Blame for all

Amal Ahmed[1] and Robert Bruce Findler[2] and Jacob Matthews[3] and Philip Wadler[4]

[1] Toyota Technological Institute at Chicago
[2] Northwestern University
[3] Google
[4] University of Edinburgh

Abstract. We present a language that integrates statically and dynamically typed components, similar to the gradual types of Siek and Taha (2006), and extend it to incorporate parametric polymorphism. Our system permits a dynamically typed value to be cast to a polymorphic type, with the type enforced by dynamic sealing along the lines proposed by Pierce and Sumii (2000), Matthews and Ahmed (2008), and Neis, Dreyer, and Rossberg (2009), in a way that ensures all terms satisfy relational parametricity. Our system includes a notion of blame, which allows us to show that when more-typed and less-typed portions of a program interact, that any type failures are due to the less-typed portion.

1 Introduction

The long tradition of work that integrates static and dynamic types includes the *coercions* of Henglein (1994), the *contracts* of Findler and Felleisen (2002), the *gradual types* of Siek and Taha (2006), the *migratory types* of Tobin-Hochstadt and Felleisen (2006), the *hybrid types* of Flanagan (2006), the *dynamic dependent types* of Ou, Tan, Mandelbaum, and Walker (2004), the *multi-language* programming of Matthews and Findler (2007), and the *blame calculus* of Wadler and Findler (2009).

A unifying theme in much of this work is to use casts to mediate between static and dynamic types. Typically, casts are introduced by compiling to an intermediate language, while the blame calculus suggests such casts can profitably be treated as a source language. The main technical innovation is to assign positive and negative blame (to either the term *contained* in the cast or the context *containing* the cast), with associated notions of positive and negative subtype. These permit Wadler and Findler to state and give a straightforward proof of the Blame Theorem, which ensures that when a program goes wrong, blame lies with the less-precisely-typed side of a cast; previous work ignores such a result or requires a complex statement and proof.

Here we show how to extend a fragment of the blame calculus to incorporate polymorphism. For simplicity, our fragment includes base types and dynamic type, as found in gradual types, but omits subset types, as found in hybrid types. Our system permits a value of dynamic type to be cast to a polymorphic type, using dynamic sealing to ensure that values of polymorphic type satisfy *relational parametricity* (Reynolds, 1983; Wadler, 1989). For instance, every function of type $\forall X.\, X \to X$ must be either the identity function or the constant bottom function, and this holds true even for dynamic code cast to this static type.

We present terms, types, and reductions for our language, extend our previous characterizations of compatibility and of ordinary, positive, negative, and naive subtypes to include polymorphic types, and state and prove an appropriate Blame Theorem.

We have demonstrated that our system satisfies relational parametricity using step-indexed Kripke logical relations similar to those devised by Neis et al. (2009); both that work and ours refine Matthews and Ahmed (2008). We omit the development due to a lack of space.

Our definition of subtyping for polymorphic types is arguably too restrictive. In the last section, we present an improved definition of subtyping and present a conjecture and a relevant counter-example.

Dynamic sealing to enforce parametricity has long been understood in folklore. Sealing for data abstraction goes back at least to Morris (1973). Cryptographic sealing for parametricity was introduced by Pierce and Sumii (2000). Extending casts to include seals, while demonstrating relational parametricity, was first explored in the context of multi-language programming by Matthews and Ahmed (2008). A practical implementation for Scheme contracts was described by Guha, Matthews, Findler, and Krishnamurthi (2007). Recently, use of dynamic sealing to restore parametricity to a non-parametric language was demonstrated by Neis et al. (2009).

Our work most closely resembles that of Matthews and Ahmed (2008) and Neis et al. (2009), which both give relational parametricity results based on step-indexed logical relations. The first uses separate static and dynamic languages; we use a single language in which each of these extremes is easily embedded. The second includes non-parametric operations, and describes wrappers that restore relational parametricity; we provide a language in which all terms satisfy parametricity. Both works use positive and negative wrappers to enforce sealing, and the second uses positive and negative logical relations; these appear closely related to our positive and negative subtyping. A precise understanding of the relationship would be an important subject for future work.

Sections 2–4 present in turn the type rules, reduction rules, and subtyping rules of our system, Section 5 presents the Blame Theorem, and Section 6 outlines an improved subtyping relation.

2 Types and terms

Figure 1 presents syntax and type rules for the blame calculus with polymorphic types. We let S, T range over types, C, D range over casts, and G, H range over grounds. A type is either a base type B, the dynamic type $\star$, a function type $S \to T$, or a polymorphic type $\forall X.\, T$. Base types B include booleans $\mathtt{B}$ and integers $\mathtt{I}$. A cast is the same as a type, but may also contain a *seal* $k(T)$. We write $|C|$ for the type erasure of cast C, which takes a seal $k(T)$ to T and all other constructors to themselves. We require that seals are always used at the same type: given two seals $k(T)$ and $k'(T')$, if $k = k'$ then $T = T'$. A ground G is either a base type B, the function type $\star \to \star$, or a seal $k(T)$. Each value of dynamic type $\star$ will be injected from a ground.

