A

Blame for All (revised)

Amal Ahmed, Northeastern University
James T. Perconti, Northeastern University
Jeremy G. Siek, Indiana University
Philip Wadler, University of Edinburgh

Several programming languages are beginning to integrate static and dynamic typing, including Racket, Microsoft's C# 4.0 (Hejlsberg 2010) and TypeScript (Hejlsberg, 2012), Facebook's PHP (Verlaguet, 2013), and the research languages Sage (Gronski, Knowles, Tomb, Freund, and Flanagan, 2006) and Thorn (Wrigstad, Eugster, Field, Nystrom, and Vitek, 2009). However, an important open question remains, which is how to add parametric polymorphism to languages that combine static and dynamic typing. We present a system that permits a value of dynamic type to be cast to a polymorphic type and vice versa, with relational parametricity enforced by a kind of dynamic sealing along the lines proposed by Matthews and Ahmed (2008) and Neis, Dreyer, and Rossberg (2009). Our development is supported by a variant of the polymorphic lambda calculus that may be of independent interest. It uses type bindings and static casts to maintain a syntactic certificate of parametricity throughout program evaluation. Our system includes a notion of blame, which allows us to show that when casting between a more precise type and a less precise type, any cast failures are due to the less-precisely-typed portion of the program. We also want to show that a cast from a subtype to a supertype cannot fail. This property holds for a relatively weak notion of subtyping (STOP 2009) but we have found a flaw in our proof for the stronger notion of subtyping, which we explain here. Finally, we show that existentials for untyped programs can be encoded in terms of casts through polymorphic types.

Categories and Subject Descriptors: D.2.4 [**Software/Program Verification**]: Programming by contract; D.3.3 [**Language Constructs and Features**]: Polymorphism; F.3.2 [**Semantics of Programming Languages**]: Operational semantics; F.3.3 [**Studies of Program Constructs**]: Type structure

General Terms: Languages, Theory

Additional Key Words and Phrases: casts, coercions, blame tracking, lambda-calculus

1. INTRODUCTION

The long tradition of work that integrates static and dynamic types includes the *partial types* of Thatte [1988], the *dynamic type* of Abadi et al. [1991], the *coercions* of Henglein [1994], the *contracts* of Findler and Felleisen [2002], the *dynamic dependent types* of Ou et al. [2004], the *hybrid types* of Gronski et al. [2006], the *gradual types* of Siek and Taha [2006], the *migratory types* of Tobin-Hochstadt and Felleisen [2006], the *multi-language* programming of Matthews and Findler [2007], and the *blame calculus* of Wadler and Findler [2009]. Integration of static and dynamic types is a feature of .NET languages including Visual Basic [Meijer 2004] and C# [Hejlsberg 2010], it is the main feature of Microsoft's TypeScript [Hejlsberg 2012] dialect of JavaScript, it has been added to PHP by Facebook [Verlaguet 2013], it is being explored for Perl, Python, and Ruby, and it is the subject of the Scripts to Programs (STOP) workshop series.

A unifying theme in this work is to use casts to mediate between statically and dynamically typed code. Casts may be introduced by compiling to an intermediate language; the blame calculus may be regarded as either such an intermediate language or as a source language. The main innovation of the blame calculus 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 support the Blame Theorem, which ensures that when a program goes wrong, blame lies with the less-precisely-typed side of a cast [Wadler and Findler 2009].

In this paper, we extend a fragment of the blame calculus to incorporate polymorphism, based on a notion of dynamic sealing. For simplicity, our fragment includes base types, function types, and the dynamic type, as found in gradual types, but omits subset types, as found in hybrid types. Our system adds the ability to cast a value of dynamic type to a polymorphic type and vice versa. We name this system the *polymorphic blame calculus*.

A fundamental semantic property of polymorphic types is *relational parametricity*, as introduced by Reynolds [1983]. Our system uses dynamic sealing to ensure that values of polymorphic type satisfy relational parametricity. For instance, every function of type $\forall X.\, X \to X$ must either be the identity function (one which always returns its argument) or an undefined function (one which never returns a value), and this property holds true even for values of dynamic type cast to a polymorphic type. Relational parametricity underlies some program optimizations, notably *shortcut deforestation* as employed by the Glasgow Haskell Compiler [Gill et al. 1993]. Our system may guarantee the validity of such optimizations even in the presence of dynamic types.

The use of dynamic sealing to enforce parametricity has a long history. Morris [1973] was the first to suggest the application of sealing to data abstraction. Pierce and Sumii [2000] described a connection between cryptographic sealing and relational parametricity. Guha et al. [2007] used sealing to implement contracts for polymorphic types in Scheme (now Racket). Matthews and Ahmed [2008] proved sealing enforces relational parametricity, in the setting of casts between a polymorphically typed language and a dynamically typed language. Neis et al. [2009] used dynamic sealing to restore parametricity in a non-parametric language.

Dynamic sealing is accomplished in our system by the use of *type bindings* to control the scope of type variables. Type bindings enable an operational semantics for the polymorphic lambda calculus that does not implement type application through type substitution, but instead uses type bindings as a kind of explicit substitution, preserving the type-hiding nature of type abstractions after they are instantiated. With this approach, type variables play the role of dynamic seals. Our development also uses *static casts* as a technical device in our proofs to precisely track where type variables are concealed and revealed.

Type bindings and static casts may be of interest independently to the blame calculus, as they connect to several strands of previous work. Type bindings resemble constructs for generating new type names in Neis et al. [2009] and Rossberg [2003]; an important difference is that our type bindings are immobile, that is, there is no scope extrusion. Static casts relate to the coercions of Rossberg [2003] and are reminiscent of the syntactic type abstractions of Grossman et al. [2000].

The conference version of this paper, Ahmed et al. [2011], appeared three years ago. Why did we not prepare the journal version earlier? That paper included a proof of the Jack-of-All-Trades Principle, which justifies our reduction rule for casts that instantiate polymorphic values. In the preparation of this article we discovered a flaw in our previously published proof. Having failed to find a fix after three years, we believe it is important to publish a description of the problem. This paper renames our principle as the Jack-of-All-Trades Conjecture, and includes a detailed description of the flaw we found and why it is difficult to fix.

The paper is structured as follows. Section 2 gives an overview of our approach. Section 3 recalls the simply-typed lambda calculus. Section 4 recalls the simply-typed blame calculus. Section 5 introduces the polymorphic lambda calculus with type binding, establishes its type safety, and proves that it is equivalent to the standard polymorphic lambda calculus. Section 6 introduces the polymorphic blame calculus, motivates its reduction rules, and establishes its type safety. Section 7 explains why we require four different subtyping relations. Section 8 proves the Blame Theorem. Section 9 proves the Subtyping Theorem, which depends on the Jack-of-All-Trades Conjecture. Section 10 adds static casts to the polymorphic lambda calculus with type binding, establishes type safety of the extended system, and shows erasure of static

$$
\begin{tikzcd}
\text{simply-typed lambda calculus (Figure 2)} \arrow[r] \arrow[d] & \text{simply-typed blame calculus (Figure 3)} \arrow[d] \\
\text{polymorphic lambda calculus with type bindings (Figure 4)} \arrow[r] \arrow[d] & \text{polymorphic blame calculus (Figure 5)} \arrow[d] \\
\text{polymorphic lambda calculus with static casts (Figure 9)} \arrow[r] & \text{polymorphic blame calculus with static casts (Figure 10)}
\end{tikzcd}
$$

Fig. 1. Summary of calculi

casts preserves semantics. Section 11 explains in detail the flaw in our previously published proof of Jack-of-All-Trades. Section 12 describes how to obtain existentials in an untyped language via casts involving polymorphic types. Section 13 summarizes related work. Section 14 concludes.

Figure 1 diagrams the six calculi presented in this paper and the dependencies between them.

We do not present a relational parametricity result. We expect it should be a relatively straightforward adaptation of the result by Matthews and Ahmed [2008], but the current state-of-the-art regarding step-indexed logical relations is such that any proof of this kind is tedious and technically difficult to write out in full.

This article is a rewrite of the conference version, Ahmed et al. [2011]. Small improvements have been made throughout. Major differences include a discussion of why sharpened versions of Jack-of-all-Trades cannot hold (Section 6.4), an explanation of the flaw in the proof of Jack-of-all-Trades (Section 11), and a new treatment of existentials (Section 12).

We have produced a Redex model [Felleisen et al. 2009] covering some of the systems in this paper, available online:

`http://plt.eecs.northwestern.edu/blame-for-all/`

2. FROM UNTYPED TO TYPED

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

Here is a simple untyped program.

$$
\begin{array}{l}
\texttt{let}\; \mathit{pos}^\star = \lceil \lambda x.\, x > 0 \rceil \;\texttt{in} \\
\texttt{let}\; \mathit{app}^\star = \lceil \lambda f.\, \lambda x.\, f\ x \rceil \;\texttt{in} \\
\lceil \mathit{app}^\star\ \mathit{pos}^\star\ 1 \rceil
\end{array}
$$

It returns $\lceil \texttt{true} \rceil : \star$. We indicate untyped code by surrounding it with ceiling brackets, $\lceil \cdot \rceil$. Untyped code is really uni-typed (a slogan due to Harper [2007]); it is a special case of typed code where every term has the dynamic type, $\star$. To aid the eye, we sometimes write variables of type $\star$ with a superscript $\star$.

Here is the same program, rewritten with types, where $\texttt{I}$ is the type for integers and $\texttt{B}$ is the type for Boolean values.

$$
\begin{array}{l}
\texttt{let}\; \mathit{pos} = \lambda x : \texttt{I}.\, x > 0 \;\texttt{in} \\
\texttt{let}\; \mathit{app} = \Lambda X.\, \Lambda Y.\, \lambda f : X {\to} Y.\, \lambda x : X.\, f\ x \;\texttt{in} \\
\mathit{app}\ \texttt{I}\ \texttt{B}\ \mathit{pos}\ 1
\end{array}
$$

This program returns $\texttt{true} : \texttt{B}$.

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 three-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 using a relatively new construct [Flanagan 2006; Siek and Taha 2006] with an old name, "cast". Casts can be between any two compatible types. Roughly speaking, type A is compatible with type B when a value of type A might be coerced to type B. We are particularly interested in the case where either the source type is $\star$ (corresponding to importing untyped code into typed code), or where the target type is $\star$ (corresponding to importing typed code into untyped code). We introduce an order on types corresponding to precision, where $\star$ is the least precise type. We introduce a notion of *blame* associated with casts, so that we can prove the following result: if a cast between a less precise type and a more precise type fails, then blame falls on the less precise side of the cast. An immediate corollary is that if a cast between untyped and typed code fails, blame lies with the untyped code—"well-typed programs can't be blamed".

We write $t : A \Rightarrow^p B$ to cast the result of term t from type A to type B. Every cast is annotated with a *blame label* p, used to ascribe fault if the cast fails. Our notation is chosen for clarity rather than compactness. Writing the source type of the cast is redundant; the type of the source can always be inferred. In a practical language, we would expect the source type to be elided.

A cast from a more precise type to a less precise type is called *widening*. Here is the previous program rewritten to demonstrate widening. It is mostly untyped, but contains one typed component cast for use in an untyped context.

$\mathtt{let}\ \mathit{pos}^\star = \lceil \lambda x.\, x > 0 \rceil\ \mathtt{in}$
$\mathtt{let}\ \mathit{app} = \Lambda X.\, \Lambda Y.\, \lambda f : X \to Y.\, \lambda x : X.\, f\ x\ \mathtt{in}$
$\mathtt{let}\ \mathit{app}^\star = \mathit{app} : \forall X.\, \forall Y.\, (X \to Y) \to X \to Y \Rightarrow^p \star\ \mathtt{in}$
$\lceil \mathit{app}^\star\ \mathit{pos}^\star\ 1 \rceil$

It returns $\lceil \mathtt{true} \rceil : \star$.

Of course, the untyped context may not satisfy the constraints required by the typed term. If in the above we replace

$$\lceil \mathit{app}^\star\ \mathit{pos}^\star\ 1 \rceil \qquad \text{by} \qquad \lceil \mathit{app}^\star\ 1\ \mathit{pos}^\star \rceil$$

it now returns $\mathtt{blame}\ \overline{p}$. Blaming $\overline{p}$ (rather than p) indicates that the fault lies with the *context containing* the cast labelled p (rather than the *term contained* in the cast). This is what we expect, because the context is untyped.

Passing a polymorphically typed value into an untyped context requires an appropriate instantiation for the type parameters. Intuitively, in this case one might instantiate X by $\mathtt{I}$ and Y by $\mathtt{B}$, but it is hard to think of a systematic method of choosing what type to instantiate. Instead, we use a simple rule: always instantiate with $\star$. How can we justify such a rule? One of the contributions of this paper is to formulate a *Jack-of-All-Trades Conjecture*, which provides such a justification. It asserts that if instantiating a type parameter to any given type yields an answer then instantiating that type parameter to $\star$ yields the same answer.

A cast from a less precise type to a more precise type is called *narrowing*. Here is the above program rewritten to demonstrate narrowing. It is mostly typed, but contains one untyped component cast for use in a typed context.

$\mathtt{let}\ \mathit{pos} = \lambda x : \mathtt{I}.\, x > 0\ \mathtt{in}$
$\mathtt{let}\ \mathit{app}^\star = \lceil \lambda f.\, \lambda x.\, f\ x \rceil\ \mathtt{in}$
$\mathtt{let}\ \mathit{app} = \mathit{app}^\star : \star \Rightarrow^p \forall X.\, \forall Y.\, (X \to Y) \to X \to Y\ \mathtt{in}$
$\mathit{app}\ \mathtt{I}\ \mathtt{B}\ \mathit{pos}\ 1$

This returns $\mathtt{true} : \mathtt{B}$.

Of course, the untyped term may not satisfy the constraints required by the typed context. If in the above we replace

$$\lceil \lambda f.\, \lambda x.\, f\ x \rceil \qquad \text{by} \qquad \lceil \lambda f.\, \lambda x.\, x \rceil$$

it now returns $\mathtt{blame}\ p$. Blaming p (rather than $\overline{p}$) indicates that the fault lies with the *term contained* in the cast labelled p (rather than the *context containing* the cast). This is what we expect, because the term is untyped.

To check for this error, the implementation must *seal* each value. That is, casting from type X to type $\star$ yields a value sealed with X, and attempting to cast from type $\star$ to type Y fails because the seals X and Y are distinct. One of the contributions of this paper is to work out the details of sealing in a setting with dynamic types. One consequence of sealing is that typed terms always satisfy appropriate parametricity properties, even when they are derived by casting from untyped terms.

We now begin our formal development.

3. SIMPLY-TYPED LAMBDA CALCULUS

All the systems in this paper extend a vanilla call-by-value simply-typed lambda calculus, shown in Figure 2.

