Theorems for Free for Free: Parametricity, With and Without Types

AMAL AHMED, Northeastern University, USA
DUSTIN JAMNER, Northeastern University, USA
JEREMY G. SIEK, Indiana University, USA
PHILIP WADLER, University of Edinburgh, UK

The polymorphic blame calculus integrates static typing, including universal types, with dynamic typing. The primary challenge with this integration is preserving parametricity: even dynamically typed code should satisfy it once it has been cast to a universal type. Ahmed et al. (2011) employ runtime type generation in the polymorphic blame calculus to preserve parametricity, but a proof that it does so has been elusive. Matthews and Ahmed (2008) gave a proof of parametricity for a closely related system that combines ML and Scheme, but later found a flaw in their proof. In this paper we present an improved version of the polymorphic blame calculus and we prove that it satisfies relational parametricity. The proof relies on a step-indexed Kripke logical relation. The step-indexing is required to make the logical relation well defined in the case for the dynamic type. The possible worlds include the mapping of generated type names to their types and the mapping of type names to relations. We prove the Fundamental Property of this logical relation and that it is sound with respect to contextual equivalence. To demonstrate the utility of parametricity in the polymorphic blame calculus, we derive two free theorems.

CCS Concepts: • **Software and its engineering → Functional languages**; **Polymorphism**; **Semantics**;

Additional Key Words and Phrases: parametricity, dynamic typing, gradual typing, logical relation

ACM Reference Format:
Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free: Parametricity, With and Without Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 39 (September 2017), 28 pages.
https://doi.org/10.1145/3110283

1 INTRODUCTION

The last decade has delivered considerable progress regarding the integration of static and dynamic typing, both in theory and in practice. On the practical side, languages such as TypeScript [Bierman et al. 2014; Hejlsberg 2012], Hack [Verlaguet 2013], Flow [Chaudhuri 2014], Dart [Bracha 2011], and C# [Hejlsberg 2010] combine elements of static and dynamic typing. Over a million lines of TypeScript code went into the Microsoft Azure portal [Turner 2014]. On the theory side, in the last decade researchers have studied *gradual typing* [Allende et al. 2013; Gronski et al. 2006; Ina and Igarashi 2011; Siek and Taha 2006; Swamy et al. 2014], *multi-language integration* [Matthews and Findler 2007; Tobin-Hochstadt and Felleisen 2006], the *blame calculus* [Wadler and Findler 2009], and *manifest contracts* [Greenberg et al. 2010]. In fact, this area of research has a long history, including the *dynamic type* of Abadi et al. [1991], the *quasi-static types* of Thatte [1990], the *coercions*

2475-1421/2017/9-ART39
https://doi.org/10.1145/3110283

of Henglein [1994], the *contracts* of Findler and Felleisen [2002], and the *dynamic dependent types* of Ou et al. [2004].

The integration of dynamic typing with languages that include polymorphism (i.e., type abstraction) is particularly delicate. Abadi et al. [1991] remark that their `typecase` construct for eliminating values of type `Dynamic` could be used to violate type abstraction and suggest static restrictions to prevent that. Leroy and Mauny [1991] implement this approach in CAML as well as a non-parametric approach. A few years later, Abadi et al. [1995] allude to using runtime type generation (RTG) as a mechanism for protecting abstraction. Indeed, Sewell [2001] uses RTG to protect abstractions in distributed systems and Rossberg [2003, 2006] uses RTG for safe dynamic linking. Rossberg [2003] proves an Opacity Theorem, showing that substitution of different values at abstract type does not affect reduction.

