with (Wrap) in the blame calculus.

Gronski and Flanagan [8] give a map ϕ from contracts to casts. It is defined using a supplementary function that removes refinements to yield ordinary types.

$$\begin{aligned} \lfloor (x : \iota)\{M\} \rfloor &= \iota \\ \lfloor A \to B \rfloor &= \lfloor A \rfloor \to \lfloor B \rfloor \end{aligned}$$

The key equation of the map from contracts to casts is

$$\phi(M \mathbin{@^p} A) \quad = \quad \phi(M) : \lfloor A \rfloor \Longrightarrow^p \phi(A) \Longrightarrow^p \lfloor A \rfloor$$

(Here $M : A \Longrightarrow^p B \Longrightarrow^q C$ abbreviates $(M : A \Longrightarrow^p B) : B \Longrightarrow^q C$.) The other equations are homomorphic; and $\phi(A)$ applies ϕ to each term M in a refined type in A. The explanation

of this translation is straightfoward. The type system for contracts requires that a term M with contract A must have type $\lfloor A \rfloor$, and after applying the contract it must have the same type. Hence the two casts ensure the terms have the proper type, while the application of $\phi(A)$ in the middle enforces the conditions imposed by the contract A.

However, the actual mapping given in Gronski and Flanagan [8] and Greenberg et al. [7] use a variant of casts that does not complement the blame label, which we indicate by adding a subscript `no`. The system is essentially identical to ours, save that the equivalent of (Wrap) does not complement blame labels in the domain.

$$(V : A \to B \Longrightarrow^{p}_{\mathsf{no}} A' \to B')\, W \quad \longrightarrow \quad (V\, (W : A' \Longrightarrow^{p}_{\mathsf{no}} A)) : B \Longrightarrow^{p}_{\mathsf{no}} B'$$

The key equation of the translation is now

$$\phi(M\ @^{p}\ A) \quad = \quad \phi(M) : \lfloor A \rfloor \Longrightarrow^{p}_{\mathsf{no}} \phi(A) \Longrightarrow^{\overline{p}}_{\mathsf{no}} \lfloor A \rfloor$$

The only difference in the equation is that the second cast, which was previously labelled p, is now labelled $\overline{p}$.

The reason why the version without complement works can be easily understood by an application of the Blame Theorem. It is easy to see that a refinement type A is more precise than the equivalent simple type $\lfloor A \rfloor$. Hence, in $\phi(M) : \lfloor A \rfloor \Longrightarrow^{p} \phi(A) \Longrightarrow^{p} \lfloor A \rfloor$ the leftmost cast can only blame p while the rightmost cast can only blame $\overline{p}$. This is why using casts without complement yields the same answer, by simply introducing the complement instead as part of the translation.

Gronski and Flanagan [8] observe that contracts with complement are potentially more expressive than casts without complement. The mapping ϕ shows that expressiveness is preserved, and they take this as a justification for limiting themselves to casts without complement. Note that, ironically, we require the Blame Theorem, which depends upon casts with complement, in order to show why the casts without complement are adequate for translating contracts with complement.

Greenberg et al. [7] introduce the name "latent" for the system with contracts and "manifest" for the system with casts. They note that either system can be formulated with or without complement, arguing as follows. [I have updated their notation to match this paper.]

> This difference is not fundamental, but rather a question of the pragramatics of assigning responsibility: both latent and manifest systems can be given more or less rich algebras of blame. Informally, a function contract check $f\ @^{p}\ A \to B$ divides responsibility for f's behaviour between its body and its environment: the programming is saying "If f is ever applied to an argument that doesn't pass A, I refuse responsibility (`blame` $\overline{p}$), whereas if f's result for good arguments doesn't satisfy B, I accept responsibity (`blame` p)." In a manifest system, the programmer who writes $f : A \to B \Longrightarrow^{p}_{\mathsf{no}} A' \to B'$ is saying "Although all I know statically about f is that its results satisfy B when it is applied to arguments satisfying A, I assert that it's OK to use it on arguments satisfying A' (because I believe that A' implies A) and that its results will always satisfy B' (because I believe that B implies B')." In the latter case, the programmer is taking responsibility for *both* assertions (so `blame` p makes sense in both cases), while the additional responsibility for checking that arguments satisfy A' will be discharged elsewhere (by another cast with a different label).

However, this throws away useful information to no benefit. If one uses complement with manifest casts, then `blame` p tells us the claim that A' implies A is incorrect, while `blame` $\overline{p}$ tells us the claim that B implies B' is incorrect, a huge aid to debugging.

Whether one is interested in systems with complement depends on the the results one finds of interest. If one has a sophisticated notion of subtyping, where $(x : A)\{M\} <: (x : A)\{N\}$ if M implies N for all x, then $A \to B <: A' \to B'$ holds exactly when A' implies A and B implies B'. The system without complement is adequate if one's interest is only in clause 1 of the Blame Theorem (Corollary 5), showing that a cast $A \Longrightarrow^p B$ never fails if $A <: B$, while the system with complement is essential if one's interest is in clause 2 or clause 3, showing that a cast $A \Longrightarrow^p B$ never blames p if $A <:_n B$ and never blames $\overline{p}$ if $B <:_n A$.