We let A, B, and C range over types. A type is either a base type ι or a function type $A \to B$. The base types include integers and Booleans, written $\mathtt{I}$ and $\mathtt{B}$ respectively. We let s and t range over terms. Terms include constants, primitive application, variables, abstractions, and application. The variables v and w range over values. A value is either a constant or an abstraction.

We write $\Gamma \vdash t : A$ if term t has type A in type environment Γ. A type environment maps variables to types. The function ty maps constants and primitive operators to their types. The function δ maps an operator and a tuple of values to a value, and must preserve types. That is, if $\mathrm{ty}(op) = \vec{A} \to B$ and $\cdot \vdash \vec{v} : \vec{A}$ then there is a w such that $\delta(op, \vec{v}\,) = w$ and $\cdot \vdash w : B$. Suitable choices of δ can specify arithmetic, conditional, and fixpoint operators.

We write $s \longrightarrow t$ to indicate that redex s reduces to t, and write $s \longmapsto t$ to indicate that reducing a redex inside s yields t. We let E range over evaluation contexts, which are standard.

4. SIMPLY-TYPED BLAME CALCULUS

Before proceeding to polymorphism, we review the fundamentals of the simply-typed blame calculus, shown in Figure 3. The blame calculus extends the simply-typed lambda-calculus with a dynamic type, written $\star$, and with four term forms: dynamic casts, grounded terms, type tests, and blame.

One can think of the dynamic type $\star$ as the sum of all the base types plus the function type at $\star$.

$$\star = \mathtt{I} + \mathtt{B} + (\star \to \star)$$

Accordingly, the *ground types* are the base types together with the type $\star \to \star$. Every value of dynamic type is constructed by a cast from ground type to dynamic type, written $v : G \Rightarrow \star$. These casts can never fail, so they are not decorated with blame labels. For example, $id^\star = (\lambda x{:}\,\star\,.\,x) : \star \to \star \Rightarrow \star$ is a value of type $\star$.

In general, a cast $s : A \Rightarrow^p B$ converts the value of term s from type A to type B. Casts are decorated with blame labels. We assume an involutive operation of negation on blame labels: if p is a blame label then $\overline{p}$ is its negation, and $\overline{\overline{p}}$ is the same as p. We write $s : A \Rightarrow^p B \Rightarrow^q C$ as shorthand for $(s : A \Rightarrow^p B) : B \Rightarrow^q C$. A cast from A to B is permitted only if the types are *compatible*, written $A \prec B$. Every type is

Syntax

Variables	x, y		Terms	s, t	$::=$	$c \mid op(\vec{t}) \mid x \mid \lambda x{:}A.\, t \mid t\ s$
Constants	c		Environments	Γ	$::=$	$\cdot \mid \Gamma, x : A$
Base types	ι	$::= \mathtt{I} \mid \mathtt{B}$	Values	v, w	$::=$	$c \mid \lambda x{:}A.\, t$
Types	A, B, C	$::= \iota \mid A \to B$	Contexts	E	$::=$	$[\cdot] \mid op(\vec{v}, E, \vec{t}) \mid E\ s \mid v\ E$

Type rules

$$\frac{\mathrm{ty}(c) = \iota}{\Gamma \vdash c : \iota} \qquad \frac{\Gamma \vdash \vec{t} : \vec{A} \quad \mathrm{ty}(op) = \vec{A} \to B}{\Gamma \vdash op(\vec{t}) : B} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x{:}A.\, t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash s : A}{\Gamma \vdash t\ s : B}$$

Reduction rules

$$(\lambda x{:}A.\, t)\ v \longrightarrow t[x{:=}v] \quad (\textsc{Beta}) \qquad\qquad \frac{s \longrightarrow t}{E[s] \longmapsto E[t]} \quad (\textsc{Step})$$

$$op(\vec{v}) \longrightarrow \delta(op, \vec{v}) \quad (\textsc{Delta})$$

Fig. 2. Simply-typed lambda calculus.

compatible with itself, the dynamic type is compatible with every type, and functions are compatible if their domain and range are compatible; note the contravariance in the function rule. For now, compatibility is symmetric, but this changes in Section 6.

A test s is G returns true if s evaluates to a value grounded on G. For example, $(1 : \mathtt{I} \Rightarrow \star)$ is $\mathtt{I}$ returns true.

Finally, the term $\mathtt{blame}\ p$ indicates a failure, identifying the relevant label. Blame terms may have any type.

We now briefly review the reduction rules. A cast from one function type to another reduces to a wrapper function that casts the argument, applies the original function, then casts the result (Wrap). A cast from a ground type to itself is the identity (Id). (The side condition $G \neq \star \to \star$ avoids overlap with (Wrap). For now, the only ground type other than $\star \to \star$ is ι, but this changes in Section 6.) A cast from type A to $\star$ factors into a cast from A to the unique ground type G that is compatible with A followed by a cast from G to $\star$ (Ground). Here we see the reason for distinguishing between casts and ground terms: otherwise whenever the (Ground) rule is applicable, it would be applicable infinitely many times. A cast from $\star$ to type A examines the ground G of the value of type $\star$. If G is compatible with A, the two casts collapse to a direct cast from G to A (Collapse). If G is not compatible with A, the offending cast is blamed (Conflict). A test checks the ground of the value of type $\star$. If it matches the test returns true, else it returns false (IsTrue), (IsFalse). An occurrence of $\mathtt{blame}\ p$ in an evaluation position causes the program to abort (Abort).

For example, say $pos = \lambda x : \mathtt{I}.\, x > 0$. Then

$$\begin{array}{ll} & (pos : \mathtt{I} \to \mathtt{B} \Rightarrow^{p} \star \to \star)\ (1 : \mathtt{I} \Rightarrow \star) \\ \longmapsto^{*} & pos\ (1 : \mathtt{I} \Rightarrow \star \Rightarrow^{\overline{p}} \mathtt{I}) : \mathtt{B} \Rightarrow^{p} \star \\ \longmapsto^{*} & pos\ 1\ : \mathtt{B} \Rightarrow^{p} \star \\ \longmapsto^{*} & \mathtt{true} : \mathtt{B} \Rightarrow \star \end{array}$$

The function cast $\mathtt{I} \to \mathtt{B} \Rightarrow^{p} \star \to \star$ factors into a pair of casts. The cast on the ranges $\mathtt{B} \Rightarrow^{p} \star$ retains the order and the blame label. The cast on the domains $\star \Rightarrow^{\overline{p}} \mathtt{I}$ swaps the order and negates the blame label. The swap is required for types to work out. Negation of the blame label is required to assign blame appropriately, as can be seen

Syntax

Blame labels	p, q		
Types	A, B, C	$::=$	$\iota \mid A \to B \mid \star$
Ground types	G, H	$::=$	$\iota \mid \star \to \star$
Terms	s, t	$::=$	$c \mid op(\vec{t}) \mid x \mid \lambda x{:}A.\, t \mid t\ s \mid s : A \Rightarrow^{p} B \mid s : G \Rightarrow \star \mid s\ \mathtt{is}\ G \mid \mathtt{blame}\ p$
Environments	Γ	$::=$	$\cdot \mid \Gamma, x : A$
Values	v, w	$::=$	$c \mid \lambda x{:}A.\, t \mid v : G \Rightarrow \star$
Contexts	E	$::=$	$[\cdot] \mid op(\vec{v}, E, \vec{t}) \mid E\ s \mid v\ E \mid E\ \mathtt{is}\ G \mid E : A \Rightarrow^{p} B \mid E : G \Rightarrow \star$
Untyped terms	M, N	$::=$	$c \mid op(\vec{M}) \mid x \mid \lambda x.\, M \mid M\ N \mid M\ \mathtt{is}\ G$

Type rules

$$\frac{\Gamma \vdash s : A \quad A \prec B}{\Gamma \vdash (s : A \Rightarrow^{p} B) : B} \qquad \frac{\Gamma \vdash s : G}{\Gamma \vdash (s : G \Rightarrow \star) : \star} \qquad \frac{\Gamma \vdash s : \star}{\Gamma \vdash s\ \mathtt{is}\ G : \mathtt{B}} \qquad \Gamma \vdash \mathtt{blame}\ p : A$$

Compatibility

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

Reduction rules

$$\begin{array}{rcll}
v : A \to B \Rightarrow^{p} A' \to B' & \longrightarrow & \lambda x'{:}A'.\, v\ (x' : A' \Rightarrow^{\overline{p}} A) : B \Rightarrow^{p} B' & & (\textsc{Wrap}) \\
v : G \Rightarrow^{p} G & \longrightarrow & v & \text{if } G \neq \star \to \star & (\textsc{Id}) \\
v : A \Rightarrow^{p} \star & \longrightarrow & v : A \Rightarrow^{p} G \Rightarrow \star & \text{if } A \prec G, A \neq \star & (\textsc{Ground}) \\
v : G \Rightarrow \star \Rightarrow^{p} A & \longrightarrow & v : G \Rightarrow^{p} A & \text{if } G \prec A & (\textsc{Collapse}) \\
v : G \Rightarrow \star \Rightarrow^{p} A & \longrightarrow & \mathtt{blame}\ p & \text{if } G \not\prec A & (\textsc{Conflict}) \\
(v : G \Rightarrow \star)\ \mathtt{is}\ G & \longrightarrow & \mathtt{true} & & (\textsc{IsTrue}) \\
(v : H \Rightarrow \star)\ \mathtt{is}\ G & \longrightarrow & \mathtt{false} & \text{if } G \neq H & (\textsc{IsFalse}) \\
E[\mathtt{blame}\ p] & \longmapsto & \mathtt{blame}\ p & \text{if } E \neq [\cdot] & (\textsc{Abort})
\end{array}$$

Fig. 3. Simply-typed blame calculus (extends Figure 2).

by changing the argument:

$$\begin{array}{ll}
 & (\mathit{pos} : \mathtt{I} \to \mathtt{B} \Rightarrow^{p} \star \to \star)\ (\mathtt{false} : \mathtt{B} \Rightarrow \star) \\
\longmapsto^{*} & \mathit{pos}\ (\mathtt{false} : \mathtt{B} \Rightarrow \star \Rightarrow^{\overline{p}} \mathtt{I}) : \mathtt{B} \Rightarrow^{p} \star \\
\longmapsto^{*} & \mathtt{blame}\ \overline{p}
\end{array}$$

The inner cast fails, ascribing blame to the label $\overline{p}$ on the cast. Blaming $\overline{p}$ (rather than p) indicates that the fault in the original cast lies with the context containing the cast (rather than the term contained in the cast). That is, we blame the untyped context for failing to supply an integer.

A cast from type dynamic to itself behaves as the identity:

$$v : \star \Rightarrow^{p} \star \longmapsto v'$$

where v' is observationally equivalent to v. To see this, take $v = w : G \Rightarrow \star$. Then

$$\begin{array}{ll}
 & w : G \Rightarrow \star \Rightarrow^{p} \star \\
\longmapsto & w : G \Rightarrow^{p} \star \\
\longmapsto & w : G \Rightarrow^{p} G \Rightarrow \star
\end{array}$$

via (COLLAPSE) and (GROUND). And a cast from a ground type to itself produces an equivalent value, via (ID) or (WRAP).

It is straightforward to define an embedding $\lceil\cdot\rceil$ from the untyped lambda calculus into the blame calculus.

$$
\begin{aligned}
\lceil c \rceil &= c : \mathrm{ty}(c) \Rightarrow \star \\
\lceil \mathit{op}(\vec{M}\,) \rceil &= \mathit{op}(\lceil \vec{M} \rceil : \vec{\star} \Rightarrow^{\vec{q}} \vec{A}) : B \Rightarrow^{r} \star, \text{ if } \mathrm{ty}(\mathit{op}) = \vec{A} \to B \\
\lceil x \rceil &= x \\
\lceil \lambda x.\, M \rceil &= (\lambda x{:}\star.\, \lceil M \rceil) : \star\to\star \Rightarrow^{q} \star \\
\lceil M\ N \rceil &= (\lceil M \rceil : \star \Rightarrow^{q} \star\to\star)\ \lceil N \rceil \\
\lceil M \;\texttt{is}\; G \rceil &= \lceil M \rceil \;\texttt{is}\; G
\end{aligned}
$$

For example, $\lceil \lambda x.\, x \rceil = (\lambda x{:}\star.\, x){:}\, \star \to\star \Rightarrow \star$.

5. TYPE BINDINGS

The traditional way to reduce a type application is by substitution, $(\Lambda X.\, t)\ A \longrightarrow t[X{:=}A]$. We begin by explaining why this cannot work in the presence of casts and type dynamic, and then introduce a variant of the polymorphic lambda calculus with an explicit type binding construct. We relate this variant to the standard polymorphic lambda calculus, establish its type safety, and then explain the relation between this new calculus with prior calculi that support dynamic name generation.

5.1. The problem

A naive integration of casts and dynamic type with type substitution cannot ensure relational parametricity.

Say we wish to cast the untyped constant function

$$K^{\star} = \lceil \lambda x.\, \lambda y.\, x \rceil$$

to a polymorphic type. We consider two casts.

$$
\begin{aligned}
&K^{\star} : \star \Rightarrow^{p} \forall X.\, \forall Y.\, X \to Y \to X \\
&K^{\star} : \star \Rightarrow^{p} \forall X.\, \forall Y.\, X \to Y \to Y
\end{aligned}
$$

We expect the first cast to succeed and the second to fail, the latter because of parametricity. The parametricity property for the type $\forall X.\, \forall Y.\, X \to Y \to Y$ guarantees that a value of this type must be either the flipped constant function (which returns its second argument) or the undefined function (which never returns a value). So an attempt to cast the constant function (which returns its first argument) to this type should fail.

The traditional way to reduce a type application is by substitution. This cannot work in our case! To see why, consider reducing each of the above by substituting $X{:=}\mathtt{I}, Y{:=}\mathtt{I}$.

$$
\begin{aligned}
&\quad (K^{\star} : \star \Rightarrow^{p} \forall X.\, \forall Y.\, X \to Y \to X)\ \mathtt{I}\ \mathtt{I}\ 2\ 3 \\
&\longmapsto^{*} (K^{\star} : \star \Rightarrow^{p} \mathtt{I} \to \mathtt{I} \to \mathtt{I})\ 2\ 3 \\
&\longmapsto^{*} 2
\end{aligned}
$$

$$
\begin{aligned}
&\quad (K^{\star} : \star \Rightarrow^{p} \forall X.\, \forall Y.\, X \to Y \to Y)\ \mathtt{I}\ \mathtt{I}\ 2\ 3 \\
&\longmapsto^{*} (K^{\star} : \star \Rightarrow^{p} \mathtt{I} \to \mathtt{I} \to \mathtt{I})\ 2\ 3 \\
&\longmapsto^{*} 2
\end{aligned}
$$

Note how, in the second line of each reduction, the substitution has erased the difference between the two programs—the system has forgotten that the terms were once polymorphic. In particular, the type variables X and Y no longer occur in the cast; they have been replaced by $\mathtt{I}$.