The strongest correctness criteria regarding the protection of type abstractions is *relational parametricity* [Reynolds 1983]. The idea is expressed in terms of a type-indexed relation, a *logical relation*, that captures when two expressions exhibit the same behavior when viewed at a particular type. Consider the following traditional example [Pitts 1998a]

$$(0, (\lambda(x{:}\mathsf{int}).\, 1 - x), (\lambda(x{:}\mathsf{int}).\, x = 0)) \quad \text{vs.} \quad (1, (\lambda(x{:}\mathsf{int}).\, 1 - x), (\lambda(x{:}\mathsf{int}).\, x = 1))$$

each of which may be given either of the types

$$\mathsf{int} \times (\mathsf{int} \to \mathsf{int}) \times (\mathsf{int} \to \mathsf{bool}) \qquad \text{vs.} \qquad \exists X.X \times (X \to X) \times (X \to \mathsf{bool}).$$

The two expressions differ when assigned the first type, but behave identically when assigned the second. To prove that equivalence, one would choose the relation $R = \{(0, 1), (1, 0)\}$ for X. The Abstraction Theorem of Reynolds [1983] proves that the polymorphic λ-calculus preserves its type abstractions by way of showing that every well-typed expression is related to itself in the logical relation. This kind of theorem is now referred to as the Fundamental Property of a logical relation.

Matthews and Ahmed [2008] studied the integration of Scheme and ML, using RTG to protect ML's polymorphic functions from the runtime type tests—e.g., checking whether a value is an integer, a Boolean, or a function—common to dynamic languages like Scheme. For their ML+Scheme multi-language, they defined a step-indexed logical relation indexed by ML types and a dynamic type (TST, the Scheme type) and presented a proof of the Fundamental Property for the multi-language. Unfortunately, they later discovered a flaw in the proof. Ahmed et al. [2011] fixed this proof by introducing possible worlds that map a type variable to a tuple of the generated type name, the instantiating type, and a relation to use when relating values at the type variable. This line of work was put on hold but the above-cited unpublished manuscript is available.

Neis et al. [2011] study G, a polymorphic language extended with Girard's J operator but without the type `Dynamic`. The J operator is a non-parametric cast that tests for type equality at runtime, after abstract types have been replaced by their instantiating types. Neis et al. [2011] define a Kripke logical relation, with some similarities to the one we use here, that supports reasoning about program equivalence and they prove the Fundamental Property. This logical relation, however, does not ensure that expressions of polymorphic type behave parametrically. It instead associates parametric behavior with type names that result from RTG. However, they show that parametric behavior can be obtained by wrapping expressions in a type-directed manner with appropriate uses of RTG. We have investigated whether a polymorphic language with the dynamic type could be compiled to G in a way that would ensure parametricity (using recursive types to encode the dynamic type), but it seems that the results in Neis et al. [2011] are not general enough to make this possible. Nevertheless, more investigation would be helpful to clarify the relationship between the polymorphic blame calculus and G.

In this paper we pick up where Ahmed et al. [2011] left off, but relocate to the more general setting of the polymorphic blame calculus (λB) of Ahmed et al. [2011]. The relocation came with many technical changes at the heart of the semantics and logical relation, including the mechanics of type application and run-time type generation. The λB calculus combines the polymorphic λ-calculus of Girard [1972] and Reynolds [1974] with the blame calculus of Wadler and Findler [2009] (minus refinement types). In Section 2 we review these three calculi and make improvements to the polymorphic blame calculus. We give the formal definition of the new version of the polymorphic blame calculus in Section 3. With the stage set, Section 4 defines a Kripke logical relation, proves the Fundamental Property, and proves soundness with respect to contextual equivalence. To demonstrate the utility of our logical relation, in Section 5 we derive free theorems in the tradition of Wadler [1989]. We prove a theorem about the K combinator and a theorem about how rearrangement functions commute with the *map* function.

To summarize, this paper makes the following technical contributions.

- An improved polymorphic blame calculus (λB) that enables a natural notion of parametricity (Sections 2.4 and 3).
- The definition of a Kripke logical relation for λB for program equivalence and parametricity (Section 4.2).
- A proof of the Fundamental Property (Section 4.3).
- A proof of Soundness with respect to Contextual Equivalence (Section 4.3).
- Two examples of free theorems derived from our parametricity result (Section 5).

The complete definitions and proofs are in the accompanying technical report [Ahmed et al. 2017]. We explain the relationship between this result and those in the literature in Section 6. We discuss further research and conclude in Section 7.

This paper subsumes and improves on sections 1 through 4 of an unpublished manuscript by Siek and Wadler [2016].