I argue it is better to formulate a system with complement. Complement yields useful information for debugging; and without complement there is no way to formulate Blame Safety or the Blame Theorem, and hence to assure that blame never lies with the more-precisely-typed side of a cast. If for some reason one wishes to ignore the distinction between p and $\overline{p}$ (for instance, because we want to assign responsibility to the same programmer in either case), that is easy to do. But one should make the distinction and then ignore it; if one fails to make the distinction from the start, then Blame Safety and the Blame Theorem become impossible to prove. Above I've sketched a reformulation of Gronski and Flanagan [8] and Greenberg et al. [7] in a system with complement, and it would be nice to see this worked out in detail.

4 Notation

As observed by Whitehead [26, Chapter V], notation can crucially effect our thought. I expend much effort on choice of notation, yet often find I don't get it right the first time. This section reviews changes to notation for blame during a decade.

Two against one. Findler and Felleisen [5] indicate blame using a pair of labels, p and n, standing for positive and negative blame; their version of (Wrap) swaps p, n so the contract for the domain is labeled n, p. Wadler and Findler [25] indicate blame with a single label p; their (Wrap) complements p so the contract for the domain is labeled $\overline{p}$. The change is at best a small step forward, but nonetheless I am proud of it. Fewer variables is better, and the abstract notion of complement may open the way to further generalisation.

Casts. Siek and Taha [17] write $\langle B \rangle\, M$ to cast term M to type B. The type A of M is is easily inferred, and is omitted to save space. But a consequence is that their equivalent of (Wrap) must mix type inference and reduction. They use no blame labels.

Flanagan [6] writes $\langle A \rhd B \rangle^p\, M$ to cast term M from type A to type B. For me, the idea that one might explicitly list both source and target types of the cast was an eye opener. While too prolix for practice, it clarifies the core calculus by highlighting the contravariant and covariant treatment of the domain and range in (Wrap). As explained in Section 3, Flanagan's interpretation of blame label p differs from ours, in that it lacks complement.

Wadler and Findler [25] write $\langle B \Leftarrow A \rangle^p\, M$ to cast term M from type A to type B, a variation of Flanagan [6] that allows reading uniformly from right to left. Blame label p now supports complement.

Ahmed et al. [1] write $M : A \Longrightarrow^p B$ to cast term M from type A to type B. I remember the day we introduced the new notation: Amal Ahmed, Robby Findler, and myself were hosted by Jacob Matthews at Google's Chicago office, where we had an unused floor to ourselves. I didn't think direction of reading was a big deal, but I had underestimated the extent to which my reading of prose influenced my reading of formulas. When we reversed right-to-left to left-to-right it was as if a weight lifted from my brow!

The new notation suggests a natural abbreviation, replacing $(M : A \Longrightarrow^p B) : B \Longrightarrow^q C$ by $M : A \Longrightarrow^p B \Longrightarrow^q C$. Again, the abbreviation helped more than I might have imagined, rendering rules and calculations far easier to read.

Refinement types and dependent function types. I also want to draw attention to an improvement in notation that had nothing to do with me.

In standard notation (found in Flanagan [6] and many others), refinement types are written $\{x : A \mid M[x]\}$, where x is a variable of type A, and $M[x]$ is a boolean-valued term that may refer to x; and dependent function types are written $x : A \to B[x]$, where x is a variable of type A, and $B[x]$ is a type that may refer to x. For instance,

$$x : \{x : \mathtt{Int} \mid x \geq 0\} \to \{y : \mathtt{Int} \mid y \geq x\}$$

is the dependent function type that accepts a natural, and returns an integer greater than its argument. A drawback of this notation is duplication of x, bound once in the refinement type and again in the dependent function type.

An improved notation is suggested by Swamy et al. [20] for F*, where refinement types are written $(x : A)\{M[x]\}$, and dependent function types are written as before. For instance,

$$x : (x : \mathtt{Int})\{x \geq 0\} \to (y : \mathtt{Int})\{y \geq x\}$$

is the dependent function type that accepts a natural, and returns an integer greater than its argument. This notation lends itself naturally to the abbreviation

$$(x : \mathtt{Int})\{x \geq 0\} \to (y : \mathtt{Int})\{y \geq x\}$$

which avoids the annoying duplication of x.

Interplay with implementation. The choice between target-alone or source-and-target casts is not only about concision versus clarity, but also relates to implementation. Simply-typed lambda calculus is typically written with types in lambda abstractions, $\lambda x{:}A.\, N$, which ensures each closed term has a unique type. But types play no part in the reduction rules, and so can be erased before evaluation. As noted above, target-alone casts must mix type inference and reduction in (Wrap), so erasure is not an option. However, source-and-target casts need no type inference in (Wrap), and so support erasure. To be precise, types may be erased in lambda abstractions (and related constructs), but source and target types cannot be erased from casts.

Hence, a source language should use target-alone casts for concision, and translate to an intermediate language with source-and-target casts to aid implementation. So far as I know, this observation has not been written down previously. I think one reason why is that authors are reluctant to make a claim – even if the point is as obvious as the one made here – without either writing out the relevant bit of the theory or building an actual implementation, both of which can be costly.