Syntax

$$
\begin{array}{llcl}
\text{Types} & A, B, C & ::= & \iota \mid A{\to}B \mid X \mid \forall X.\, B \\
\text{Terms} & s, t & ::= & c \mid op(\vec{t}) \mid x \mid \lambda x{:}A.\, t \mid t\ s \mid \Lambda X.\, t \mid t\ A \mid \nu X{:=}A.\, t \\
\text{Environments} & \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, X \mid \Gamma, X{:=}A \\
 & \Delta & ::= & \cdot \mid \Delta, X \mid \Delta, X{:=}A \\
\text{Values} & v, w & ::= & c \mid \lambda x{:}A.\, t \mid \Lambda X.\, v \\
\text{Contexts} & E & ::= & [\cdot] \mid op(\vec{v}, E, \vec{t}) \mid E\ s \mid v\ E \mid \Lambda X.\, E \mid E\ A \mid \nu X{:=}A.\, E
\end{array}
$$

Type rules

$$
\frac{\Gamma, X \vdash t : B}{\Gamma \vdash \Lambda X.\, t : \forall X.\, B}\ (\textsc{TyAbs})
\qquad
\frac{\Gamma \vdash t : \forall X.\, B \quad \Gamma \vdash A}{\Gamma \vdash t\ A : B[X{:=}A]}\ (\textsc{TyApp})
\qquad
\frac{\Gamma, X{:=}A \vdash t : B \quad \Gamma \vdash A \quad X \notin \mathrm{ftv}(B)}{\Gamma \vdash \nu X{:=}A.\, t : B}\ (\textsc{New})
$$

$$
\frac{\Gamma \vdash t : B \quad (X{:=}A) \in \Gamma}{\Gamma \vdash t : B[X{:=}A]}\ (\textsc{Reveal})
\qquad
\frac{\Gamma \vdash t : B[X{:=}A] \quad (X{:=}A) \in \Gamma}{\Gamma \vdash t : B}\ (\textsc{Conceal})
$$

Reduction rules

$$
\begin{array}{rcll}
(\Lambda X.\, v)\ A & \longrightarrow & \nu X{:=}A.\, v & (\textsc{TyBeta}) \\
\nu X{:=}A.\, c & \longrightarrow & c & (\textsc{NuConst}) \\
\nu X{:=}A.\, (\lambda y{:}B.\, t) & \longrightarrow & \lambda y{:}B[X{:=}A].\, (\nu X{:=}A.\, t) & (\textsc{NuWrap}) \\
\nu X{:=}A.\, (\Lambda Y.\, v) & \longrightarrow & \Lambda Y.\, (\nu X{:=}A.\, v) \quad \text{if } Y \neq X, Y \notin \mathrm{ftv}(A) & (\textsc{NuType})
\end{array}
$$

Fig. 4. Polymorphic lambda calculus with type bindings (extends Figure 2).

Thus, we see that special run-time support is needed to enforce parametricity. In the literature, such run-time support is called *dynamic sealing*, which we review in Section 5.5. In particular, our approach is inspired by the dynamic sealing of Matthews and Ahmed [2008], the dynamic type name generation of Neis et al. [2009], and the syntactic type abstraction of Grossman et al. [2000]. Based on these ideas, we introduce an alternate semantics for the polymorphic lambda calculus as a step towards defining our polymorphic blame calculus.

5.2. Polymorphic lambda calculus with type binding

We avoid the problems above by introducing a polymorphic lambda calculus with explicit type binding, shown in Figure 4. As usual, types are augmented by adding type variables X and universal quantifiers $\forall X.\, B$, and terms are augmented by adding type abstractions $\Lambda X.\, t$ and type applications $t\ A$. The key new construct is explicit type binding, $\nu X{:=}A.\, t$.

Type environments are augmented to include, as usual, type variables X and, more unusually, type bindings $X{:=}A$. As usual, we assume an implicit side condition when writing Γ, X or $\Gamma, X{:=}A$ that X is not in Γ.

The type rules for type abstraction and application are standard (TYABS), (TYAPP). The type rule for binding augments the type environment with the binding, and a side condition ensures that free type variables do not escape the binding (NEW). Two additional type rules, which are not syntax directed, permit a type variable to be replaced by its bound type, or vice versa, within the scope of a type binding (REVEAL), (CONCEAL).

We now briefly consider the reduction rules. Our rule for type applications, instead of performing substitution, introduces an explicit type binding (TYBETA). Three new rules push explicit type bindings into the three value forms: constants (NuConst), value abstractions (NuWrap), and type abstractions (NuType). (Side conditions on the last rule avoid capture of type variables.) For example,

$$
\begin{array}{l}
\quad (\Lambda X.\, \lambda x{:}X.\, (\lambda y{:}X.\, y)\ x)\ \mathtt{I}\ 2 \\
\longmapsto (\nu X{:=}\mathtt{I}.\, \lambda x{:}X.\, (\lambda y{:}X.\, y)\ x)\ 2 \\
\longmapsto (\lambda x{:}\mathtt{I}.\, \nu X{:=}\mathtt{I}.\, (\lambda y{:}X.\, y)\ x)\ 2 \\
\longmapsto \nu X{:=}\mathtt{I}.\, (\lambda y{:}X.\, y)\ 2 \\
\longmapsto \nu X{:=}\mathtt{I}.\, 2 \\
\longmapsto 2
\end{array}
$$

by rules (TyBeta), (NuWrap), (Beta), (Beta) and (NuConst), respectively. Note that for the term $\nu X{:=}\mathtt{I}.\, (\lambda y{:}X.\, y)\ 2$ to be well-typed, the term 2 must be regarded as having type X—this is why the type rules permit both replacing a type variable by its binding and the converse.

As is well known, allowing type abstraction over terms with arbitrary effects can be problematic. As we see in Section 6.5, the same issue arises here, due to raising of blame as a possible side effect. The usual solution is to restrict type abstraction to apply only to values, as in the value polymorphism restriction of SML [Wright 1995]. We would like to do the same here, and restrict our syntax to only include type abstractions of the form $\Lambda X.\, v$. However, this would not be consistent with the reduction (NuType), which may push the non-value type binding construct underneath a type abstraction. (A similar issue arises with the reduction (Generalize), introduced in Section 6.) Instead, therefore, we allow the body of a type abstraction to be any term (hence, the term form $\Lambda X.\, t$), but only consider a type abstraction to be a value if its body is a value (hence, the value form $\Lambda X.\, v$). This further requires, unusually, that we permit reduction under type abstractions (hence, the context form $\Lambda X.\, E$).

5.3. Relation to standard calculus

We relate the polymorphic lambda calculus with type binding to the standard polymorphic lambda calculus based on type substitution. We omit the definitions of the latter to save space. We define the erasure t° from the calculus with type bindings to the standard calculus as follows:

$$
\begin{array}{rlrl}
c^\circ = & c & (t\ s)^\circ = & t^\circ\ s^\circ \\
(\boldsymbol{op}(\vec{t}))^\circ = & \boldsymbol{op}(\vec{t}^\circ) & (\Lambda X.\, t)^\circ = & \Lambda X.\, t^\circ \\
x^\circ = & x & (t\ A)^\circ = & t^\circ\ A \\
(\lambda x{:}A.\, t)^\circ = & \lambda x{:}A.\, t^\circ & (\nu X{:=}A.\, t)^\circ = & t^\circ[X{:=}A]
\end{array}
$$

The only clause of interest is that for a binder, which is erased by performing the type substitution. We also define the application of an environment to a type $\Gamma(A)$ and the erasure of environments Γ°.

$$
\begin{array}{rlrl}
(\Gamma, x{:}B)(A) = & \Gamma(A) & (\Gamma, x{:}A)^\circ = & \Gamma^\circ, x{:}\Gamma(A) \\
(\Gamma, X)(A) = & \Gamma(A) & (\Gamma, X)^\circ = & \Gamma^\circ, X \\
(\Gamma, X{:=}B)(A) = & \Gamma(A[X{:=}B]) & (\Gamma, X{:=}A)^\circ = & \Gamma^\circ
\end{array}
$$

We can now state that the polymorphic lambda calculus with bindings correctly implements the standard calculus, that is, erasure preserves types and reductions.

Proposition 5.1 (Erasure). *If* $\Gamma \vdash s : A$ *then* $\Gamma^\circ \vdash s^\circ : \Gamma(A)$*, and if* $s \longmapsto s'$ *then either* $s^\circ = s'^\circ$ *or* $s^\circ \longmapsto s'^\circ$.

5.4. Type safety

It is straightforward to show the usual type safety results for the calculus with type binding. Typically these results are formulated with respect to closed terms and empty environments, but because we allow reduction under type abstractions and binding our results are formulated with regard to terms that may contain free type variables and environments that may contain type variables and bindings (but not term variables). We let Δ range over such environments. With this caveat, we have the usual results for canonical forms, progress, and preservation.

PROPOSITION 5.2 (CANONICAL FORMS). *If $\Delta \vdash v : C$ then either*

(*1*) $v = c$ *and* $C = \iota$ *for some* c *and* ι*, or*
(*2*) $v = \lambda x{:}A.\, t$ *and* $C = A \to B$ *for some* x*,* t*,* A*, and* B*, or*
(*3*) $v = \Lambda X.\, w$ *and* $C = \forall X.\, A$ *for some* w*,* X*, and* A*.*

PROPOSITION 5.3 (PROGRESS). *If $\Delta \vdash s : A$ then either $s = v$ for some value v or $s \longmapsto s'$ for some term s'.*

PROPOSITION 5.4 (PRESERVATION). *If $\Delta \vdash s : A$ and $s \longmapsto s'$ then $\Delta \vdash s' : A$.*

5.5. Relation to dynamic type name generation

Neis et al. [2009] present a form for generating type names: $\mathsf{new}\, X \approx A \,\mathsf{in}\, t$. The main difference between our bindings and new is that new adds its binding to a global list of bindings, σ.

$$\sigma; \mathsf{new}\, X{\approx}A \,\mathsf{in}\, t \longrightarrow \sigma, X{\approx}A; t \qquad \text{if } X \notin \mathrm{dom}(\sigma)$$

Earlier versions of our system also used a global list of bindings, but two aspects of our system require the change to local bindings.

First, evaluation proceeds under Λ in our system, which makes it problematic to use the global binding approach. Let

$$s = (\lambda x{:}X.\lambda y{:}Y.x) : X \to Y \to X$$

and consider the following program and hypothetical reduction sequence.

$$\begin{array}{lll}
 & \epsilon; & \mathsf{let}\, f = \Lambda X.(\Lambda Y.s)\ X \,\mathsf{in}\, (f\ \mathtt{I}, f\ \mathtt{B}) \\
\longmapsto & Y{\approx}X; & \mathsf{let}\, f = \Lambda X.s \,\mathsf{in}\, (f\ \mathtt{I}, f\ \mathtt{B}) \\
\longmapsto & Y{\approx}X; & ((\Lambda X.s)\ \mathtt{I}, (\Lambda X.s)\ \mathtt{B}) \\
\longmapsto & Y{\approx}X, X{\approx}I; & (s, (\Lambda X.s)\ \mathtt{B})
\end{array}$$

But the next step in the sequence is problematic. We would like to α-rename the X in $\Lambda X.\, s$, but that would lose the connection with Y. Also, Y should really get two different bindings. Local bindings solve this problem by binding $Y{:=}X$ locally, inside the ΛX. The sequence with type bindings would be as follows.

$$\begin{array}{ll}
 & \mathsf{let}\, f = \Lambda X.\, (\Lambda Y.\, s)\ X \,\mathsf{in}\, (f\ \mathtt{I}, f\ \mathtt{B}) \\
\longmapsto & \mathsf{let}\, f = \Lambda X.\nu Y{:=}X.\, s \,\mathsf{in}\, (f\ \mathtt{I}, f\ \mathtt{B}) \\
\longmapsto & ((\Lambda X.\, \nu Y{:=}X.\, s)\ \mathtt{I}, (\Lambda X.\, \nu Y{:=}X.\, s)\ \mathtt{B}) \\
\longmapsto^* & ((\nu X{:=}\mathtt{I}.\, \nu Y{:=}X.\, s), (\nu X{:=}\mathtt{B}.\, \nu Y{:=}X.\, s))
\end{array}$$

Second, bindings play a role in enforcing parametricity, which we discuss in detail in Section 6.2. An earlier system, the λ_N-calculus by [Rossberg 2003], uses local type bindings, but λ_N performs scope extrusion, that is, the type bindings float upwards. The type bindings in this paper are immobile because they can trigger errors and we want those errors to occur at predictable locations.

Syntax

Types	A, B, C	$::=$	$\iota \mid A \to B \mid \star \mid X \mid \forall X.\, B$
Ground types	G, H	$::=$	$\iota \mid \star \to \star \mid X$
Terms	s, t	$::=$	$c \mid op(\vec{t}) \mid x \mid \lambda x{:}A.\, t \mid t\ s \mid s : A \Rightarrow^p B \mid s : G \Rightarrow \star \mid s\ \mathtt{is}\ G \mid$ $\mathtt{blame}\ p \mid \Lambda X.\, t \mid t\ A \mid \nu X{:=}A.\, t$
Values	v, w	$::=$	$c \mid \lambda x{:}A.\, t \mid v : G \Rightarrow \star \mid \Lambda X.\, v$
Contexts	E	$::=$	$[\cdot] \mid op(\vec{v}, E, \vec{t}) \mid E\ s \mid v\ E \mid E\ \mathtt{is}\ G \mid E : A \Rightarrow^p B \mid E : G \Rightarrow \star \mid$ $\Lambda X.\, E \mid E\ A \mid \nu X{:=}A.\, E$

Compatibility

$$\frac{A \prec B}{A \prec \forall X.\, B}\, X \notin \mathrm{ftv}(A) \qquad \frac{A[X{:=}\star] \prec B}{\forall X.\, A \prec B}$$

Reduction rules

$$(v : G \Rightarrow \star)\ \mathtt{is}\ G \longrightarrow \mathtt{true} \qquad \text{if } G \neq X \text{ for any } X \qquad (\textsc{IsTrue})$$

$$(v : H \Rightarrow \star)\ \mathtt{is}\ G \longrightarrow \mathtt{false} \qquad \text{if } G \neq H \text{ and } H \neq X \text{ for any } X \qquad (\textsc{IsFalse})$$

$$(v : X \Rightarrow \star)\ \mathtt{is}\ G \longrightarrow \mathtt{blame}\ p_{\mathtt{is}} \qquad (\textsc{IsTamper})$$

$$\nu X{:=}A.\, (v : G \Rightarrow \star) \longrightarrow (\nu X{:=}A.\, v) : G \Rightarrow \star \qquad \text{if } G \neq X \qquad (\textsc{NuGround})$$

$$\nu X{:=}A.\, (v : X \Rightarrow \star) \longrightarrow \mathtt{blame}\ p_{\nu} \qquad (\textsc{NuTamper})$$

$$v : A \Rightarrow^p (\forall X.\, B) \longrightarrow \Lambda X.\, (v : A \Rightarrow^p B) \qquad \text{if } X \notin \mathrm{ftv} A \qquad (\textsc{Generalize})$$

$$v : (\forall X.\, A) \Rightarrow^p B \longrightarrow (v\ \star) : A[X{:=}\star] \Rightarrow^p B \qquad (\textsc{Instantiate})$$

$$\text{if } B \neq \star \text{ and } B \neq \forall X'.\, B' \text{ for any } X', B'$$

Fig. 5. Polymorphic blame calculus (extends and updates Figures 2, 3 and 4).