2 THE ROAD TO POLYMORPHIC BLAME

The polymorphic blame calculus combines the polymorphic λ-calculus [Girard 1972; Reynolds 1974] with the blame calculus [Wadler and Findler 2009]. We review the polymorphic λ-calculus and its notion of parametricity in subsection 2.1. We review the blame calculus, and how it integrates typed and untyped languages using casts and the dynamic type, in subsection 2.2. We then review the design considerations in prior versions of the polymorphic blame calculus of Ahmed et al. [2011] (subsection 2.3) before presenting our improvements to it in subsection 2.4.

2.1 Review of the Polymorphic Lambda Calculus and Parametricity

The polymorphic λ-calculus (λF) extends the λ-calculus to capture the essence of generics. That is, it enables code to be reused on data of different types. λF adds two language constructs to simply typed λ-calculus: type abstraction $\Lambda X.e$ and type application $e\ [B]$. Type abstraction parameterizes an expression e with respect to a type variable X. Values of type X may flow through the expression e but e may not manipulate those values in a way that depends on their actual type. A simple example of a type abstraction is the swap function that exchanges the two elements of a pair:

$$swap : \forall X.\, \forall Y.\, X \times Y \to Y \times X$$
$$swap = \Lambda X.\Lambda Y.\lambda(p : X \times Y).\, (\mathsf{snd}(p), \mathsf{fst}(p))$$

A type abstraction $\Lambda X.e$ has universal (or polymorphic) type, written $\forall X.\, A$, meaning that e has type A for any type X. One can nest type abstractions, as in *swap* above, to parameterize over multiple types.

$$
\begin{array}{lcl}
\mathcal{E}\,[\![A]\!]\,\rho & = & \{(e_1,e_2) \mid \exists v_1, v_2.\ e_1 \longrightarrow^* v_1 \text{ and } e_2 \longrightarrow^* v_2 \text{ and } (v_1,v_2) \in \mathcal{V}\,[\![A]\!]\,\rho\} \\
\mathcal{V}\,[\![X]\!]\,\rho & = & \rho(X) \\
\mathcal{V}\,[\![\mathsf{int}]\!]\,\rho & = & \{(n,n) \mid n \in \mathbb{Z}\} \\
\mathcal{V}\,[\![\mathsf{bool}]\!]\,\rho & = & \{(b,b) \mid b \in \mathbb{B}\} \\
\mathcal{V}\,[\![A \times B]\!]\,\rho & = & \{(p_1,p_2) \mid (\mathsf{fst}(p_1),\mathsf{fst}(p_2)) \in \mathcal{E}\,[\![A]\!]\,\rho \text{ and } (\mathsf{snd}(p_1),\mathsf{snd}(p_2)) \in \mathcal{E}\,[\![A]\!]\,\rho\} \\
\mathcal{V}\,[\![A \to B]\!]\,\rho & = & \{(v_f,v_g) \mid \forall (v_1,v_2) \in \mathcal{V}\,[\![A]\!]\,\rho.\ (v_f\ v_1, v_g\ v_2) \in \mathcal{E}\,[\![B]\!]\,\rho\} \\
\mathcal{V}\,[\![\forall X.\,A]\!]\,\rho & = & \{(v_1,v_2) \mid \forall B_1,B_2,R \subseteq B_1 \times B_2.\ (v_1\ [B_1], v_2\ [B_2]) \in \mathcal{E}\,[\![A]\!]\,\rho(X{:=}R)\} \\
\mathcal{V}\,[\![\exists X.\,A]\!]\,\rho & = & \{(\mathsf{pack}(B_1,v_1),\mathsf{pack}(B_2,v_2)) \mid \exists R \subseteq B_1 \times B_2.\ (v_1,v_2) \in \mathcal{V}\,[\![A]\!]\,\rho(X{:=}R)\}
\end{array}
$$

Fig. 1. A Logical Relation for λF.