5 Practice and theory

Nothing is so practical as a good theory.

C#. C# introduces a type `dynamic`, which plays a role almost identical to our $\star$ [3]. As with gradual typing, a translation introduces casts to and from `dynamic` where needed. However, the introduced casts are first-order; higher-order casts must be written by hand, implementing the equivalent of (Wrap).

TypeScript. TypeScript allows the programmer to specify in an `interface` declaration types for a JavaScript module or library supplied by another party [10]. The DefinitelyTyped repository contains over 150 such declarations for a variety of popular JavaScript libraries [28]. Interestingly, TypeScript is not concerned with type soundness, which it does not provide, but instead with exploiting types to provide effective prompts in Visual Studio, for instance to populate a pulldown menu with methods that might be invoked at a given point.

TS*. TS* is a variant of TypeScript which ensures type safety by assigning run-time type information (RTTI) to some JavaScript objects [21]. One desirable property of gradual typing is a *type guarantee*: any untyped program can be cast to any type so long as its semantic properties correspond to that type. The embedding of untyped programs into blame calculus satisfies the type guarantee, while the use of RTTI violates it. Assessing the tradeoffs between blame calculus and TS* remains interesting future work.

TypeScript TNG. In TypeScript, the information supplied by `interface` declarations is taken on faith; failures to conform to the declaration are not reported. Microsoft has funded myself and a PhD student to build a tool, TypeScript TNG, that uses blame calculus to generates 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. Our design satisfies the *type guarantee* described above.

Initial results 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 this difficult in problematic cases; we need to determine to what extent these cases are an issue in practice.

Wide-spectrum language. Programming languages offer a range of type structures. Here are four of the most important, listed from weakest to strongest:

1. Dynamic types, as in Racket, Python, and JavaScript.
2. Polymorphic types, as in ML, Haskell, and F#.
3. Refinement types, as in Dependent ML and F7.
4. Dependent types, as in Coq, Agda, and F*.

A variant of blame calculus augmented with polymorphic types, refinement types, and dependent types might be used to mediate between each of these systems. Kinds or effects could delimit where programs may loop and where they are guaranteed to terminate, as required for the strongest form of dependent types. We hypothesise that a wide-spectrum language will increase the utility of the different styles of typing, by allowing dynamic checks to be used as a fallback when static validation is problematic.

6 Who needs gradual types?

I always assumed gradual types were to help those poor schmucks using untyped languages to migrate to typed languages. I now realise that I am one of the poor schmucks. My recent research involves session types, a linear type system that declares protocols for sending messages along channels. Sending messages along channels is an example of an effect. Haskell uses monads to track effects [24], and a few experimental languages such as Links [4], Eff [2],

and Koka [13] support effect typing. But, by and large, every programming language is untyped when it comes to effects. To encourage migration from legacy code to code with effect types, such as session types, some form of gradual typing may be essential.

Acknowledgements. My thanks to Shriram Krishnamurthi, Sam Lindley, Don Sannella, Jeremy Siek, and the staff of the Western General Hospital. This work is supported by EPSRC Programme Grant EP/K034413/1.

References

1 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *Principles of Programming Languages (POPL)*, pages 201–214, 2011.

2 Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10(4), 2014.

3 Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C#. In *European Conference on Object-Oriented Programming*, ECOOP'10. Springer-Verlag, 2010.

4 Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*, pages 266–296. Springer, 2007.

5 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, pages 48–59, October 2002.

6 Cormac Flanagan. Hybrid type checking. In *Principles of Programming Languages (POPL)*, January 2006.

7 Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Principles of Programming Languages (POPL)*, 2010.

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

9 Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2012.

10 Anders Hejlsberg. Introducing TypeScript. Microsoft Channel 9 Blog, October 2012.

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

12 David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23:167–189, 2010.

13 Daan Leijen. Koka: Programming with row polymorphic effect types. In *Mathematically Structured Functional Programming (MSFP)*, pages 100–126, 2014.

14 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *Principles of Programming Languages (POPL)*, pages 3–10, January 2007.

15 John Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing*, pages 513–523. North-Holland, 1983.

16 Jeremy Siek, Peter Thiemann, and Philip Wadler. Blame and coercions: Together again for the first time. Technical report, University of Edinburgh, 2014.

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

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

19 Richard Statman. A local translation of untyped [lambda] calculus into simply typed [lambda] calculus. Technical report, Carnegie-Mellon University, 1991.

20 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *International Conference on Functional Programming (ICFP)*, September 2011.

21 Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in javascript. In *Principles of Programming Languages (POPL)*, January 2014.

22 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium (DLS)*, pages 964–974, October 2006.

23 Philip Wadler. Theorems for free. In *Functional Programming Languages and Computer Architecture (FPCA)*, September 1989.

24 Philip Wadler. Comprehending monads. *Mathemetical Structures in Computer Science*, 2(4):461–493, 1992.

25 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*, pages 1–16, March 2009.

26 Alfred North Whitehead. *An Introduction to Mathematics*. Henry Holt and Company, 1911.

27 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

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