As usual, we write $\Gamma \vdash t : T$ if term t has type T in a type environment Γ. There are two new term forms. A cast term $\langle D \Leftarrow C \rangle^p\ s$ converts a term s from type $|C|$ to type $|D|$, where the casts C and D are required to be compatible (as defined below)

base type	B	$::= \mathtt{I} \mid \mathtt{B}$
ground	G, H	$::= B \mid \star \to \star \mid k(T)$
type	S, T	$::= \star \mid B \mid S \to T \mid \forall X.\, T \mid X$
cast	C, D	$::= \star \mid B \mid C \to D \mid \forall X.\, D \mid X \mid k(T)$
term	s, t, u	$::= \langle D \Leftarrow C \rangle^p\, s \mid s\ \mathtt{is}^p\ G \mid c \mid x \mid \lambda x : S.\, t \mid t\ s \mid \Lambda X.\, t \mid t\ S$
constant	c	$::= \mathtt{true} \mid \mathtt{false} \mid \cdots$
untyped term	M, N	$::= c \mid x \mid \lambda x.\, N \mid M\ N$

Type erasure $\boxed{|C| = T}$

$$|\star| = \star \qquad |\forall X.\, C| = \forall X.\, |C| \qquad |X| = X$$
$$|B| = B \qquad |C \to D| = |C| \to |D| \qquad |k(T)| = T$$

Type rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash s : |C| \qquad C \lhd D}{\Gamma \vdash \langle D \Leftarrow C \rangle^p\, s : |D|} \qquad \frac{\Gamma \vdash s : \star}{\Gamma \vdash s\ \mathtt{is}^p\ G : \mathtt{B}}$$

$$\frac{}{\Gamma \vdash c : \mathrm{ty}(c)} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x : S.\, t : S \to T} \qquad \frac{\Gamma \vdash t : S \to T \qquad \Gamma \vdash s : S}{\Gamma \vdash t\ s : T}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \Lambda X.\, t : \forall X.\, T}\ X \notin \Gamma \qquad \frac{\Gamma \vdash t : \forall X.\, T}{\Gamma \vdash t\ S : T[X := S]}$$

Compatibility $\boxed{C \lhd D}$

$$\frac{}{C \lhd \star} \qquad \frac{}{\star \lhd D} \qquad \frac{}{B \lhd B} \qquad \frac{}{X \lhd X} \qquad \frac{}{k(T) \lhd k(T)}$$

$$\frac{C' \lhd C \qquad D \lhd D'}{C \to D \lhd C' \to D'} \qquad \frac{C[X := \star] \lhd D}{\forall X.\, C \lhd D} \qquad \frac{C \lhd D}{C \lhd \forall X.\, D}\ X \notin C$$

Embedding dynamic types $\boxed{\lceil M \rceil = t}$

$$\lceil c \rceil = \langle \star \Leftarrow \mathrm{ty}(c) \rangle\, c \qquad \lceil \lambda x.\, N \rceil = \langle \star \Leftarrow \star \to \star \rangle\, (\lambda x : \star.\, \lceil N \rceil)$$
$$\lceil x \rceil = x \qquad \lceil M\ N \rceil = (\langle \star \to \star \Leftarrow \star \rangle\, \lceil M \rceil)\, \lceil N \rceil$$

Fig. 1. Type system

and where p is a blame label. An instance term $s\ \mathtt{is}^p\ G$ checks whether a value s of dynamic type is castable to ground G; it returns a boolean. The other term forms are standard.

We say cast C is compatible with cast D if some value of type $|C|$ can be converted to a value of type $|D|$, and write $C \lhd D$. In previous work, compatibility is symmetric,

but here it is not. For instance a cast from $\forall X.\, X \to X$ to $\mathtt{I} \to \mathtt{I}$ is permitted, as it amounts to instantiating the polymorphic type, while a cast from $\mathtt{I} \to \mathtt{I}$ to $\forall X.\, X \to X$ is not permitted, as a function on integers may not apply to other types. However, sometimes compatibility is symmetric. For instance a cast from $\forall X.\, X \to X$ to $\star \to \star$ is permitted, as it amounts to instantiating the polymorphic type, while a cast from $\star \to \star$ to $\forall X.\, X \to X$ is also permitted, since a function on dynamics may apply to all types.

Type $\star$ is compatible with any type, and every base type, type variable, and seal is compatible with itself. Two function types are compatible if their domains and ranges are compatible—note that, as with subtyping rules, the comparison is contravariant in the domain and covariant in the range. In previous work, compatibility is covariant in both the domain and range of a function, but here we must introduce contravariance because compatiblity is no longer symmetric.