Type application $e\ [B]$ enables the use of a type abstraction (the result of e) by instantiating its type variable at type B. To swap a pair of an integer and a bool, we apply *swap* to int, bool, and then to the pair. We swap back similarly.

$$\mathit{swap}\ [\mathsf{int}]\ [\mathsf{bool}]\ (1,\mathsf{true}) \longrightarrow^* (\mathsf{true},1)$$
$$\mathit{swap}\ [\mathsf{bool}]\ [\mathsf{int}]\ (\mathsf{true},1) \longrightarrow^* (1,\mathsf{true})$$

The following example is rejected by the type system of λF because it tries to add the elements of the pair, which would depend on both elements having type int.

$$\mathit{bad} = \Lambda X.\Lambda Y.\lambda(p : X \times Y).\ \mathsf{fst}(p) + \mathsf{snd}(p)$$

In return for this restriction on type abstractions, λF provides a reasoning principle called *relational parametricity* [Reynolds 1983], for establishing when two expressions of the same type A have identical behavior, which is expressed in terms of a *logical relation*. Figure 1 defines the logical relation for call-by-value λF, consisting of a relation $\mathcal{E}\,[\![A]\!]$ over pairs of expressions, and an auxiliary relation $\mathcal{V}\,[\![A]\!]$ over pairs of values; both relations take an extra argument, an environment ρ mapping type variables to relations on values. Relation $\mathcal{E}\,[\![A]\!]$ holds when two expressions behave the same: when they evaluate to values that behave the same according to $\mathcal{V}\,[\![A]\!]$.

The relation $\mathcal{V}\,[\![A]\!]$ is defined by induction on the type A. For integers and Booleans, it requires the two values to be literally the same. For two pairs of type $A \times B$, the first elements and second elements must behave the same, respectively. For two functions of type $A \to B$ to behave the same, they are only required to produce outputs related at B when given inputs related at A. (There need not be any syntactic similarity between the two functions.) For two type abstractions of type $\forall X.\,A$ to behave the same, their instantiations must behave the same. However, because the two abstractions do not manipulate values of type X, it is not required that the values at type X behave the same. Instead, they can be related according to *any* arbitrary relation R! The reason for the mapping ρ becomes apparent: it maps each type variable X to a relation R that says how values at type X are related. Dually, for two existential packages to be related at type $\exists X.\,A$, there must *exist* some relation R such that the packed values v_1 and v_2 are related at type A under an environment extended with X bound to R.

The Abstraction Theorem of Reynolds [1983] (aka. Fundamental Property, aka. Parametricity) says that every well-typed expression e of λF behaves the same as itself according to its type A, that is, $(e,e) \in \mathcal{E}\,[\![A]\!]\,\rho$. At first glance this is underwhelming; of course e behaves the same as itself! The powerful part is that e behaves "according to its type A". It is powerful enough to provide behavioral guarantees, which Wadler [1989] christened *theorems for free*.

Consider again the *swap* function of type $\forall X.\,\forall Y.\,X \times Y \to Y \times X$. The following free theorem completely determines the output of *swap* and any other function with the same type.

Theorem 2.1. *(A Free Theorem of $\forall X.\,\forall Y.\,X{\times}Y \to Y{\times}X$)*
Suppose e is any expression of type $\forall X.\,\forall Y.\,X{\times}Y \to Y{\times}X$. Let A and B be arbitrary types. For any v_1 of type A and v_2 of type B,

$$e\ [A]\ [B]\ (v_1, v_2) \longrightarrow^* (v_2, v_1)$$

Proof (sketch)

By the Fundamental Property, we have $(e, e) \in \mathcal{E}\,[\![\forall X.\,\forall Y.\,X{\times}Y \to Y{\times}X]\!]$. Choose $R_1 = \{(v_1, v_1)\}$ and $R_2 = \{(v_2, v_2)\}$. So $e\ [A]\ [B] \longrightarrow^* v$ for some v and

$$(v, v) \in \mathcal{E}\,[\![X{\times}Y \to Y{\times}X]\!]\,(X{:=}R_1, Y{:=}R_2)$$

Note that the input (v_1, v_2) is related to itself at $X \times Y$

$$((v_1, v_2), (v_1, v_2)) \in \mathcal{V}\,[\![X \times Y]\!]\,(X{:=}R_1, Y{:=}R_2)$$

so the outputs of v are related at type $Y \times X$.

$$(v\ (v_1, v_2), v\ (v_1, v_2)) \in \mathcal{E}\,[\![Y{\times}X]\!]\,(X{:=}R_1, Y{:=}R_2)$$