6. POLYMORPHIC BLAME CALCULUS

Now that we have established the machinery of type binding, we consider how to combine dynamic casts with polymorphism. Figure 5 defines the polymorphic blame calculus. The syntax is the union of the constructs of the blame calculus and the polymorphic lambda calculus, and the type rules are the union of the previous type rules.

Two new cases for quantified types are added to the definition of type compatibility, one each corresponding to casts to and from quantified types. Note that these break the symmetry of compatibility enjoyed by the simply-typed blame calculus. We discuss compatibility in tandem with the corresponding reductions, in Sections 6.1 and 6.3.

The intuition behind parametric polymorphism is that functions must behave uniformly with regard to type variables. To maintain parametricity in the presence of dynamic types, we arrange that dynamic values corresponding to type variables must be treated abstractly. Recall that values of dynamic type have the form $v : G \Rightarrow \star$, where G is a ground type. A key difference in moving to polymorphism is that the ground types, in addition to including base types ι and the function type $\star \to \star$, now also include type variables X. A value of the form $v : X \Rightarrow \star$ is called a *sealed value*.

We now briefly consider the reduction rules. Tests are updated so that if the value is sealed then the test indicates blame rather than returning true or false (IsTrue), (IsFalse), (IsTamper); the reason for this change is discussed in Section 6.2. Two rules are added to push bindings into the one new value form, ground values (NuGround), (NuTamper); the motivation for these rules is also discussed in Section 6.2. Finally, the last two rules extend casts to the case where the target type or source type is a quantified type (Generalize), (Instantiate); these rules are discussed in Sections 6.1 and 6.3. A side condition on (GENERALIZE) avoids capture of type variables, and a side condition on (INSTANTIATE) avoids overlap with (GROUND) and (GENERALIZE). The rules (ISTAMPER) and (NUTAMPER) introduce two global blame labels, $p_{\mathtt{is}}$ and p_ν, which are presumed not to label any cast.

6.1. Generalization

Perhaps the two rules of greatest interest are those that cast to and from a quantified type. We begin by discussing casts to a quantified type, postponing the reverse direction to Section 6.3.

Rule (GENERALIZE) casts a value to a quantified type by abstracting over the type variable and recursively casting the value; note that the abstracted type variable may appear free in the target type of the cast. Observe that the corresponding rule for compatibility asserts that if the cast on the left of this rule is compatible then the cast on the right is also compatible.

We now have enough rules in place to revisit our problematic examples from Section 5.1. Here is the first example that yields 2, as expected.[1]

$$
\begin{array}{l}
\quad (K^\star : \star \Rightarrow^p \forall X.\, \forall Y.\, X \to Y \to X)\ \mathtt{I}\ \mathtt{I}\ 2\ 3 \\
\longmapsto^* (\Lambda X.\, \Lambda Y.\, K^\star : \star \Rightarrow^p X \to Y \to X)\ \mathtt{I}\ \mathtt{I}\ 2\ 3 \\
\longmapsto^* (\nu Y{:=}\mathtt{I}.\, \nu X{:=}\mathtt{I}.\, K^\star : \star \Rightarrow^p X \to Y \to X)\ 2\ 3 \\
\longmapsto^* \nu Y{:=}\mathtt{I}.\, \nu X{:=}\mathtt{I}. \\
\qquad K^\star\ (2 : X \Rightarrow^{\overline{p}} \star)\ (3 : Y \Rightarrow^{\overline{p}} \star) : \star \Rightarrow^p X \\
\longmapsto^* \nu Y{:=}\mathtt{I}.\, \nu X{:=}\mathtt{I}. \\
\qquad K^\star\ (2 : X \Rightarrow \star)\ (3 : Y \Rightarrow \star) : \star \Rightarrow^p X \\
\longmapsto^* \nu Y{:=}\mathtt{I}.\, \nu X{:=}\mathtt{I}.\, (2 : X \Rightarrow \star) : \star \Rightarrow^p X \\
\longmapsto^* \nu Y{:=}\mathtt{I}.\, \nu X{:=}\mathtt{I}.\, 2 \\
\longmapsto^* 2
\end{array}
$$

The first step applies (GENERALIZE) twice, while the penultimate step applies (COLLAPSE) and (ID).

The second example should fail because the constant function does not satisfy the parametricity property for $\forall X.\, \forall Y.\, X \to Y \to Y$. The reduction sequence for this example is similar to the above, save for the last steps.

$$
\begin{array}{l}
\quad (K^\star : \star \Rightarrow^p \forall X.\, \forall Y.\, X \to Y \to Y)\ \mathtt{I}\ \mathtt{I}\ 2\ 3 \\
\longmapsto^* \nu Y{:=}\mathtt{I}.\, \nu X{:=}\mathtt{I}.\, (2 : X \Rightarrow \star) : \star \Rightarrow^p Y \\
\longmapsto^* \nu Y{:=}\mathtt{I}.\, \nu X{:=}\mathtt{I}.\, \mathtt{blame}\ p \\
\longmapsto^* \mathtt{blame}\ p
\end{array}
$$

Here the penultimate step applies (CONFLICT) and the final step applies (ABORT). This yields $\mathtt{blame}\ p$, as expected.

6.2. Parametricity

We now consider some further examples, with an eye to understanding how sealing preserves parametricity.

The parametricity property for the type $\forall X.\, X \to X$ guarantees that a value of this type must be either the identity function or the undefined function. Consider the fol-

[1]Careful readers will spot that some reductions are shown out of order, so as to group related reductions together.

lowing three untyped terms.

$$\begin{aligned}
\mathit{id}^\star &= \lceil \lambda x.\, x \rceil \\
\mathit{inc}^\star &= \lceil \lambda x.\, x+1 \rceil \\
\mathit{test}^\star &= \lceil \lambda x.\, \texttt{if}\ (x\ \texttt{is}\, \texttt{I})\ \texttt{then}\ (x+1)\ \texttt{else}\ x \rceil
\end{aligned}$$

Function $\mathit{id}^\star$ is parametric, because it acts uniformly on values of all types; while functions $\mathit{inc}^\star$ and $\mathit{test}^\star$ are not, because the former acts only on integers, while the latter acts on values of any type but behaves differently on integers than on other arguments. However, casting all three functions to type $\forall X.\, X{\to}X$ yields values that satisfy the corresponding parametricity property. Casting $\mathit{id}^\star$, as one might expect, yields the identity function, while casting $\mathit{inc}^\star$ and $\mathit{test}^\star$, perhaps surprisingly, both yield the only other parametric function of this type, the everywhere undefined function.

Here is the first example.

$$\begin{aligned}
&(\mathit{id}^\star : \star \Rightarrow^p \forall X.\, X{\to}X)\ \texttt{I}\ 2 \\
\longmapsto^*\ &\nu X{:=}\texttt{I}.\, \mathit{id}^\star(2 : X \Rightarrow^{\overline{p}} \star) : \star \Rightarrow^p X \\
\longmapsto^*\ &\nu X{:=}\texttt{I}.\, 2 : X \Rightarrow \star \Rightarrow^p X \\
\longmapsto^*\ &2
\end{aligned}$$

The last step is by rules (COLLAPSE) and (ID). No matter which type and value are supplied, the casts match up, so this behaves as the identity function.

Here is the second example.

$$\begin{aligned}
&(\mathit{inc}^\star : \star \Rightarrow^p \forall X.\, X{\to}X)\ \texttt{I}\ 2 \\
\longmapsto^*\ &\nu X{:=}\texttt{I}.\, \mathit{inc}^\star(2 : X \Rightarrow^{\overline{p}} \star) : \star \Rightarrow^p X \\
\longmapsto^*\ &\nu X{:=}\texttt{I}.\, ((2 : X \Rightarrow \star \Rightarrow^q \texttt{I}) + 1) : \texttt{I} \Rightarrow^q \star \Rightarrow^p X \\
\longmapsto^*\ &\texttt{blame}\ q
\end{aligned}$$

The last step is by rules (CONFLICT) and (ABORT); here q labels casts in $\mathit{inc}^\star$ introduced by embedding typed integer addition into the untyped lambda calculus. Regardless of what type and value are supplied, the casts still do not match, so this behaves as the everywhere undefined function.

Here is the third example.

$$\begin{aligned}
&(\mathit{test}^\star : \star \Rightarrow^p \forall X.\, X{\to}X)\ \texttt{I}\ 2 \\
\longmapsto^*\ &\nu X{:=}\texttt{I}.\, \mathit{test}^\star(2 : X \Rightarrow^{\overline{p}} \star) : \star \Rightarrow^p X \\
\longmapsto^*\ &\nu X{:=}\texttt{I}.\, \texttt{if}\ (2 : X \Rightarrow \star)\ \texttt{is}\ \texttt{I}\ \texttt{then}\ \cdots\ \texttt{else}\ \cdots \\
\longmapsto^*\ &\texttt{blame}\ p_{\texttt{is}}
\end{aligned}$$

The last step is by rules (ISTAMPER) and (ABORT). Sealed values should never be examined, so rule (ISTAMPER) ensures that applying a type test to a sealed value *always* allocates blame. Rules (ISTRUE) and (ISFALSE) add side-conditions to ensure they do not overlap with (ISTAMPER). The use of type binding plays a central role: the test $(2 : \texttt{I} \Rightarrow \star)$ is $\texttt{I}$ returns true, while the test $(2 : X \Rightarrow \star)$ is $\texttt{I}$ allocates blame to $p_{\texttt{is}}$, even when X is bound to type $\texttt{I}$. Regardless of what type and value are supplied, the test always fails, so this behaves as the everywhere undefined function.

An alternative choice might be for $(v : X \Rightarrow \star)$ is G to always return false (on the grounds that a sealed value is distinct from any ground value). This choice would still retain parametricity, because under this interpretation the result of casting $\mathit{test}^\star$ would be the identity function. However, we would lose another key property; we want to ensure that casting can lead to blame but cannot otherwise change a value. In this case, casting converts $\mathit{test}^\star$ to the everywhere undefined function, which is acceptable, while converting it to the identity function would violate our criterion.

Finally, consider the polymorphic type $\forall X.\, X{\to}\star$. The parametricity property for this function states that it must be either a constant function (ignoring its argument and always returning the same value) or the everywhere undefined function. Let's see what happens when we cast $id^\star$ to this type.

$$
\begin{array}{ll}
& (id^\star : \star \Rightarrow^p \forall X.\, X{\to}\star)\ \mathtt{I}\ 2 \\
\longmapsto^* & \nu X{:=}\mathtt{I}.\, id^\star(2 : X \Rightarrow^{\overline{p}} \star) : \star \Rightarrow^p \star \\
\longmapsto^* & \nu X{:=}\mathtt{I}.\, 2 : X \Rightarrow \star \Rightarrow^p \star \\
\longmapsto^* & \nu X{:=}\mathtt{I}.\, 2 : X \Rightarrow \star \\
\longmapsto^* & \mathtt{blame}\ p_\nu
\end{array}
$$

Here rule (NUTAMPER) plays a key role, ensuring that the attempt to pass a value grounded at type X through the binder for X must fail. In an earlier system we devised that did not have bindings [Ahmed et al. 2009], this term would in fact reduce to a value of type $\star$, violating a strict interpretation of the parametricity requirement. It was only a mild violation, because the value of type $\star$ was sealed, so any attempt to examine it would fail. Still, from both a theoretical and practical point of view the current system seems preferable because it detects errors earlier, and even if the result of the offending cast is not examined.

6.3. Instantiation

Having considered casts to a quantified type, we now turn our attention to the reverse, casts from a quantified type.

Rule (INSTANTIATE) casts a value from a quantified type by instantiating the quantified type variable to the dynamic type and recursively casting the result. Observe that the corresponding rule for compatibility asserts that if the cast on the left of this rule is compatible then the cast on the right is also compatible.

The rule always instantiates with the dynamic type. Often, we are casting to the dynamic type, and in that case it seems natural to instantiate with the dynamic type itself. However, is this still sensible if we are casting to a type other than the dynamic type? We show that there is a strong sense in which instantiating to the dynamic type is always an appropriate choice.

Let us look at some examples. Let K be a polymorphically typed constant function.

$$K = \Lambda X.\, \lambda x{:}X.\, \lambda y{:}X.\, x$$

Here is an example casting to dynamic type.

$$
\begin{array}{ll}
& (K : \forall X.\, X{\to}X{\to}X \Rightarrow^p \star{\to}\star{\to}\star)\ \lceil 2 \rceil\ \lceil 3 \rceil \\
\longmapsto^* & (K\ \star : \star{\to}\star{\to}\star \Rightarrow^p \star{\to}\star{\to}\star)\ \lceil 2 \rceil\ \lceil 3 \rceil \\
\longmapsto^* & \lceil 2 \rceil
\end{array}
$$

Unsurprisingly, instantiating polymorphically typed code to $\star$ works perfectly when casting typed code to untyped code.

Perhaps more surprisingly, it also works well when casting polymorphically typed code to a different type. Because every value embeds into the type $\star$, instantiating to $\star$ yields an answer if instantiating to any type yields an answer. Here is an example of casting to static type.

$$
\begin{array}{ll}
& (K : \forall X.\, X{\to}X{\to}X \Rightarrow^p \mathtt{I} \to \mathtt{I} \to \mathtt{I})\ 2\ 3 \\
\longmapsto^* & (K\ \star : \star{\to}\star{\to}\star \Rightarrow^p \mathtt{I}{\to}\mathtt{I}{\to}\mathtt{I})\ 2\ 3 \\
\longmapsto^* & 2
\end{array}
$$

This, of course, gives us exactly the same answer as if we had instantiated K to $\mathtt{I}$ instead of $\star$:

$$\begin{array}{ll} & (K\ \mathtt{I} : \mathtt{I} \to \mathtt{I} \to \mathtt{I} \Rightarrow^p \mathtt{I} \to \mathtt{I} \to \mathtt{I})\ 2\ 3 \\ \longmapsto^* & 2 \end{array}$$

In this sense, we say that $\star$ is a Jack-of-All-Trades: if instantiating to any type yields an answer, then so does instantiating to $\star$.