A polymorphic type is compatible with a given type if *some* instance of the polymorphic type is compatible with the given type, while a given type is compatible with a polymorphic type if the given type is compatible with *every* instance of the polymorphic type. In particular, we have that $\forall X.\, C \lhd D$ if $C[X := \star] \lhd D$; this will be true whenever there is *some* type T such that $C[X := T] \lhd D$. In the other direction, we have that $C \lhd \forall X.\, D$ if $C \lhd D$ and X does not appear free in C (which we can always achieve by renaming the bound type variable X); this will be true whenever for *all* types T we have $C \lhd D[X := T]$. It is easy to check that these rules yield the compatibilities and incompatibilities described above:

$$\begin{array}{ll} \forall X.\, X \to X \lhd \mathtt{I} \to \mathtt{I}, & \mathtt{I} \to \mathtt{I} \not\lhd \forall X.\, X \to X, \\ \forall X.\, X \to X \lhd \star \to \star, & \star \to \star \lhd \forall X.\, X \to X. \end{array}$$

One consequence of the rules is that if $C \lhd D$ then $\forall X.\, C \lhd \forall X.\, D$. It is easy to check that $C \lhd D$ implies $C[X := \star] \lhd D$. We then have

$$\frac{\dfrac{C[X := \star] \lhd D}{\forall X.\, C \lhd D}}{\forall X.\, C \lhd \forall X.\, D}\ X \notin \forall X.\, C$$

The order in which the rules is applied is crucial: we first quantify on the left, and then since X is no longer free in $\forall X.\, C$ we may quantify on the right. An easy consequence of this is that every cast is compatible with itself.

The type system assigns a unique type to each term, and typability is decidable.

There is a standard embedding of untyped terms M into corresponding blame calculus terms $\lceil M \rceil$.

3 Reductions

Figure 2 defines additional syntax and the rules for reduction. We let v, w range over values. A value is either a constant, an abstraction, a type abstraction, or of dynamic type. The first four of these are standard. A value of dynamic type takes the form $\langle \star \Leftarrow G \rangle^p\ v$ where G is ground.

$$
\begin{array}{lll}
\text{values} & v, w & ::= c \mid \lambda x : S.\, t \mid \Lambda X.\, t \mid \langle \star \Leftarrow G \rangle^p\, v \\
\text{evaluation context} & E & ::= [\,] \mid E\ \mathtt{is}^p\ G \mid E\ t \mid v\ E \mid E\ S \mid \langle D \Leftarrow C \rangle^p\ E
\end{array}
$$

Substitution (key lines) $\boxed{s[X := k(T)]}$

$$
\begin{aligned}
(\langle D \Leftarrow C \rangle^p\ s)[X := k(T)] &= \langle D[X := k(T)] \Leftarrow C[X := k(T)] \rangle^p\ (s[X := k(T)]) \\
(\lambda x : S.\, t)[X := k(T)] &= \lambda x : S[X := T].\, (t[X := k(T)]) \\
(t\ S)[X := k(T)] &= (t[X := k(T)])\ (S[X := T])
\end{aligned}
$$

Reductions $\boxed{K;\ s \longrightarrow t;\ K' \quad s \longrightarrow \mathtt{blame}\ p}$

$$
\begin{aligned}
K;\ (\Lambda X.\, t)\ S &\longrightarrow t[X := k(S)];\ K \cup \{k\}, \quad \text{if } k \notin K \\
\langle D \Leftarrow \forall X.\, C \rangle^p\ v &\longrightarrow \langle D \Leftarrow C[X := \star] \rangle^p\ (v\ \star) \\
\langle \forall X.\, D \Leftarrow C \rangle^p\ v &\longrightarrow \Lambda X.\, \langle D \Leftarrow C \rangle^p\ v, \quad \text{if } X \notin C \text{ and } X \notin v \\
\langle \star \Leftarrow G \rangle^p\ v\ \mathtt{is}^q\ G &\longrightarrow \mathtt{true}, \quad \text{if } G \neq k(T) \\
\langle \star \Leftarrow G \rangle^p\ v\ \mathtt{is}^q\ H &\longrightarrow \mathtt{false}, \quad \text{if } G \neq H \text{ and } G \neq k(T) \\
\langle \star \Leftarrow k(T) \rangle^p\ v\ \mathtt{is}^q\ H &\longrightarrow \mathtt{blame}\ q \\
c\ v &\longrightarrow [\![c]\!](v) \\
(\lambda x.\, t)\ v &\longrightarrow t[x := v] \\
\langle C' \to D' \Leftarrow C \to D \rangle^p\ v &\longrightarrow \lambda x : C'.\, \langle D' \Leftarrow D \rangle^p\ (v\ (\langle C \Leftarrow C' \rangle^{\bar{p}}\ x)) \\
\langle \star \Leftarrow \star \rangle^p\ v &\longrightarrow v \\
\langle B \Leftarrow B \rangle^p\ v &\longrightarrow v \\
\langle k(T) \Leftarrow k(T) \rangle^p\ v &\longrightarrow v \\
\langle \star \Leftarrow C \to D \rangle^p\ v &\longrightarrow \langle \star \Leftarrow \star \to \star \rangle^p\ \langle \star \to \star \Leftarrow C \to D \rangle^p\ v, \quad \text{if } C \to D \neq \star \to \star \\
\langle C \to D \Leftarrow \star \rangle^p\ v &\longrightarrow \langle C \to D \Leftarrow \star \to \star \rangle^p\ \langle \star \to \star \Leftarrow \star \rangle^p\ v, \quad \text{if } C \to D \neq \star \to \star \\
\langle G \Leftarrow \star \rangle^q \langle \star \Leftarrow G \rangle^p\ v &\longrightarrow v \\
\langle H \Leftarrow \star \rangle^q \langle \star \Leftarrow G \rangle^p\ v &\longrightarrow \mathtt{blame}\ q, \quad \text{if } G \neq H
\end{aligned}
$$