Thus, the two copies of $v\ (v_1, v_2)$ reduce to some pairs p_1 and p_2 related at $Y \times X$.

$$v\ (v_1, v_2) \longrightarrow^* p_1 \qquad v\ (v_1, v_2) \longrightarrow^* p_2 \qquad (p_1, p_2) \in \mathcal{V}\,[\![Y{\times}X]\!]\,(X{:=}R_1, Y{:=}R_2)$$

With a little work we obtain $p_1 = (v_{11}, v_{12})$ and $p_2 = (v_{21}, v_{22})$. We have v_{11} and v_{21} related at Y, therefore $(v_{11}, v_{21}) \in R_2$. Which means that $v_{11} = v_{21} = v_2$. Similarly, we have v_{12} and v_{22} related at X, therefore $(v_{12}, v_{22}) \in R_1$. Which means that $v_{12} = v_{22} = v_1$. So we have shown $e\ [A]\ [B]\ (v_1, v_2) \longrightarrow^* (v_2, v_1)$. □

If we had chosen a weaker polymorphic type we would have obtained a weaker theorem. For example, for the type $\forall X.\,X{\times}X \to X{\times}X$ the free theorem would be $e\ [A]\ (v_1, v_2) \longrightarrow^* (v_3, v_4)$ for some v_3, v_4 where $v_3 \in \{v_1, v_2\}$ and $v_4 \in \{v_1, v_2\}$.

2.2 Review of the Blame Calculus

The blame calculus defines interaction between a typed and untyped λ-calculus. It introduces a dynamic type $\star$ to characterize expressions of statically unknown type and one language construct: the *cast*

$$(e : A \xRightarrow{p} B)$$

which checks whether the value produced by e has type B. If not, it ascribes fault to the *blame label* p. We abbreviate a sequence of casts in the obvious way,

$$e : A_1 \xRightarrow{p} A_2 \xRightarrow{q} A_3 \stackrel{\text{def}}{=} (e : A_1 \xRightarrow{p} A_2) : A_2 \xRightarrow{q} A_3$$

One scenario is that a programmer begins with untyped code and then adds types. Here is a program assembling untyped components. Untyped code is surrounded by ceiling brackets, $\lceil\cdot\rceil$.

$$\begin{array}{l}\texttt{let}\ \mathit{inc}^\star = \lceil\lambda(x).\,x + 1\rceil\ \texttt{in}\\ \texttt{let}\ \mathit{twice}^\star = \lceil\lambda(f).\,\lambda(x).\,f\ (f\ x)\rceil\ \texttt{in}\\ \lceil\mathit{twice}^\star\ \mathit{inc}^\star\ 0\rceil\end{array}$$

It evaluates to $\lceil 2\rceil : \star$.

Following a slogan of Dana Scott [Harper 2013; Statman 1991] we treat "untyped as unityped": untyped code is typed code where every term has the dynamic type $\star$. By convention, we append $\star$ to the name of untyped functions to distinguish them from their typed counterpart.

It is trivial to rewrite a three-line program to have types, but we wish to add types gradually: our technique should work as well when each one-line definition is replaced by a thousand-line module. We use casts to manage the transition between typed and untyped code.

Here is our program again but we add types to the *twice* component and insert a cast from its type to $\star$.

$$\begin{array}{l}\texttt{let } \mathit{inc}^\star = \lceil \lambda(x).\, x+1 \rceil \texttt{ in} \\ \texttt{let } \mathit{twice} = (\lambda(f{:}\mathsf{int}\to\mathsf{int}).\, \lambda(x{:}\mathsf{int}).\, f\ (f\ x)) \texttt{ in} \\ \texttt{let } \mathit{twice}^\star = (\mathit{twice} : (\mathsf{int}\to\mathsf{int})\to\mathsf{int}\to\mathsf{int} \overset{+\ell}{\Longrightarrow} \star) \texttt{ in} \\ \lceil \mathit{twice}^\star\ \mathit{inc}^\star\ 0 \rceil\end{array} \tag{1}$$