However, instantiating to $\star$ is something of a *laissez faire* policy, in that it may yield an answer when a strict instantiation would fail. For instance, consider a slight variant on the example above.

$$\begin{array}{ll} & (K : \forall X.\, X \to X \to X \Rightarrow^p \mathtt{I} \to \star \to \mathtt{I})\ 2\ \lceil\mathtt{true}\rceil \\ \longmapsto^* & (K\ \star : \star \to \star \to \star \Rightarrow^p \mathtt{I} \to \star \to \mathtt{I})\ 2\ \lceil\mathtt{true}\rceil \\ \longmapsto^* & 2 \end{array}$$

Here, instantiating to $\mathtt{I}$ directly is more strict, yielding blame rather than a value.

$$\begin{array}{ll} & (K\ \mathtt{I} : \mathtt{I} \to \mathtt{I} \to \mathtt{I} \Rightarrow^p \mathtt{I} \to \star \to \mathtt{I})\ 2\ \lceil\mathtt{true}\rceil \\ \longmapsto^* & \mathtt{blame}\ \overline{p} \end{array}$$

In other words, $\star$, though a Jack-of-All-Trades, is a master of none.

To formulate the relevant property precisely, we need to capture what we mean by saying that one term yields an answer if another does, so we formulate a notion of *contextual approximation* $\sqsubseteq$.

First, we define convergence and divergence. A term that neither converges nor diverges must allocate blame.

Definition 6.1. A closed term s *converges*, written $s \Downarrow$, if $s \longmapsto^* v$ for some value v, and *diverges*, written $s \Uparrow$, if the reduction sequence beginning with s does not terminate.

Next, we define a variant of contextual approximation, where a term that allocates blame approximates every term.

Definition 6.2. Term s *approximates* term t, written $s \sqsubseteq t$, if for all evaluation contexts E we have

(1) $E[s] \Uparrow$ implies $E[t] \Uparrow$, and
(2) $E[s] \Downarrow$ implies $E[t] \Downarrow$.

We can now state the required property.

CONJECTURE 6.3 (JACK-OF-ALL-TRADES). *If* $\Delta \vdash v : \forall X.\, A$ *and* $A[X{:=}C] \prec B$ *(and hence* $A[X{:=}\star] \prec B$*) then*

$$(v\ C : A[X{:=}C] \Rightarrow^p B) \ \sqsubseteq\ (v\ \star : A[X{:=}\star] \Rightarrow^p B).$$

We discuss our attempt to prove the Jack-of-All-Trades Conjecture in Section 11.

6.4. Jack killers

The Jack-of-All-Trades Conjecture is not quite as precise as one might hope. It promises that choosing a more specific instantiation in the cast may yield more blame, but it does not constrain the blame that results—in particular, it does not promise that blame falls on the cast in question, it could fall on a different cast. We now give counterexamples showing that indeed the blame could fall elsewhere, killing any possibility of a more precise variant of Jack-of-All-Trades.

We considered changing the design of the polymorphic blame calculus to allow for a more precise variant Jack-of-All-Trades, but the alternative that we considered suffers from the same problem. In particular, we considered adding labels to sealed values and using the following rule instead of (CONFLICT) to assign blame to the sealed value instead of to the narrowing cast.

$$v : X \Rightarrow^p \star \Rightarrow^q A \longmapsto \texttt{blame}\, p \qquad \text{if } X \not\prec A \qquad (\text{CONFLICTSEAL})$$

We will give counterexamples to show that regardless of which design choice is made, it is not possible to always pin blame to the cast in question.

To be precise, Jack-of-All-Trades guarantees that if $v : \forall X.\, A$ and $A[X{:=}C] \prec B$ that

$$v\ C : A[X{:=}C] \Rightarrow^p B \sqsubseteq v\ \star : A[X{:=}\star] \Rightarrow^p B$$

The term on the left is allowed to produce blame when the term on the right does not. We want to exhibit a context E such that

$$\begin{array}{l} E[v\ C : A[X{:=}C] \Rightarrow^p B] \longmapsto^* \texttt{blame}\, q \\ E[v\ \star : A[X{:=}\star] \Rightarrow^p B] \longmapsto^* v' \end{array}$$

where q is a blame label other than p or $\bar{p}$. Further, we want to show that this can happen either with the (CONFLICT) rule as written, or with the variant rule (CONFLICTSEAL).

First, an example to show that blame can fall outside the cast in question with the (CONFLICT) rule. Take $v = \mathit{id} = \Lambda X.\, \lambda x{:}X.\, x$ and $\forall X.\, A = \mathit{Id} = \forall X.\, X \to X$ and $B = \star \to \mathtt{I} \to \mathtt{I}$ and $C = \mathit{Id}$ and $E = [\cdot]\ \mathit{inc}^\star\ 1$ where

$$\mathit{inc}^\star = (\lambda x : \star.\, (x : \star \Rightarrow^q \mathtt{I}) + 1 : \mathtt{I} \Rightarrow^r \star)$$

Then

$$\begin{array}{l} (\mathit{id}\ \mathit{Id} : \mathit{Id} \to \mathit{Id} \Rightarrow^p \star \to \mathtt{I} \to \mathtt{I})\ \mathit{inc}^\star\ 1 \longmapsto^* \texttt{blame}\, q \\ (\mathit{id}\ \star : \star \to \star \Rightarrow^p \star \to \mathtt{I} \to \mathtt{I})\ \mathit{inc}^\star\ 1 \longmapsto^* 2 \end{array}$$

giving the desired counterexample, since blame falls on q and not p. (This is not a counterexample for the alternative (CONFLICTSEAL) rule, because with the alternate rule blame falls on $\bar{p}$ and not q.)

Second, an example to show that blame can fall outside the cast in question with the alternative (CONFLICTSEAL) rule. Take $v = \mathit{id} = \Lambda X.\, \lambda x{:}X.\, x$ and $\forall X.\, A = \mathit{Id} = \forall X.\, X \to X$ and $B = \star \to \star$ and $C = \mathtt{I}$ and $E = ([\cdot] : \star \to \star \Rightarrow^q \mathit{Id})\ \mathtt{I}\ 2$. Then

$$\begin{array}{l} (\mathit{id}\ \mathtt{I} : \mathtt{I} \to \mathtt{I} \Rightarrow^p \star \to \star \Rightarrow^q \mathit{Id})\ \mathtt{I}\ 2 \longmapsto^* \texttt{blame}\, q \\ (\mathit{id}\ \star : \star \to \star \Rightarrow^p \star \to \star \Rightarrow^q \mathit{Id})\ \mathtt{I}\ 2 \longmapsto^* 2 \end{array}$$

giving the desired counterexample, since blame falls on q and not p. (This is not a counterexample for the (CONFLICT) rule, because with that rule blame falls on $\bar{p}$ and not q.)

6.5. Evaluation under type abstraction

As noted in Section 5.2, an unusual feature of our presentation is that we evaluate underneath type abstractions. We now provide an example, promised there, of why such evaluation is necessary.

Parametricity guarantees that a term of type $\forall X.\, X$ cannot reduce to a value. One term with this type is $\Lambda X.\, \texttt{blame}\ r$. In our calculus, this term is not a value, and it evaluates to $\texttt{blame}\, r$. However, if we did not evaluate under type abstractions then this term would be a value.

We want it to be the case that $v : A \Rightarrow^p \star \Rightarrow^q A$ is equivalent to v for any value v of type A. (Among other things, it is easy to show that this is a consequence of the

Jack-of-All-Trades Conjecture.) However, if $\Lambda X.\, \texttt{blame}\, r$ is a value, this is not the case.

$$\begin{array}{ll} & (\Lambda X.\, \texttt{blame}\, r) : \forall X.\, X \Rightarrow^p \star \Rightarrow^q \forall X.\, X \\ \longmapsto^* & (\Lambda X.\, \texttt{blame}\, r)\ \star : \star \Rightarrow^p \star \Rightarrow^q \forall X.\, X \\ \longmapsto^* & (\nu X{:=}\star\, .\, \texttt{blame}\, r) : \star \Rightarrow^p \star \Rightarrow^q \forall X.\, X \\ \longmapsto^* & \texttt{blame}\, r \end{array}$$

This is bad—a cast that should leave a value unchanged has instead converted it to blame!

The solution to this difficulty, as described in Section 5.2, is to permit evaluation under type abstractions, and to only regard terms of the form $\Lambda X.\, v$ as values. We conjecture that if we based our system on call-by-name rather than call-by-value that evaluation under type abstraction would not be necessary.

6.6. Type safety

The usual type safety properties hold for the polymorphic blame calculus.

LEMMA 6.4 (CANONICAL FORMS). *If $\Delta \vdash v : C$, then*

(*1*) $v = c$ *and* $C = \iota$ *for some* c *and* ι, *or*
(*2*) $v = w : G \Rightarrow \star$ *and* $C = \star$, *for some* w *and* G, *or*
(*3*) $v = \lambda x{:}A.\, t$ *and* $C = A{\to}B$, *for some* x, t, A, *and* B, *or*
(*4*) $v = \Lambda X.\, w$ *and* $C = \forall X.\, A$, *for some* w, X, *and* A.

PROPOSITION 6.5 (PRESERVATION). *If* $\Delta \vdash s : A$ *and* $s \longmapsto s'$, *then* $\Delta \vdash s' : A$.

PROPOSITION 6.6 (PROGRESS). *If* $\Delta \vdash s : A$, *then either*

(*1*) $s = v$ *for some value* v, *or*
(*2*) $s \longmapsto s'$ *for some term* s', *or*
(*3*) $s = \texttt{blame}\, p$ *for some blame label* p.

Preservation and progress on their own do not guarantee a great deal because they do not rule out blame as a result. In sections 8 and 9 we characterize situations in which blame cannot arise.

7. SUBTYPING RELATIONS

Figure 6 presents the compatibility relation and four forms of subtyping—ordinary, positive, negative, and naive. Compatibility determines when it is sensible to attempt to cast one type to another type, and the different forms of subtyping characterize when a cast cannot give rise to certain kinds of blame. All five relations are reflexive, and all four subtyping relations are transitive.

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

For example, a cast $\star \to \texttt{I} \Rightarrow \texttt{I} \to \star$ never yields blame (that is, $\star \to \texttt{I} <: \texttt{I} \to \star$), but such casts are relatively rare. Much more common is to see casts of the form $\texttt{I} \to \texttt{I} \Rightarrow \star \to \star$ or $\star \to \star \Rightarrow \texttt{I} \to \texttt{I}$, which never yield positive or negative blame, respectively (that is, $\texttt{I} \to \texttt{I} <:^+ \star \to \star$ and $\star \to \star <:^- \texttt{I} \to \texttt{I}$). The type $\texttt{I} \to \texttt{I}$ is more precise than the type $\star \to \star$ (that is $\texttt{I} \to \texttt{I} <:_n \star \to \star$), so blame always lies on the more precisely typed side of the cast.

The four definitions are related, in that $A <: B$ holds if $A <:^+ B$ and $A <:^- B$ hold (but not conversely), and $A <:_n B$ holds if and only if $A <:^+ B$ and $B <:^- A$ hold. (Note the reversal! We have $A <:^- B$ in the first and $B <:^- A$ in the second.) We

Compatibility $\boxed{A \prec B}$

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

$$\frac{A[X := \star] \prec B}{\forall X.A \prec B} \qquad \frac{A \prec B}{A \prec \forall X.B} X \notin \mathrm{ftv}(A)$$

Subtype $\boxed{A <: B}$

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

$$\frac{A[X{:=}C] <: B}{\forall X.\, A <: B} \qquad \frac{A <: B}{A <: \forall X.\, B} X \notin \mathrm{ftv}(A)$$

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

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

$$\frac{A[X{:=}\star] <:^{+} B}{\forall X.\, A <:^{+} B} \qquad \frac{A <:^{+} B}{A <:^{+} \forall X.\, B} X \notin \mathrm{ftv}(A)$$

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

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

$$\frac{A[X{:=}\star] <:^{-} B}{\forall X.\, A <:^{-} B} \qquad \frac{A <:^{-} B}{A <:^{-} \forall X.\, B} X \notin \mathrm{ftv}(A)$$

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

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

$$\frac{A[X{:=}\star] <:_n B}{\forall X.\, A <:_n B} \qquad \frac{A <:_n B}{A <:_n \forall X.\, B} X \notin \mathrm{ftv}(A)$$

Fig. 6. Subtyping Relations

tried to massage our definitions so that the first clause, like the second, would be an equivalence, but failed to do so.

We now go through the definitions in detail.

Compatibility is written $A \prec B$. It is reflexive, and the dynamic type is compatible with every other type. The remaining three compatibility rules can be read off directly from the reductions (WRAP), (INSTANTIATE), and (GENERALIZE): each cast $A \Rightarrow B$ in the reduction becomes a compatibility $A \prec B$ in the corresponding rule. The cast on the left-hand side of a reduction becomes the conclusion of the corresponding rule, and casts on the right-hand side become hypotheses. Thus, the rules ensure reduction of a compatible cast yields a term containing compatible casts.

Function compatibility is contravariant in the domain and covariant in the range, corresponding to the (WRAP) rule, which swaps source and target in the domain and

$\boxed{s \text{ sf } p}$

$$\frac{A <:^+ B \quad s \text{ sf } p}{(s : A \Rightarrow^p B) \text{ sf } p} \quad \frac{A <:^- B \quad s \text{ sf } p}{(s : A \Rightarrow^{\overline{p}} B) \text{ sf } p} \quad \frac{q \neq p \quad q \neq \overline{p} \quad s \text{ sf } p}{(s : A \Rightarrow^q B) \text{ sf } p} \quad \frac{s \text{ sf } p}{(s : G \Rightarrow \star) \text{ sf } p}$$

$$\frac{s \text{ sf } p}{(s \;\mathtt{is}\; G) \text{ sf } p} \quad \frac{q \neq p}{(\mathtt{blame}\; q) \text{ sf } p} \quad \frac{}{c \text{ sf } p} \quad \frac{\vec{t} \text{ sf } p}{(op(\vec{t})) \text{ sf } p} \quad \frac{}{x \text{ sf } p}$$

$$\frac{t \text{ sf } p}{(\lambda x : A.\, t) \text{ sf } p} \quad \frac{t \text{ sf } p \quad s \text{ sf } p}{(t\ s) \text{ sf } p} \quad \frac{t \text{ sf } p}{(\Lambda X.\, t) \text{ sf } p} \quad \frac{t \text{ sf } p}{(t\ A) \text{ sf } p} \quad \frac{t \text{ sf } p}{(\nu X{:=}A.\, t) \text{ sf } p}$$

Fig. 7. Safety for $<:^+$ and $<:^-$

preserves their order in the range. A polymorphic type $\forall X.\, A$ is compatible with type B if its instance $A[X{:=}\star]$ is compatible with B, corresponding to the (INSTANTIATE) rule. A type A is compatible with polymorphic type $\forall X.\, B$ if type A is compatible with B corresponding to the (GENERALIZE) rule (assuming X does not appear free in A, so there is no capture of bound variables).