$$
\frac{K;\ s \longrightarrow t;\ K'}{K;\ E[s] \longrightarrow E[t];\ K'} \qquad \frac{s \longrightarrow \mathtt{blame}\ p}{E[s] \longrightarrow \mathtt{blame}\ p}
$$

Fig. 2. Reduction

We let E range over evaluation contexts, which are standard. We write $K; s \longrightarrow t; K'$, where K and K' are sets of seals, to indicate that term s reduces in a single step to term t. The sets of seals are used to ensure that newly generated seals are fresh. Set

K contains the seals in use before the reduction (so contains all the seals in s) and set K' contains the seals in use after (so contains all the seals in t). These sets grow monotonically, so we always have $K \subseteq K'$. (If we wish to abandon monotonicity, we may add a garbage collection rule, $K \cup \{k\}; s \longrightarrow s; K$ when k does not appear in s.) When the initial and final sets of seals are identical (as happens in all but one of the given reduction rules) we write $s \longrightarrow t$ to abbreviate $K; s \longrightarrow t; K$. We also write $s \longrightarrow \texttt{blame}\ p$ to indicate that s fails in a single step, allocating blame to p; we omit K because no new seals are allocated in such a reduction step. We write multi-step reductions as $s \longrightarrow^* t$ or $s \longrightarrow^* \texttt{blame}\ p$, leaving the seal sets implicit.

The three novel reduction rules are beta reduction for type abstractions, and casting to or from a polymorphic type. Beta reduction for type abstractions is standard, except it introduces a fresh seal. This makes use of the substitution $s[X := k(T)]$, which is standard save that it replaces a type variable by T if the type variable appears in a type, but replaces the type variable by $k(T)$ if it appears in a cast. Key lines of the definition of $s[X := k(T)]$ are included in the figure; all other lines are standard.

A value of polymorphic type is cast by applying it to the dynamic type and recursively casting the result. This reduction takes compatible casts into compatible casts, since $\forall X.\, C \lhd D$ only if $C[X := \star] \lhd D$. A value is cast to a polymorphic type by abstracting over the type variable and recursively casting the result; note that the abstracted type variable may appear free in the cast. Again, this reduction takes compatible casts into compatible casts, since $C \lhd \forall X.\, D$ only if $C \lhd D$.

Constants of function type are interpreted by a semantic function consistent with their type: if $\mathrm{ty}(c) = S \to T$ and value v has type S, then $[\![c]\!](v)$ is a term of type T.

The remaining reductions are standard or appear in previous work on gradual typing. Beta reduction is standard. A cast to a function type from another function type decomposes into separate casts on the argument and result—note the reversal in the argument cast, and the corresponding negating of the blame label. A cast from $\star$, a base type, or a seal to itself is the identity. A cast between a function $C \to D$ and $\star$ factors through a cast to the ground function type $\star \to \star$; it is required that $C \to D$ differs from $\star \to \star$ to avoid an infinite regression. The final two rules concern casting from a ground into $\star$ and back again. If the cast is back to the same ground, it becomes an identity cast, while if the cast is back to a different ground it results in blame. Reductions are closed under evaluation contexts E in the usual way.

The fundamental semantic property satisfied by values of polymorphic type is *relational parametricity*, as introduced by Reynolds (1983), which captures the notion that a polymorphic value is treated abstractly, independently of its representation. Relational parametricity is equivalent to asserting that every value of a polymorphic type satisfies a theorem determined by the type (called 'theorems for free' by Wadler (1989)). For example, corresponding to the type

$$f : \forall X.\, X \to X$$

we have the theorem

> for all types X and X' and all relations $R \subseteq X \times X'$,
> for all values $x : X$ and $x' : X'$, if $(x, x') \in R$ then $(f\ x, f\ x') \in R$.