It evaluates to $\lceil 2 \rceil : \star$. If in (1) we replace

$$\lceil \mathit{twice}^\star\ \mathit{inc}^\star\ 0 \rceil \qquad \text{by} \qquad \lceil \mathit{twice}^\star\ 0\ \mathit{inc}^\star \rceil \tag{2}$$

it now evaluates to $\texttt{blame}\ -\ell$ because the number 0 is cast to function type $\mathsf{int}\to\mathsf{int}$. Blaming $-\ell$ indicates fault lies with the *context containing* the cast. The positive and negative blame labels, $+\ell$ and $-\ell$, correspond to p and $\bar{p}$, respectively, in prior blame calculi [Ahmed et al. 2011]. The above example demonstrates a benefit of casts and blame tracking: the fault lies at the boundary between the typed and untyped code, providing encapsulation for the typed code.

Conversely, here is the program mostly typed, with $\mathit{twice}^\star$ the only untyped component, cast from $\star$ to its type.

$$\begin{array}{l}\texttt{let } \mathit{inc} = (\lambda(x{:}\mathsf{int}).\, x+1) \texttt{ in} \\ \texttt{let } \mathit{twice}^\star = \lceil \lambda(f).\, \lambda(x).\, f\ (f\ x) \rceil \texttt{ in} \\ \texttt{let } \mathit{twice} = (\mathit{twice}^\star : \star \overset{+\ell}{\Longrightarrow} (\mathsf{int}\to\mathsf{int})\to\mathsf{int}\to\mathsf{int}) \texttt{ in} \\ \mathit{twice}\ \mathit{inc}\ 0\end{array} \tag{3}$$

It evaluates to $2 : \mathsf{int}$. If in (3) we replace

$$\lceil \lambda(f).\, \lambda(x).\, f\ (f\ x) \rceil \qquad \text{by} \qquad \lceil \lambda(f).\, \lambda(x).\, x\ f \rceil \tag{4}$$

it now evaluates to $\texttt{blame}\ +\ell$ because the untyped code tries to use an number as a function. Blaming $+\ell$ indicates fault lies with the *term contained* in the cast.

Untyped λ-calculus is defined by embedding into blame calculus. For example, $\lceil \lambda(x).\, x+1 \rceil$ is equivalent to

$$(\lambda(x:\star).\, ((x : \star \overset{+m}{\Longrightarrow} \mathsf{int}) + 1) : \mathsf{int} \overset{+n}{\Longrightarrow} \star) : \star\to\star \overset{+o}{\Longrightarrow} \star$$

where m, n, o are fresh blame labels.

Casting an int to $\star$ and back to int acts as the identity.

$$2 : \mathsf{int} \overset{+\ell}{\Longrightarrow} \star \overset{+m}{\Longrightarrow} \mathsf{int} \longrightarrow 2$$

On the other hand, casting the integer to a function type raises blame.

$$2 : \mathsf{int} \overset{+\ell}{\Longrightarrow} \star \overset{+m}{\Longrightarrow} (\mathsf{int}\to\mathsf{int}) \longrightarrow \texttt{blame}\ +m$$

A cast that yields a function reduces to two casts, one contravariant on the domain and one covariant on the range.

$$(\lceil \lambda(x).\, x{+}1 \rceil : \star \overset{+\ell}{\Longrightarrow} \mathsf{int}\to\mathsf{int})\ 2 \longrightarrow^* (\lambda(x{:}\star).\, \lceil x{+}1 \rceil)\ (2 : \mathsf{int} \overset{-\ell}{\Longrightarrow} \star) : \star \overset{+\ell}{\Longrightarrow} \mathsf{int} \longrightarrow^* 3$$

The blame label is negated on the contravariant cast.