Ordinary subtyping is written $A <: B$. It characterizes when a cast cannot give rise to blame. Every subtype of a ground type is a subtype of $\star$, because a cast from a ground type to $\star$ never allocates blame. As with all the relations, function subtyping is contravariant in the domain and covariant in the range. A polymorphic type $\forall X.\, A$ is a subtype of a type B if some instance $A[X{:=}C]$ is a subtype of B—this is the one way in which subtyping differs from all the other relations, which instantiate with $\star$ rather than an arbitrary type C. It is easy to see that $A <:^+ B$ and $A <:^- B$ together imply $A <: B$, but not conversely.

The next two relations are concerned with positive and negative blame. If reducing a cast with label p allocates blame to p we say it yields *positive* blame, and if it allocates blame to $\overline{p}$ we say it yields *negative* blame. The positive and negative subtyping relations characterize when positive and negative blame can arise. In the next section, we show that a cast from A to B with $A <:^+ B$ cannot give rise to positive blame, and with $A <:^- B$ cannot give rise to negative blame.

The two judgments are defined in terms of each other, and track the negating of blame labels that occurs in the contravariant position of function types. We have $A <:^+ \star$ and $\star <:^- B$ for every type A and B, because casting to $\star$ can never give rise to positive blame, and casting from $\star$ can never give rise to negative blame. We also have $A <:^- G$ implies $A <:^- B$, because a cast from a ground type to $\star$ cannot allocate blame, and a cast from $\star$ to any type cannot allocate negative blame.

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

8. THE BLAME THEOREM

The Blame Theorem asserts that a cast from a positive subtype cannot lead to positive blame, and a cast from a negative subtype cannot lead to negative blame. The structure of the proof is similar to a type safety proof, depending on progress and preservation lemmas. However, the invariant we preserve is not well-typing, but instead a safety relation, t sf p, as defined in Figure 7. This style of proof of the Blame Theorem was developed by Siek [2008].

A term t is safe for blame label p with respect to $<:^+$ and $<:^-$, written t sf p, if every cast with label p has a source that is a positive subtype of the target, and every cast with label $\overline{p}$ has a source that is a negative subtype of the target; we assume that $p \neq p_{\texttt{is}}$ and $p \neq p_\nu$.

LEMMA 8.1 (BLAME PROGRESS). *If* s sf p *then* $s \not\longmapsto \texttt{blame}\, p$.

LEMMA 8.2 (BLAME PRESERVATION). *If* s sf p *and* $s \longmapsto s'$, *then* s' sf p.

Positive and negative subtyping are closely related to naive subtyping.

PROPOSITION 8.3 (FACTORING). $A <:_n B$ *iff* $A <:^+ B$ *and* $B <:^- A$.

The proof of Proposition 8.3 requires four observations.

LEMMA 8.4.
If $A <:^+ B$ *and* $X \notin A$, *then* $X \notin B$.
If $A <:^- B$ *and* $X \notin B$, *then* $X \notin A$.
Given $X \notin B$, *we have* $A[X{:=}\star] <:^+ B$ *iff* $A <:^+ B$.
Given $X \notin A$, *we have* $A <:^- B[X{:=}\star]$ *iff* $A <:^- B$.

We may now characterize how positive, negative, and naive subtyping relate to positive and negative blame. Note that, typically, each cast in a source program has a unique blame label.

COROLLARY 8.5 (BLAME THEOREM). *Let* t *be a program with a subterm* $s : A \Rightarrow^p B$ *where the cast is labelled by the only occurrence of* p *in* t, *and* $\overline{p}$ *does not appear in* t.

(*1*) *If* $A <:^+ B$, *then* $t \not\longmapsto^* \texttt{blame}\, p$.
(*2*) *If* $A <:^- B$, *then* $t \not\longmapsto^* \texttt{blame}\, \overline{p}$.
(*3*) *If* $A <:_n B$, *then* $t \not\longmapsto^* \texttt{blame}\, p$.
(*4*) *If* $B <:_n A$, *then* $t \not\longmapsto^* \texttt{blame}\, \overline{p}$.

The first two results are an immediate consequence of blame progress and preservation (Lemmas 8.1 and 8.2) while the second two results are an immediate consequence of the first two and factoring (Proposition 8.3).

Because 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.

The Blame Theorem gives no guarantees regarding the two global blame labels $p_{\texttt{is}}$ and p_ν, nor does the Subtyping Theorem of the next section. We conjecture there may be an alternative design in which the `is`, Λ, and ν forms are individually labelled and the safety relations can guarantee the absence of blame going to those labels under suitable static conditions.

9. THE SUBTYPING THEOREM

The Subtyping Theorem asserts that a cast from a subtype to a supertype cannot lead to any blame whatsoever. As with the Blame Theorem, the structure of the proof is similar to that of a type safety proof, depending on progress and preservation lemmas. Again, we use a safety relation, $s\ \mathrm{sf}^{<:}\ p$, as defined in Figure 8. A term t is safe for blame label p with respect to $<:$, written $s\ \mathrm{sf}^{<:}\ p$, if every cast with label p or $\overline{p}$ has a source that is a subtype of the target; we assume that $p \neq p_{\texttt{is}}$ and $p \neq p_\nu$.

LEMMA 9.1 (SUBTYPING PROGRESS). *If* $s\ \mathrm{sf}^{<:}\ p$ *then* $s \not\longmapsto \texttt{blame}\, p$.

The preservation result is a little more complex than that for the Blame Theorem, because it involves approximation as introduced in Section 6.3.

$$\boxed{s\ \mathrm{sf}^{<:}\ p}$$

$$\frac{A <: B \quad s\ \mathrm{sf}^{<:}\ p}{(s : A \Rightarrow^{p} B)\ \mathrm{sf}^{<:}\ p} \qquad \frac{A <: B \quad s\ \mathrm{sf}^{<:}\ p}{(s : A \Rightarrow^{\overline{p}} B)\ \mathrm{sf}^{<:}\ p} \qquad \frac{q \neq p \quad q \neq \overline{p} \quad s\ \mathrm{sf}^{<:}\ p}{(s : A \Rightarrow^{q} B)\ \mathrm{sf}^{<:}\ p} \qquad \frac{s\ \mathrm{sf}^{<:}\ p}{(s : G \Rightarrow \star)\ \mathrm{sf}^{<:}\ p}$$

$$\frac{s\ \mathrm{sf}^{<:}\ p}{(s\ \mathtt{is}\ G)\ \mathrm{sf}^{<:}\ p} \qquad \frac{q \neq p \quad q \neq \overline{p}}{(\mathtt{blame}\ q)\ \mathrm{sf}^{<:}\ p} \qquad \frac{}{c\ \mathrm{sf}^{<:}\ p} \qquad \frac{\vec{t}\ \mathrm{sf}^{<:}\ p}{(op(\vec{t}))\ \mathrm{sf}^{<:}\ p} \qquad \frac{}{x\ \mathrm{sf}^{<:}\ p}$$

$$\frac{t\ \mathrm{sf}^{<:}\ p}{(\lambda x : A.\, t)\ \mathrm{sf}^{<:}\ p} \qquad \frac{t\ \mathrm{sf}^{<:}\ p \quad s\ \mathrm{sf}^{<:}\ p}{(t\ s)\ \mathrm{sf}^{<:}\ p} \qquad \frac{t\ \mathrm{sf}^{<:}\ p}{(\Lambda X.\, t)\ \mathrm{sf}^{<:}\ p} \qquad \frac{t\ \mathrm{sf}^{<:}\ p}{(t\ A)\ \mathrm{sf}^{<:}\ p} \qquad \frac{t\ \mathrm{sf}^{<:}\ p}{(\nu X{:=}A.\, t)\ \mathrm{sf}^{<:}\ p}$$

Fig. 8. Safety for $<:$

LEMMA 9.2 (SUBTYPING PRESERVATION). *Assuming Jack-of-All-Trades holds, if s $\mathrm{sf}^{<:}$ p and $s \longmapsto s'$, then either s' $\mathrm{sf}^{<:}$ p or there exists s'' such that $s'' \sqsubseteq s'$ and s'' $\mathrm{sf}^{<:}$ p.*

The proof is by case analysis on $s \longrightarrow s'$ and $s \longmapsto s'$, where the case for (INSTANTIATE) depends on Jack-of-All-Trades. We may now characterize how subtyping relates to blame.

COROLLARY 9.3 (SUBTYPING THEOREM). *Assume Jack-of-All-Trades holds, and let t be a program with a subterm $s : A \Rightarrow^{p} B$ where the cast is labelled by the only occurrence of p in t, and $\overline{p}$ does not appear in t. If $A <: B$, then $t \not\longmapsto^{*} \mathtt{blame}\ p$ and $t \not\longmapsto^{*} \mathtt{blame}\ \overline{p}$.*

The result is an immediate consequence of subtyping progress and preservation.

10. STATIC CASTS

The polymorphic lambda calculus with type bindings (Figure 4) includes two type rules that are not syntax directed, (REVEAL) and (CONCEAL). In this section, we introduce the polymorphic lambda calculus with static casts, which extends the earlier calculus by adding two new constructs so that the two type rules in question become syntax directed. The result is a calculus which syntactically records exactly where type abstraction occurs, similar in some respects to that of Grossman et al. [2000]. The more refined type information provided by the new calculus is used in the attempted proof of Jack-of-all-Trades discussed in the next section.

10.1. Polymorphic lambda calculus with static casts

We introduce the polymorphic lambda calculus with static casts in Figure 9. It proves convenient for the new constructs to use a notation similar to that for dynamic casts, and hence we call them static casts. Dynamic casts may fail and are decorated with a blame label. Static casts may not fail, and are decorated with a *binding reference*.

Static casts come in two forms, corresponding to the rules (REVEAL) and (CONCEAL) in the polymorphic lambda calculus with type binding. Assume binding $X{:=}A$ appears in the environment Γ. We reveal the binding of a type variable with the construct

$$s : B \Rightarrow^{X} B[X{:=}A]$$

and we conceal the binding with the construct

$$s : B[X{:=}A] \Rightarrow^{\overline{X}} B.$$

For convenience in the reduction rules, we use the syntax

$$s : A \Rightarrow^{P} B$$

Syntax

Binding reference	P, Q	$::=$	$X \mid \overline{X}$
Terms	s, t	$::=$	$c \mid op(\vec{t}) \mid x \mid \lambda x{:}A.\, t \mid t\ s \mid \Lambda X.\, v \mid t\ A \mid \nu X{:=}A.\, t \mid s : A \Rightarrow^{P} B$
Values	v, w	$::=$	$c \mid \lambda x{:}A.\, t \mid \Lambda X.\, v \mid v : A \Rightarrow^{\overline{X}} X$
Contexts	E	$::=$	$[\cdot] \mid op(\vec{v}, E, \vec{t}) \mid E\ s \mid v\ E \mid \Lambda X.\, E \mid E\ A \mid \nu X{:=}A.\, E \mid E : A \Rightarrow^{P} B$

Type rules

$$\text{(REVEAL)}\quad \frac{\Gamma \vdash t : B \quad (X{:=}A) \in \Gamma}{\Gamma \vdash (t : B \Rightarrow^{X} B[X{:=}A]) : B[X{:=}A]} \qquad \text{(CONCEAL)}\quad \frac{\Gamma \vdash t : B[X{:=}A] \quad (X{:=}A) \in \Gamma}{\Gamma \vdash (t : B[X{:=}A] \Rightarrow^{\overline{X}} B) : B}$$

Reduction rules

$$(\Lambda X.\, v)\ A \longrightarrow \nu X{:=}A.\, (v : B \Rightarrow^{X} B[X{:=}A]) \quad \text{if } v : B \quad \text{(TYBETA)}$$
$$\nu X{:=}A.\, (v : B \Rightarrow^{\overline{Y}} Y) \longrightarrow (\nu X{:=}A.\, v) : B \Rightarrow^{\overline{Y}} Y \quad \text{(SNU)}$$
$$v : \iota \Rightarrow^{P} \iota \longrightarrow v \quad \text{(SBASE)}$$
$$(\lambda x : A.\, t) : A{\to}B \Rightarrow^{P} A'{\to}B' \longrightarrow \lambda x{:}A'.\, (t[x{:=}(x : A' \Rightarrow^{\overline{P}} A)] : B \Rightarrow^{P} B') \quad \text{(SWRAP)}$$
$$(\Lambda X.\, v) : \forall X.\, B \Rightarrow^{P} \forall X.\, B' \longrightarrow \Lambda X.\, (v : B \Rightarrow^{P} B') \quad \text{if } X \neq P, \overline{P} \quad \text{(STYPE)}$$
$$v : X \Rightarrow^{P} X \longrightarrow v \quad \text{if } X \neq P, \overline{P} \quad \text{(SSEAL)}$$
$$v : A \Rightarrow^{\overline{X}} X \Rightarrow^{X} A \longrightarrow v \quad \text{(SCANCEL)}$$

Fig. 9. Polymorphic lambda calculus with static casts (extends and updates Figures 2 and 4).

to range over both forms, where P is a binding reference that is either X or $\overline{X}$. We write $\overline{P}$ for the involution that adds an overbar when one is missing, or removes the overbar when one is present.

With the addition of static casts, we have a new value form. It is now the case that a value of type X always has the form

$$v : A \Rightarrow^{\overline{X}} X$$

where v has type A and X is bound to A in the environment.

The rule for type application is modified to also insert a suitable static cast (TYBETA). The static cast depends upon the type of the type abstraction; it is easy to instead annotate terms to preserve this information.

We introduce a reduction rule to push type bindings through the one new value form (SNU). Surprisingly, the (SNU) reduction rule requires no side conditions; the type system already ensures that $X \neq Y$ and $X \notin \mathrm{ftv}(B)$. We also introduce reduction rules to perform static casts for each type constructor: base types (SBASE), functions (SWRAP), quantified types (STYPE), and type variables (SSEAL). The rules to push a static cast through a base type (SBASE) or a type variable (SSEAL) both resemble the rule for dynamic casts (ID).