In particular, for any $v : S$ and $v' : S'$ by taking X to be S and X' to be S' and R to be the relation $\{(v, v')\}$, it follows that $f\ S\ v$ is v and $f\ S'\ v'$ is v', so f must be observably equivalent to the identity function. We also consider that a pair of non-terminating expressions satisfy any relation, so a second possibility is that f is a function that never terminates (for instance, f might be $\lambda x.\, t_0$ where t_0 is any closed term that always reduces to $\texttt{blame}\ p$). These are the only two possibilities. Note that under non-terminating we include expressions that allocate blame, so non-terminating means 'does not reduce to a value'.

Consider the following three casts. (We use the embedding from Figure 1 of untyped terms M into blame calculus terms $\lceil M \rceil$.)

$$\langle \forall X.\, X \to X \Leftarrow \star \rangle^p\ \lceil \lambda x.\, x \rceil$$
$$\langle \forall X.\, X \to X \Leftarrow \star \rangle^p\ \lceil \lambda x.\, x + 1 \rceil$$
$$\langle \forall X.\, X \to X \Leftarrow \star \rangle^p\ \lceil \lambda x.\, \texttt{if}\ x\ \texttt{is}^q\ \texttt{I}\ \texttt{then}\ x + 1\ \texttt{else}\ x \rceil$$

As one might expect, the first is equivalent to the polymorphic identity function. More surprisingly, we can arrange that the second and third both behave as a function that never terminates. Hence, parametricity is satisfied.

How can we define polymorphic casts in order to ensure this behaviour? The intuition behind parametricity is that a value of polymorphic type must be treated uniformly. Given a value of type X, the only things one can do with it are to directly return the value, or to pass it to a function expecting a value of type X; there should be no way to examine a value of type X. Therefore, we arrange our system to ensure that when we cast from type X to type $\star$ that we seal the value in a way that prevents it from being examined, and that when we coerce from type $\star$ back to type X that the corresponding seal is removed.

Here is the first example.

$$\begin{aligned}
&(\langle \forall X.\, X \to X \Leftarrow \star \rangle^p\ \lceil \lambda x.\, x \rceil)\ \texttt{I}\ 3 \\
\longrightarrow^*\ &(\langle k(\texttt{I}) \to k(\texttt{I}) \Leftarrow \star \rangle^p\ \lceil \lambda x.\, x \rceil)\ 3 \\
\longrightarrow^*\ &\langle k(\texttt{I}) \Leftarrow \star \rangle^p\ (\lambda x : \star.\, x)\ (\langle \star \Leftarrow k(\texttt{I}) \rangle^{\bar{p}}\ 3) \\
\longrightarrow^*\ &\langle k(\texttt{I}) \Leftarrow \star \rangle^p\ \langle \star \Leftarrow k(\texttt{I}) \rangle^{\bar{p}}\ 3 \\
\longrightarrow^*\ &3
\end{aligned}$$

To perform the type application to $\texttt{I}$, each occurrence of X in the cast is replaced by $k(\texttt{I})$, where k is a fresh seal. Regardless of what type and value are supplied the casts still match, so this behaves as the identity function.

Here is the second example.

$$\begin{aligned}
&(\langle \forall X.\, X \to X \Leftarrow \star \rangle^p\ \lceil \lambda x.\, x + 1 \rceil)\ \texttt{I}\ 3 \\
\longrightarrow^*\ &(\langle k(\texttt{I}) \to k(\texttt{I}) \Leftarrow \star \rangle^p\ \lceil \lambda x.\, x + 1 \rceil)\ 3 \\
\longrightarrow^*\ &\langle k(\texttt{I}) \Leftarrow \star \rangle^p\ \langle \star \Leftarrow \texttt{I} \rangle^q\ (\langle \texttt{I} \Leftarrow \star \rangle^q\ \langle \star \Leftarrow k(\texttt{I}) \rangle^{\bar{p}}\ 3) + 1 \\
\longrightarrow^*\ &\texttt{blame}\ q
\end{aligned}$$

Here we've used q to label the casts introduced in translating $\lceil \lambda x.\, x + 1 \rceil$. Again, k is a freshly generated seal. Regardless of what type and value are supplied the casts still don't match, so this behaves as a function that never terminates. It always fails, blaming q.

Here is the third example.

$$
\begin{aligned}
& (\langle \forall X.\, X \to X \Leftarrow \star \rangle^p \, \lceil \lambda x.\, \mathtt{if}\ x\ \mathtt{is}^q\ \mathtt{I}\ \mathtt{then}\ x+1\ \mathtt{else}\ x \rceil)\ \mathtt{I}\ 3 \\
\longrightarrow^* & \ (\langle k(\mathtt{I}) \to k(\mathtt{I}) \Leftarrow \star \rangle^p \, \lceil \lambda x.\, \mathtt{if}\ x\ \mathtt{is}^q\ \mathtt{I}\ \mathtt{then}\ x+1\ \mathtt{else}\ x \rceil)\ 3 \\
\longrightarrow^* & \ \langle k(\mathtt{I}) \Leftarrow \star \rangle^p\ \mathtt{if}\ \langle \star \Leftarrow k(\mathtt{I}) \rangle^{\bar{p}}\ 3\ \mathtt{is}^q\ \mathtt{I}\ \mathtt{then}\ \cdots\ \mathtt{else}\ \cdots \\
\longrightarrow^* & \ \mathtt{blame}\ q
\end{aligned}
$$

Sealed values should not be examined, so we arrange for $s\ \mathtt{is}^q\ G$ to allocate blame to q if its argument is sealed. One might expect the expression $s\ \mathtt{is}^q\ G$ to simply return false in this case, and in fact this would retain parametricity (the expression in question would become the identity function). But we would lose another key property, since we want casting to another type (including to a polymorphic type) to leave the value unaltered or to indicate blame, but never to change the value returned. In this case, wrapping the expression changes it to a function that never terminates, which is acceptable, while changing it to the identity function violates our criterion.

4 Subtyping

Subtyping characterises when a cast cannot give rise to blame. Figure 3 presents four subtyping judgements—ordinary, positive, negative, and naive. (Strictly speaking, these are subcasting judgements, since we compare casts rather than types.)

We write $C <: D$ if C is a subtype of D. Every subtype of a ground is a subtype of $\star$, since a cast from a ground to $\star$ cannot allocate blame. A base type, type variable, or seal is a subtype of itself. Function subtyping is contravariant in the domain and covariant in the range. Finally, we have the rules for polymorphic types. A polymorphic type $\forall X.\, C$ is a subtype of D if its instance $C[X := \star]$ is a subtype of D. (A more general definition would say $\forall X.\, C$ is a subtype of D if *any* instance $C[X := T]$ is a subtype of D; we return to this point in Section 6.) We also have, by analogy with the compatibility rules, that a type C is a subtype of a polymorphic type $\forall X.\, D$ only if C is a subtype of D, when X does not appear free in C. As with compatibility, these rules allow us to show that $C <: D$ implies $\forall X.\, C <: \forall X.\, D$.

In order to characterize when positive and negative blame cannot occur, we factor subtyping into two subsidiary relations, positive subtyping, written $C <:^+ D$ and negative subtyping, written $C <:^- D$. We will show that if $C <:^+ D$ then a cast from C to D cannot receive positive blame, and that if $C <:^- D$ then a cast from C to D cannot receive negative blame.

The two judgements are defined in terms of each other, and track the swapping of positive and negative blame labels that occurs with function types, with the contravariant position in the function typing rule reversing the roles. We have $C <:^+ \star$ and $\star <:^- D$ for every cast C and D, since casting to $\star$ can never give rise to positive blame, and casting from $\star$ can never give rise to negative blame. We also have $C <:^- G$ implies $C <:^- D$, since a cast from a ground to $\star$ cannot allocate blame, and a cast from $\star$ to any type cannot allocate negative blame.

We also define a naive subtyping judgement, $C <:_n D$, which corresponds to our informal notion of type C being more precise than type D, and is covariant for both the domain and range of functions.

Subtype $\boxed{C <: D}$

$$\frac{C <: G}{C <: \star} \qquad \frac{}{\star <: \star} \qquad \frac{}{B <: B} \qquad \frac{}{X <: X} \qquad \frac{}{k(T) <: k(T)}$$

$$\frac{C' <: C \qquad D <: D'}{C \to D <: C' \to D'} \qquad \frac{C[X := \star] <: D}{\forall X.\, C <: D} \qquad \frac{C <: D}{C <: \forall X.\, D}\, X \notin C$$

Positive subtype $\boxed{C <:^+ D}$

$$\frac{}{C <:^+ \star} \qquad \frac{}{B <:^+ B} \qquad \frac{}{X <:^+ X} \qquad \frac{}{k(T) <:^+ k(T)}$$

$$\frac{C' <:^- C \qquad D <:^+ D'}{C \to D <:^+ C' \to D'} \qquad \frac{C[X := \star] <:^+ D}{\forall X.\, C <:^+ D} \qquad \frac{C <:^+ D}{C <:^+ \forall X.\, D}\, X \notin C$$

Negative subtype $\boxed{C <:^- D}$

$$\frac{C <:^- G}{C <:^- D} \qquad \frac{}{\star <:^- D} \qquad \frac{}{B <:^- B} \qquad \frac{}{X <:^- X} \qquad \frac{}{k(T) <:^- k(T)}$$

$$\frac{C' <:^+ C \qquad D <:^- D'}{C \to D <:^- C' \to D'} \qquad \frac{C[X := \star] <:^- D}{\forall X.\, C <:^- D} \qquad \frac{C <:^- D}{C <:^- \forall X.\, D}\, X \notin C$$

Naive subtype $\boxed{C <:_n D}$

$$\frac{}{C <:_n \star} \qquad \frac{}{B <:_n B} \qquad \frac{}{X <:_n X} \qquad \frac{}{k(T) <:_n k(T)}$$

$$\frac{C <:_n C' \qquad D <:_n D'}{C \to D <:_n C' \to D'} \qquad \frac{C[X := \star] <:_n D}{\forall X.\, C <:_n D} \qquad \frac{C <:_n D}{C <:_n \forall X.\, D}\, X \notin C$$

Fig. 3. Subtypes

All four subtyping relations are transitive. They are not reflexive; for example $\forall X.\, X$ is not related to itself in any of the four relations. Modifying the rules to ensure reflexivity is important future work.