The rule to apply a static cast to a function (SWRAP) resembles the corresponding rule for dynamic casts (WRAP). Just as (WRAP) flips the cast on the arguments and negates the blame label, (SWRAP) also flips the static cast on the arguments and negates the binding reference. One notable difference between (SWRAP) and (WRAP) is that (SWRAP) does not introduce a new wrapper function to apply the cast, but instead performs substitution directly in the body of the lambda abstraction. This greatly simplified the simulation relation used in the attempted proof of the Jack-of-All-Trades

Syntax

Terms	s, t	$::=$	$c \mid op(\vec{t}) \mid x \mid \lambda x{:}A.\, t \mid t\ s \mid s{:}A \Rightarrow^p B \mid s{:}G \Rightarrow \star \mid s \ \mathtt{is}\ G \mid \mathtt{blame}\ p \mid$ $\Lambda X.\, t \mid t\ A \mid \nu X{:=}A.\, t \mid s{:}A \Rightarrow^P B$
Values	v, w	$::=$	$c \mid \lambda x{:}A.\, t \mid v : G \Rightarrow \star \mid \Lambda X.\, v \mid v : A \Rightarrow^{\overline{X}} X$
Contexts	E	$::=$	$[\cdot] \mid op(\vec{v}, E, \vec{t}) \mid E\ s \mid v\ E \mid E \ \mathtt{is}\ G \mid E{:}A \Rightarrow^p B \mid E{:}G \Rightarrow \star \mid$ $\Lambda X.\, E \mid E\ A \mid \nu X{:=}A.\, E \mid E{:}A \Rightarrow^P B$

Reduction rules

$$v : \star \Rightarrow^P \star \longrightarrow v \qquad \text{(SDYN)}$$

Fig. 10. Polymorphic blame calculus with static casts (extends and updates Figures 2, 3, 4, 5, and 9).

Conjecture. The substitution-based approach is not viable for (WRAP) because a dynamic cast can fail, but works here because a static cast cannot fail.

The rule to apply a static cast to a quantified type (STYPE) is simpler than the corresponding rules for applying a dynamic cast. For dynamic casts we require separate rules for universal quantifiers in the source (INSTANTIATE) and in the target (GENERALIZE); while for static casts it suffices to use a single rule to handle a universal quantifier in both the source and target (STYPE), since one will be a substitution instance of the other.

Finally, if a static cast meets its negation, the two casts cancel (SCANCEL).

10.2. Relation to the calculus with type binding

We relate the polymorphic lambda calculus with static casts to the polymorphic lambda calculus with type binding. We define the erasure $t^\bullet$ from the calculus with static casts to the calculus with type binding as follows:

$$\begin{array}{rclcrcl}
c^\bullet &=& c & \quad & (\Lambda X.\, t)^\bullet &=& \Lambda X.\, t^\bullet \\
(op(\vec{t}))^\bullet &=& op(\vec{t^\bullet}) & & (t\ A)^\bullet &=& t^\bullet\ A \\
x^\bullet &=& x & & (\nu X{:=}A.\, t)^\bullet &=& \nu X{:=}A.\, t^\bullet \\
(\lambda x{:}A.\, t)^\bullet &=& \lambda x{:}A.\, t^\bullet & & (t : A \Rightarrow^P B)^\bullet &=& t^\bullet \\
(t\ s)^\bullet &=& t^\bullet\ s^\bullet & & & &
\end{array}$$

PROPOSITION 10.1 (ERASURE). *If $\Gamma \vdash s : A$ then $\Gamma \vdash s^\bullet : A$, and if $s \longmapsto s'$ then either $s^\bullet = s'^\bullet$ or $s^\bullet \longmapsto s'^\bullet$.*

10.3. Type safety

It is straightforward to show the usual type safety results for the polymorphic lambda calculus with static casts. Notably, there is now one additional canonical form, for a term whose type is a type variable.

PROPOSITION 10.2 (CANONICAL FORMS). *If $\Delta \vdash v : C$ then either*

(*1*) $v = c$ *and* $C = \iota$ *for some* c *and* ι*, or*
(*2*) $v = \lambda x{:}A.\, t$ *and* $C = A \to B$ *for some* x, t, A, *and* B*, or*
(*3*) $v = \Lambda X.\, w$ *and* $C = \forall X.\, A$ *for some* w, X, *and* A.
(*4*) $v = w : A \Rightarrow^{\overline{X}} X$ *and* $C = X$ *for some* w, X, *and* A.

PROPOSITION 10.3 (PROGRESS). *If $\Delta \vdash s : A$ then either $s = v$ for some value v or $s \longmapsto s'$ for some term s'.*

PROPOSITION 10.4 (PRESERVATION). *If $\Delta \vdash s : A$ and $s \longmapsto s'$ then $\Delta \vdash s' : A$.*

10.4. Polymorphic blame calculus with static casts

Given the above development, it is straightforward to augment the polymorphic blame calculus to include static casts, as shown in Figure 10. The syntax is just the union of the syntaxes of the previous calculi. Only one additional reduction rule is required, to apply a static cast to the dynamic type (SDYN). Analogues of the previous results to relate the calculus with static casts to the one without are straightforward, as are analogues of the type safety results, and we omit the details.

11. THE ELUSIVE JACK-OF-ALL-TRADES

We now explain the flaw in the proof of the Jack-of-All-Trades Conjecture from Ahmed et al. [2011].

CONJECTURE 11.1 (JACK-OF-ALL-TRADES CONJECTURE). *If $\Delta \vdash v : \forall X.\, A$ and $A[X{:=}C] \prec B$ (and hence $A[X{:=}\star] \prec B$) then*

$$(v\; C : A[X{:=}C] \Rightarrow^p B) \sqsubseteq (v\; \star : A[X{:=}\star] \Rightarrow^p B).$$

The flawed proof introduces a relation $s \precsim t$ (Figure 11) that is contained in $\sqsubseteq$ and tries to prove that $\precsim$ is a simulation. The $\precsim$ relation relies on the following notion of a type simulating another type.

Definition 11.2. If Σ is a map from type variables to types, its *erasure* $\Sigma^\star$ is the map that takes each X in the domain of Σ to $\star$. We say that type A *simulates* type A', written $A \precsim A'$, if there exists a type A'' and a map Σ such that $A = \Sigma(A'')$ and $A' = \Sigma^\star(A'')$.

For example, if $A = (X \to X) \to B$ and $A' = \star \to \star$ we have $A \precsim A'$ by taking $A'' = Y \to Z$ and $\Sigma = Y{:=}X \to X, Z{:=}B$ (and hence $\Sigma^\star = Y{:=}\star, Z{:=}\star$). As a second example, consider what type A may simulate a type variable X, $A \precsim X$? The answer is that X is the only type that simulates X, so $A = X$.

One of the key lemmas for proving that $\precsim$ is a simulation is that if the left-hand side is a value, then the right-hand side can catch up and also become a value (Lemma 26 in [Ahmed et al. 2011]).

LEMMA 11.3 (VALUE ON THE LEFT OF $\precsim$). *If $v \precsim t$, then $t \longmapsto^* w$ and $v \precsim w$ for some value w.*

Unfortunately, there was a missing subcase in the proof of this lemma and there is not a straightforward way to fill it in. In particular, consider the case for the (CONGS) rule:

$$\frac{s \precsim t \quad A \precsim A' \quad B \precsim B'}{s : A \Rightarrow^P B \precsim t : A' \Rightarrow^P B'} \quad (\text{CONGS})$$

Because $v = (s : A \Rightarrow^P B)$ is a value, s is a value (call it v'), $B = X$, and $P = \overline{X}$. The induction hypothesis gives a value w such that $t \longmapsto^* w$ and $v' \precsim w$. Now, we know that $X \precsim B'$, so either $B' = X$ or $B' = \star$. The $B' = \star$ case is the problematic one. In this case we have $A' = \star$ and so the right-hand side has the form $w : \star \Rightarrow^{\overline{X}} \star$ which is not a value. It can reduce in one step to a value $w : \star \Rightarrow^{\overline{X}} \star \longrightarrow w$. But then we need to show that $v' : A \Rightarrow^{\overline{X}} X \precsim w$ and we do not have a rule for this. Adding a rule for this case does not make sense. Consider what would happen when the above two terms go through a cast associated with the POSCAST rule. We get blame *on the right* in the case when $G \neq X$. (We only want blame on the left.)

$$v' : A \Rightarrow^{\overline{X}} X \Rightarrow^p X \longrightarrow v' : A \Rightarrow^{\overline{X}} X$$

$$w' : G \Rightarrow \star \Rightarrow^p X \longrightarrow \mathtt{blame}\; p$$

$$\frac{s \mathrel{\underset{\sim}{\sqsubset}} t \quad A \mathrel{\underset{\sim}{\sqsubset}} A'}{s : A \Rightarrow^{p} B \mathrel{\underset{\sim}{\sqsubset}} t : A' \Rightarrow^{p} B} \quad (\textsc{PosCast})$$

$$\frac{s \mathrel{\underset{\sim}{\sqsubset}} t \quad A \mathrel{\underset{\sim}{\sqsubset}} A'}{s : B \Rightarrow^{\bar{p}} A \mathrel{\underset{\sim}{\sqsubset}} t : B \Rightarrow^{\bar{p}} A'} \quad (\textsc{NegCast})$$

$$\texttt{blame}\, q \mathrel{\underset{\sim}{\sqsubset}} t \quad (\textsc{Blame})$$

$$\frac{s \mathrel{\underset{\sim}{\sqsubset}} t}{\Lambda Y.\, s \mathrel{\underset{\sim}{\sqsubset}} t} \quad (\textsc{LeftTyAbs})$$

$$\frac{s \mathrel{\underset{\sim}{\sqsubset}} t}{s : A \Rightarrow^{P} B \mathrel{\underset{\sim}{\sqsubset}} t} \quad (\textsc{LeftS})$$

$$\frac{s \mathrel{\underset{\sim}{\sqsubset}} t}{\nu X{:=}A.\, s \mathrel{\underset{\sim}{\sqsubset}} t} \quad (\textsc{LeftNu})$$

$$\frac{v \mathrel{\underset{\sim}{\sqsubset}} w}{v \mathrel{\underset{\sim}{\sqsubset}} w : G \Rightarrow \star} \quad (\textsc{RightGround})$$

$$\frac{v \mathrel{\underset{\sim}{\sqsubset}} w}{(\Lambda X.\, v)\, \star \mathrel{\underset{\sim}{\sqsubset}} w} \quad (\textsc{LeftTyApp})$$

$$c \mathrel{\underset{\sim}{\sqsubset}} c \quad (\textsc{CongConst})$$

$$x \mathrel{\underset{\sim}{\sqsubset}} x \quad (\textsc{CongVar})$$

$$\frac{s \mathrel{\underset{\sim}{\sqsubset}} t \quad A \mathrel{\underset{\sim}{\sqsubset}} A'}{\lambda x{:}A.\, s \mathrel{\underset{\sim}{\sqsubset}} \lambda x{:}A'.\, t} \quad (\textsc{CongAbs})$$

$$\frac{s_1 \mathrel{\underset{\sim}{\sqsubset}} t_1 \quad s_2 \mathrel{\underset{\sim}{\sqsubset}} t_2}{s_1\, s_2 \mathrel{\underset{\sim}{\sqsubset}} t_1\, t_2} \quad (\textsc{CongApp})$$

$$\frac{s \mathrel{\underset{\sim}{\sqsubset}} t \quad A \mathrel{\underset{\sim}{\sqsubset}} A'}{\Lambda Y.\, s \mathrel{\underset{\sim}{\sqsubset}} \Lambda Y.\, t} \quad (\textsc{CongTyAbs})$$

$$\frac{s \mathrel{\underset{\sim}{\sqsubset}} t}{s\, A \mathrel{\underset{\sim}{\sqsubset}} t\, A} \quad (\textsc{CongTyApp})$$

$$\frac{s \mathrel{\underset{\sim}{\sqsubset}} t}{s \;\mathsf{is}\; G \mathrel{\underset{\sim}{\sqsubset}} t \;\mathsf{is}\; G} \quad (\textsc{CongIs})$$

$$\frac{s \mathrel{\underset{\sim}{\sqsubset}} t}{s : A \Rightarrow^{p} B \mathrel{\underset{\sim}{\sqsubset}} t : A \Rightarrow^{q} B} \quad (\textsc{CongCast})$$

$$\frac{v \mathrel{\underset{\sim}{\sqsubset}} w}{v : G \Rightarrow \star \mathrel{\underset{\sim}{\sqsubset}} w : G \Rightarrow \star} \quad (\textsc{CongGround})$$

$$\frac{s \mathrel{\underset{\sim}{\sqsubset}} t \quad A \mathrel{\underset{\sim}{\sqsubset}} A' \quad B \mathrel{\underset{\sim}{\sqsubset}} B'}{s : A \Rightarrow^{P} B \mathrel{\underset{\sim}{\sqsubset}} t : A' \Rightarrow^{P} B'} \quad (\textsc{CongS})$$

$$\frac{t \mathrel{\underset{\sim}{\sqsubset}} t' \quad A \mathrel{\underset{\sim}{\sqsubset}} A'}{\nu X{:=}A.\, t \mathrel{\underset{\sim}{\sqsubset}} \nu X{:=}A'.\, t'} \quad (\textsc{CongNu})$$

Fig. 11. Simulation relation $\mathrel{\underset{\sim}{\sqsubset}}$

So it seems that the (CONGS) rule is too general in that it allows both the source and target types to vary. That is, if we only needed to handle the $B' = X$ case and not $B' = \star$, the proof would go through. However, the problem is not just in the (CONGS) rule, but many of the rules allow the types to vary when they do not need too. As it stands, when we have that $s \mathrel{\underset{\sim}{\sqsubset}} t$ with $s : A$ and $t : A'$, we only can prove $A \mathrel{\underset{\sim}{\sqsubset}} A'$ but it many places we have in fact $A = A'$. The solution may be to isolate the terms with differing types, making it the common case that the terms have the same type.

Consider the situation in the Jack-of-all-trades lemma. After one reduction, the two terms look like the following

$$E[(\nu X{:=}A.\, t : B_1 \Rightarrow^{X} B_2) : B_2 \Rightarrow^{p} B_3]$$
$$E[(\nu X{:=}\star.\, t : B_1 \Rightarrow^{X} B_2') : B_2' \Rightarrow^{p} B_3]$$

where $B_2 = B_1[X := A], B_2' = B_1[X := *]$. Ignoring the ν's for the moment (they are less important) we have a sequence of two casts, a static cast and a normal cast, with the differing types in the middle.

$$t : B_1 \Rightarrow^X B_2 \Rightarrow^p B_3$$
$$t : B_1 \Rightarrow^X B_2' \Rightarrow^p B_3$$

Here we want to allow the target type of the $\Rightarrow^X$ cast to vary. Suppose these types are function types. Then after some function application steps, the arguments have the following form

$$t_1 : B_3 \Rightarrow^{\overline{p}} B_2 \Rightarrow^{\overline{X}} B_1$$
$$t_2 : B_3 \Rightarrow^{\overline{p}} B_2' \Rightarrow^{\overline{X}} B_1$$

Here we want to allow the source type of the $\Rightarrow^{\overline{X}}$ cast to vary and not the target type. So instead of having the Jack simulation relation track just a single cast, the relation should track these pairs of casts. Note also that the types B_1 and B_3 match up exactly on both the left and right hand sides. This means that for terms not associated with these "Jack casts", we can require their types to be identical.