We will show that $C <: D$ if and only if $C <:^+ D$ and $C <:^- D$, hence casting from a subtype can never allocate positive or negative blame. We will also show that $C <:_n D$ if and only if $C <:^+ D$ and $D <:^- C$. (Note the reversal! Compare $C <:^- D$ in the previous sentence with $D <:^- C$ here.) Hence casting from a naive subtype can never allocate positive blame, while casting to a naive subtype can never allocate negative blame.

$$\frac{C <:^{+} D \qquad s \text{ sf } p}{\langle D \Leftarrow C \rangle^{p}\ s \text{ sf } p} \qquad \frac{C <:^{-} D \qquad s \text{ sf } p}{\langle D \Leftarrow C \rangle^{\bar{p}}\ s \text{ sf } p} \qquad \frac{p \neq q \qquad \bar{p} \neq q \qquad s \text{ sf } p}{\langle D \Leftarrow C \rangle^{q}\ s \text{ sf } p} \qquad \frac{}{x \text{ sf } p}$$

$$\frac{t \text{ sf } p \qquad p \neq q}{s\ \mathtt{is}^{q}\ G \text{ sf } p} \qquad \frac{}{c \text{ sf } p} \qquad \frac{t \text{ sf } p}{\lambda x : S.\, t \text{ sf } p} \qquad \frac{t \text{ sf } p \qquad s \text{ sf } p}{t\ s \text{ sf } p} \qquad \frac{t \text{ sf } p}{\Lambda X.\, t \text{ sf } p} \qquad \frac{t \text{ sf } p}{t\ S \text{ sf } p}$$

Fig. 4. Safe terms

5 The Blame Theorem

All results carry over directly from our previous work. To begin, we have the usual preservation and progress results.

Proposition 1. *(Preservation) If* $\Gamma \vdash s : T$ *and* $K;\ s \longrightarrow t;\ K'$ *then* $\Gamma \vdash t : T$.

Proposition 2. *(Progress) If* $\vdash s : T$ *and* K *contains all seals in* s *then either*

- s *is a value, or*
- $K;\ s \longrightarrow t;\ K'$ *for some term* t *and seal set* K'*, or*
- $s \longrightarrow \mathtt{blame}\ p$ *for some blame label* p.

Preservation and progress on their own do not guarantee a great deal, since they do not rule out blame as a result. We now turn our attention to results that characterise situations in which blame cannot arise.

Figure 4 defines the safety relation. A term t is safe for a blame label p, written t sf p, if all of the casts that have the label p are positive subtypes and all of the casts that have the label $\bar{p}$, are negative subtypes. We have the following results.

Proposition 3. *(Preservation of safe terms) If* $\Gamma \vdash s : S$ *and* s sf p *and* $K;\ s \longrightarrow t;\ K'$ *then* t sf p.

Proposition 4. *(Progress of safe terms) If* $\vdash s : S$ *and* s sf p *then* $s \not\longrightarrow \mathtt{blame}\ p$.

Corollary 1. *(Positive and negative blame) Let* t *be a closed, well-typed term with a subterm* $\langle D \Leftarrow C \rangle^{p}\ s$ *containing the only occurrence of* p *in* t.

- *If* $C <:^{+} D$ *then* $t \not\longrightarrow^{*} \mathtt{blame}\ p$.
- *If* $C <:^{-} D$ *then* $t \not\longrightarrow^{*} \mathtt{blame}\ \bar{p}$.

Subtyping factors into positive and negative subtyping, and naive subtyping also factors into positive and negative subtyping, this time with the direction of negative subtyping reversed.

Proposition 5. *(Factoring subtyping)* $C <: D$ *iff* $C <:^{+} D$ *and* $C <:^{-} D$.

Proposition 6. *(Factoring naive subtyping)* $C <:_{n} D$ *iff* $C <:^{+} D$ *and* $D <:^{-} C$.

The proof of Proposition 6 requires four observations. First, if $C <:^+ D$ and $X \notin C$, then $X \notin D$ and second if $C <:^- D$ and $X \notin D$, then $X \notin C$. Third, given $X \notin D$, we have $C[X := \star] <:^+ D$ iff $C <:^+ D$ and fourth given $X \notin C$, we have $C <:^- D[X := \star]$ iff $C <:^- D$.