As with many simulation proofs, the difficulty is making sure that reducing a pair of terms in the simulation relation results in a pair of terms that are again related. Our attempts to do this have resulted in a very large numbers of simulation rules that require even more rules. Either we need to find a way to unify and simplify many of the rules or we need to apply automated theorem provers.

We also note a problem with Lemma 25 of [Ahmed et al. 2011], which reads

LEMMA 11.4. *If* $s \sqsubseteq\!\!\sim t$*,* $\Gamma \vdash s : A$*, and* $\Gamma' \vdash t : A'$*, then* $A \sqsubseteq\!\!\sim A'$*.*

This lemma is false. For example, we have $\Lambda Y.\, \lambda x{:}\mathtt{I}.\, 5 \sqsubseteq\!\!\sim \lambda x{:}\star.\, 5$ by (LEFTTYABS) and (CONGABS) but $\forall Y.\, \mathtt{I}\to\mathtt{I} \not\sqsubseteq\!\!\sim \star\to\mathtt{I}$.

12. EXISTENTIALS

In this section, we show how existentials can be encoded in terms of the existing calculus. Interestingly, this provides a convenient way to define abstract data types in a dynamically typed language.

The definition of pairing and existentials in the typed language is standard.

$$\begin{array}{rcll}
A \times B & = & \forall Z.\,(A \to B \to Z) \to Z & \\
(s,t) & = & \Lambda Z.\,\lambda k : A{\to}B{\to}Z.\, k\ s\ t & \\
\mathtt{fst}\, t & = & t\ (\lambda x : A.\, \lambda y : B.\, x), & \text{if } t : A \times B \\
\mathtt{snd}\, t & = & t\ (\lambda x : A.\, \lambda y : B.\, y), & \text{if } t : A \times B \\[1ex]
\exists X.\, B & = & \forall Z.\,(\forall X.\, B \to Z) \to Z & \\
\mathtt{pack}\,(A,s)\ \mathtt{as}\ \exists X.\, B & = & \Lambda Z.\,\lambda k : \forall X.\, B{\to}Z.\, k\ A\ s & \\
\mathtt{unpack}\,(X,y) = s\ \mathtt{in}\ t & = & s\ (\Lambda X.\, \lambda y : B.\, t), & \text{if } s : \exists X.\, B
\end{array}$$

For instance, here is a translation of a standard example from Neis et al. [2009].

$$s_0 \ = \ \mathtt{pack}\,(\mathtt{I}, (1, \lambda x : \mathtt{I}.\, -x, \lambda x : \mathtt{I}.\, x \neq 0))\ \mathtt{as}\ \exists X.\, X \times (X \to X) \times (X \to \mathtt{B})$$

Here is a correct use of the abstract type:

$$\mathtt{unpack}\,(X, (x, \mathit{toggle}, \mathit{poll})) = s_0\ \mathtt{in}\ (\mathit{poll}\ x, \mathit{poll}\ (\mathit{toggle}\ x))$$

We assume triples are built and deconstructed along the lines of pairs as specified above. This term returns $(\mathtt{true}, \mathtt{false})$. Here is an incorrect use of the abstract type:

$$\mathtt{unpack}\,(X, (x, \mathit{toggle}, \mathit{poll}) = s_0\ \mathtt{in}\ \mathtt{let}\ x' = 666\ \mathtt{in}\ (\mathit{poll}\ x', \mathit{poll}\ (\mathit{toggle}\ x'))$$

Here x' violates the invariant expected by the abstract type, which assumes the argument to *poll* and *toggle* is always either 0 or 1. But the use fails to type check, because arguments to *poll* and *toggle* must always have type X, and here the argument has type `I`. The two types are considered distinct, even though within the given scope X is bound to `I`.

Interestingly, we can extend our translation of the untyped language to support abstract types in a similar way.

$$\begin{array}{rcl}
\lceil (M, N) \rceil & = & \lceil \lambda k.\, k\ M\ N \rceil \\
\lceil \mathtt{fst}\, M \rceil & = & \lceil M\ (\lambda x.\, \lambda y.\, x) \rceil \\
\lceil \mathtt{snd}\, M \rceil & = & \lceil M\ (\lambda x.\, \lambda y.\, y) \rceil \\
\lceil \mathtt{pack}^p\ M\ \mathtt{as}\ \exists X.\, B \rceil & = & \lceil \lambda k.\, k\ M \rceil : \star \Rightarrow^p \forall Z.\, (\forall X.\, A \to Z) \to Z \Rightarrow^p \star \\
\lceil \mathtt{unpack}\, y = M\ \mathtt{in}\ N \rceil & = & \lceil M\ (\lambda y.\, N) \rceil
\end{array}$$

For instance, here is the untyped variant of the example above

$$M_0 \ = \ \lceil \mathtt{pack}^p\ (\mathtt{I}, (1, \lambda x.\, -x, \lambda x.\, x \neq 0))\ \mathtt{as}\ \exists X.\, X \times (X \to X) \times (X \to \mathtt{B}) \rceil$$

Note that we could replace `B` by $\star$ above if wished—the important thing is to use the types to clarify where abstract values of type X are expected or returned. Here is a correct use of the abstract type:

$$\lceil \mathtt{unpack}\ (x, \mathit{toggle}, \mathit{poll}) = s_0\ \mathtt{in}\ (\mathit{poll}\ x, \mathit{poll}\ (\mathit{toggle}\ x)) \rceil$$

Again, this term returns $\lceil (\mathtt{true}, \mathtt{false}) \rceil$. Here is an incorrect use of the abstract type:

$$\lceil \mathtt{unpack}\ (x, \mathit{toggle}, \mathit{poll}) = M_0\ \mathtt{in}\ \mathtt{let}\ x' = 666\ \mathtt{in}\ (\mathit{poll}\ x', \mathit{poll}\ (\mathit{toggle}\ x')) \rceil$$

This version type checks, but at run-time the attempt of *poll* or *toggle* to access the integer value 666 will return $\mathtt{blame}\ \overline{p}$, indicating that the environment containing the abstract type has accessed it incorrectly. This is because *poll* expects, and *toggle* expects and returns, values of the form $M : \star \Rightarrow X$. Thus, our notation gives rise to a convenient way to support abstract data types even in an untyped language.

Observe that the typed and untyped interpretations are closely related:

$$\begin{array}{rcl}
\lceil (M, N) \rceil : \star \Rightarrow^p A \times B & = & (\lceil M \rceil : \star \Rightarrow^p A, \lceil N \rceil : \star \Rightarrow^p B) \\
\lceil \mathtt{pack}^p\ M\ \mathtt{as}\ \exists X.\, B \rceil : \star \Rightarrow^p \exists X.\, B & = & \mathtt{pack}\ (\star, \lceil M \rceil : \star \Rightarrow^p B[X{:=}\star])\ \mathtt{as}\ \exists X.\, B
\end{array}$$

Here is the proof of the first:

$$\begin{array}{rl}
& \lceil (M, N) \rceil : \star \Rightarrow^p A \times B \\
= & \lceil \lambda k.\, k\ M\ N \rceil : \star \Rightarrow^p \forall Z.\, (A \to B \to Z) \to Z \\
= & \Lambda Z.\, (\lceil \lambda k.\, k\ M\ N \rceil : \star \Rightarrow^p (A \to B \to Z) \to Z) \\
= & \Lambda Z.\, \lambda k : A {\to} B {\to} Z.\, k\ (\lceil M \rceil : \star \Rightarrow^p A)\ (\lceil N \rceil : \star \Rightarrow^p B) : Z \Rightarrow^{\overline{p}} \star \Rightarrow^p Z \\
= & \Lambda Z.\, \lambda k : A {\to} B {\to} Z.\, k\ (\lceil M \rceil : \star \Rightarrow^p A)\ (\lceil N \rceil : \star \Rightarrow^p B) \\
= & (\lceil M \rceil : \star \Rightarrow^p A, \lceil N \rceil : \star \Rightarrow^p B)
\end{array}$$

Here is the proof of the second:

$$\begin{array}{rl}
& \lceil \mathtt{pack}^p\ M\ \mathtt{as}\ \exists X.\, B \rceil : \star \Rightarrow^p \exists X.\, B \\
= & \lceil \lambda k.\, k\ M \rceil : \star \Rightarrow^p \forall Z.\, (\forall X.\, B \to Z) \to Z \Rightarrow^p \star \Rightarrow^p \forall Z.\, (\forall X.\, B \to Z) \to Z \\
= & \lceil \lambda k.\, k\ M \rceil : \star \Rightarrow^p \forall Z.\, (\forall X.\, B \to Z) \to Z \\
= & \Lambda Z.\, (\lceil \lambda k.\, k\ M \rceil : \star \Rightarrow^p \forall Z.\, (\forall X.\, B \to Z) \to Z) \\
= & \Lambda Z.\, \lambda k : \forall X.\, B {\to} Z.\, k\ \star\ (\lceil M \rceil : \star \Rightarrow^p B[X{:=}\star]) : Z \Rightarrow^{\overline{p}} \star \Rightarrow^p Z \\
= & \Lambda Z.\, \lambda k : \forall X.\, B {\to} Z.\, k\ \star\ (\lceil M \rceil : \star \Rightarrow^p B[X{:=}\star]) \\
= & \mathtt{pack}\ (\star, \lceil M \rceil : \star \Rightarrow^p B[X{:=}\star])\ \mathtt{as}\ \exists X.\, B
\end{array}$$

Similar rules also apply to `fst`, `snd`, and `unpack`.

13. RELATED WORK

Run-time sealing. Matthews and Ahmed [2008] present semantics for a multi-language system (Scheme and ML) that enforces the parametricity of ML values with polymorphic type (with embedded Scheme values). Their system places boundaries between the two languages. Their boundaries roughly correspond to a combination of a static and dynamic cast in our system. The contributions of our work with respect to the work of Matthews and Ahmed [2008] is that 1) we tease apart the notion of dynamic casting and sealing, associating sealing with type abstraction instead of the interface between languages, and 2) we establish the blame and subtyping theorems and present the Jack-of-All-Trades principle.

Syntactic type abstraction. Grossman et al. [2000] develop a general theory of syntactic type abstraction in which multiple agents interact and have varying degrees of knowledge regarding the types at the interfaces between agents. Their general theory can be used to express the type abstraction in the polymorphic lambda calculus, as well as many other kinds of syntactic abstractions. They present two systems, a simple two-agent system and a multi-agent system. The two-agent system can handle a program with one type abstraction whereas the multi-agent system is needed for arbitrary programs, using one agent per type abstraction. However, the multi-agent system adds considerable complexity for generality that is unnecessary in our setting. The advantage of our system is that it scales up to handle arbitrary number of type abstractions while retaining much of the simplicity of the two-agent system.

Sulzmann et al. [2007] develop an extension of System F with type equality coercions. Their coercions closely resemble the static casts of this paper, including the reduction rules. Their system does not have an analogue of our type bindings and instead uses substitution to perform type application.

Integrating static and dynamic. Tobin-Hochstadt and Felleisen [2006] formalize the interaction between static and dynamic typing at the granularity of modules and develop a precursor to the Blame Theorem. Wadler and Findler [2009] design the blame calculus drawing on the blame tracking of higher-order contracts [Findler and Felleisen 2002], and prove the Blame Theorem.

Gronski et al. [2006] explore the interaction of type `Dynamic` with refinement types and first-class types, that is, allowing types to be passed to and returned from functions. This provides a form of polymorphism, but not relational parametricity.

In the language Thorn, Wrigstad et al. [2010] show how to integrate typed and untyped code, using *like types* to bridge the gap in a way that better enables compiler optimizations in statically typed regions of code. Their formal development includes classes and objects but not polymorphism.

14. CONCLUSION

We have extended the blame calculus with support for first-class parametric polymorphism, using explicit type binding to maintain relational parametricity for values of polymorphic type. Our calculus supports casts between the dynamic type and polymorphic types. When casting from a polymorphic type, our system instantiates the type variable with the dynamic type, a choice justified by the Jack-of-All-Trades Conjecture: if instantiating a type parameter to any given type yields an answer then instantiating that type parameter to the dynamic type yields the same answer. Unfortunately, the proof of this principle, and its corollary the strong Subtyping Theorem, remain an open problem. We have proved the Blame Theorem, so in the new polymorphic blame calculus, “well-typed programs can’t be blamed”.

Looking forward, there are interesting questions regarding how to extend this work to subset and dependent types. Ultimately we hope to obtain a language with a full spectrum type system, supporting dynamic typing all the way to total correctness.

Acknowledgments

Our thanks to Jacob Matthews and Robby Findler for their support and participation in discussions of this work. Siek's work on this paper was supported by NSF grant 0846121 and by a Distinguished Visiting Fellowship from the Scottish Informatics and Computer Science Alliance.

REFERENCES

Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

Amal Ahmed, Jacob Matthews, Robert Bruce Findler, and Philip Wadler. Blame for all. In *Workshop on Script-to-Program Evolution (STOP)*, pages 1–13, 2009.

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for All. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2011.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009. ISBN 978-0-262-06275-6.

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

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

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *ACM Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 223–232, September 1993.

Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop (Scheme)*, pages 93–104, September 2006.

Dan Grossman, Greg Morrisett, and Steve Zdancewic. Syntactic type abstraction. *ACM Transactions on Programming Languages and Systems*, 22(6):1037–1080, November 2000.

Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *Dynamic Languages Symposium (DLS)*, pages 29–40, 2007.

Robert Harper. *Practical Foundations for Programming Languages*. 2007. Working Draft.

Anders Hejlsberg. C# 4.0 and beyond by anders hejlsberg. Microsoft Channel 9 Blog, April 2010.

Anders Hejlsberg. Introducting typescript. Microsoft Channel 9 Blog, October 2012.

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)*, pages 3–10, January 2007.

Erik Meijer. Static typing where possible, dynamic typing where needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, October 2004.

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

Georg Neis, Derek Dreyer, and Andreas Rossberg. Non-parametric parametricity. In *ACM International Conference on Functional Programming (ICFP)*, pages 135–148, September 2009.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *IFIP International Conference on Theoretical Computer Science*, pages 437–450, August 2004.

Benjamin Pierce and Eijiro Sumii. Relating cryptography and polymorphism. Manuscript, 2000. URL `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.

Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *ACM Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 241–252, 2003.

Jeremy G. Siek. A theory of gradual typing (draft). September 2008.

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

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *ACM Workshop on Types in Languages Design and Implementation (TLDI)*, pages 53–66, 2007.

Satish Thatte. Type inference with partial types. In *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 317 of *Lecture Notes in Computer Science*, pages 615–629. Springer-Verlag, 1988.

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

Julien Verlaguet. Facebook: Analyzing PHP statically. In *Commercial Users of Functional Programming (CUFP)*, September 2013.

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

Andrew K. Wright. Simple imperative polymorphism. *Higher-Order and Symbolic Computation*, 8(4):343–355, Dec. 1995.

Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 377–388, 2010.