We may now characterize how ordinary and naive subtyping relate to blame.

Corollary 2. *(Well typed programs can't be blamed) Let t be a closed, well-typed term with a subterm $\langle D \Leftarrow C \rangle^p\ s$ containing the only occurrence of p in t.*

- *If $C <: D$ then $t \not\longrightarrow^* \texttt{blame}\, p$ and $t \not\longrightarrow^* \texttt{blame}\, \bar{p}$.*
- *If $C <:_n D$ then $t \not\longrightarrow^* \texttt{blame}\, p$.*
- *If $D <:_n C$ then $t \not\longrightarrow^* \texttt{blame}\, \bar{p}$.*

Since our notion of more and less precise types is captured by naive subtyping, the last two clauses show that any failure of a cast from a more-precisely-typed term to a less-precisely-typed context must be blamed on the less-precisely-typed context, and any failure of a cast from a less-precisely-typed term to a more-precisely-typed context must be blamed on the less-precisely-typed term.

6 Improved subtyping

The penultimate rule in the definition of subtyping (Figure 3), says that a polymorphic type is a subtype of another if its instance at the dynamic type is a subtype of the other.

$$\frac{C[X := \star] <: D}{\forall X.\, C <: D}$$

Arguably, this rule is too weak. For instance, it does not allow us to conclude that $\forall X.\, X \to X <: \mathtt{I} \to \mathtt{I}$, because we do not have $\star \to \star <: \mathtt{I} \to \mathtt{I}$.

But nonetheless it seems sensible to assert a subtype relation in this case, since for any closed, well-typed term t with a subterm

$$\langle \mathtt{I} \to \mathtt{I} \Leftarrow \forall X.\, X \to X \rangle^p\ s$$

containing the only occurrence of p in t, we have $t \not\longrightarrow^* \texttt{blame}\, p$ and $t \not\longrightarrow^* \texttt{blame}\, \bar{p}$.

To fix the problem, we define a new subtyping relation $<:'$, identical to $<:$, but with the subtyping rule above replaced by

$$\frac{C[X := S] <:' D}{\forall X.\, C <:' D}$$

where $C[X := \star]$ in the old rule becomes $C[X := S]$ in the new. With this change, we can conclude that $\forall X.\, X \to X <:' \mathtt{I} \to \mathtt{I}$ by picking $\mathtt{I}$ for S.

We then define an analogue of safety, saying that t is safe for p if for every subterm of the form $\langle D \Leftarrow C \rangle^q\ s$ we have $C <:' D$ whenever $p = q$ or $\bar{p} = q$. We conjecture that progress and preservation also hold for this second form of safety.

Our original conjecture was that we could similarly generalize positive, negative, and naive subtyping, but this is not the case. For instance, say we give a definition of $<:'_n$

that is identical to $<:_n$, except that we replace $C[X := \star] <:_n D$ by $C[X := S] <:'_n D$ in the hypothesis of the penultimate rule. Then since $\mathtt{I} \to \mathtt{I} <:'_n \star \to \mathtt{I}$, by taking $X := \mathtt{I}$ we may conclude $\forall X.\, X \to X <:'_n \star \to \mathtt{I}$. But

$$(\langle \star \to \mathtt{I} \Leftarrow \forall X.\, X \to X \rangle^p \; \mathit{id})\; (\langle \star \Leftarrow \mathtt{B} \rangle^q \mathtt{true})$$
$$\longrightarrow^* \langle \mathtt{I} \Leftarrow \star \rangle^p (\lambda x : \star.\, x)\; (\langle \star \Leftarrow \star \rangle^{\bar{p}}\; (\langle \star \Leftarrow \mathtt{B} \rangle^q \mathtt{true})) \longrightarrow^* \mathtt{blame}\, p$$

where *id* is the polymorphic identity function. So while proper subtyping may generalize to arbitrary instances of polymorphic types, naive subtyping does not do so. Though our work on blame has emphasized the importance of naive subtyping, this shows that ordinary subtyping may still have an important role to play.

Bibliography

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

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

Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *Dynamic Languages Symposium (DLS)*, October 2007.

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

Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing. In *European Symposium on Programming (ESOP)*, pages 16–31, 2008.

Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2007.

James H. Morris, Jr. Types are not sets. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 120–124, 1973.

Georg Neis, Derek Dreyer, and Andreas Rossberg. Non-parametric parametricity. Manuscript submitted for publication, March 2009. URL http://www.mpi-sws.org/∼dreyer/papers/npp/main.pdf.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *IFIP International Conference on Theoretical Computer Science*, August 2004.

Benjamin Pierce and Eijiro Sumii. Relating cryptography and polymorphism. Manuscript, 2000. URL http://www.cis.upenn.edu/∼bcpierce/papers/infohide.ps.

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

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

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

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

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