2.3 Prior Work on Polymorphic Blame

The polymorphic blame calculus of Ahmed et al. [2011] seeks to integrate the blame calculus into λF without sacrificing parametricity. To see why this is challenging, consider the following variation on the *swap* function. It casts a dynamically typed function to one which has the type of the *swap* function. The dynamically typed function has no restrictions enforced by its type; we choose it to be the identity function.

$$\mathit{badSwap} = \lceil \lambda(x).\, x \rceil : \star \overset{+\ell}{\Longrightarrow} (\forall X.\forall Y.X \times Y \to Y \times X)$$

First, note that the polymorphic blame calculus does not reject this *badSwap* statically because it allows casts from any type to $\star$ and from $\star$ to any type, which is needed to accommodate interactions with untyped code. So we turn to the dynamic behavior of *badSwap*. Given its type, parametricity tells us that if we instantiate it at int twice and apply it to $(1, 2)$, the result, if there is one, should be $(2, 1)$. Let's see what happens if we evaluate *badSwap* by simply combining the reduction rules of λF and the blame calculus, assuming that we can simply replace the type variables X and Y by their instantiations int.

$$\begin{aligned}
&\mathit{badSwap}\,[\mathsf{int}]\,[\mathsf{int}]\,(1,2)\\
\longrightarrow^* &\ ((\lambda(x:\star).\, x)\ ((1,2) : \mathsf{int}\times\mathsf{int} \overset{-\ell}{\Longrightarrow} \star)) : \star \overset{+\ell}{\Longrightarrow} \mathsf{int}\times\mathsf{int}\\
\longrightarrow^* &\ (1,2) : \mathsf{int}\times\mathsf{int} \overset{-\ell}{\Longrightarrow} \star \overset{+\ell}{\Longrightarrow} \mathsf{int}\times\mathsf{int}\\
\longrightarrow^* &\ ((1 : \mathsf{int} \overset{-\ell}{\Longrightarrow} \star \overset{+\ell}{\Longrightarrow} \mathsf{int}), (2 : \mathsf{int} \overset{-\ell}{\Longrightarrow} \star \overset{+\ell}{\Longrightarrow} \mathsf{int}))\\
\longrightarrow^* &\ (1,2)
\end{aligned}$$

This answer is wrong! It violates the Fundamental Property for the type of *badSwap*, namely $\forall X.\forall Y.X \times Y \to Y \times X$. We would prefer the program to halt with an error, blaming the cast with label $+\ell$.

The problem is that in λF, type application is accomplished by substitution, with the reduction rule

$$(\Lambda X.e)\ [B] \longrightarrow e[B/X]$$

In the *badSwap* example, the distinction between the type variables X and Y was erased when they were both replaced by int. Ahmed et al. [2011] solve this problem by delaying the substitution, using a ν binder and a conversion operator that replaces a ν-bound type variable with its binding (called a *static cast* in that paper).

$$(\Lambda X.v)\ [B] \longrightarrow \nu X{:=}B.\, (v : A \overset{+X}{\Longrightarrow} A[X{:=}B]) \qquad \text{if } \vdash \Lambda X.v : \forall X.A$$

With that change, *badSwap* halts with an error when it fails to cast from X to $\star$ to Y, because $X \neq Y$.

$$\mathit{badSwap}[\mathsf{int}]\,[\mathsf{int}]\,(1,2)$$

$$\longrightarrow^* \nu X{:=}\mathsf{int}.\,\nu Y{:=}\mathsf{int}.\,\begin{pmatrix} 1 : \mathsf{int} \overset{-X}{\Longrightarrow} X \overset{-\ell}{\Longrightarrow} \star \overset{+\ell}{\Longrightarrow} Y, \\ 2 : \mathsf{int} \overset{-Y}{\Longrightarrow} Y \overset{-\ell}{\Longrightarrow} \star \overset{+\ell}{\Longrightarrow} X \end{pmatrix} \longrightarrow^* \quad \text{blame } +\ell$$

Ahmed et al. [2011] conjecture that this design ensures parametricity, but they did not formulate a logical relation for the polymorphic blame calculus nor prove the Fundamental Property [1].

[1] Ahmed et al. [2009] prove blame and subtyping theorems for a polymorphic blame calculus. Ahmed et al. [2011] present a proof of a stronger subtyping theorem based on the jack-of-all-trades property, but unfortunately that proof is flawed [Ahmed et al. 2014].

2.4 Righting the Topsy-Turvy System

Regardless of parametricity, the design of Ahmed et al. [2011] is 'topsy turvy' in that it evaluates things one might expect to be values, and leaves as values things one might expect to be evaluated. In particular, following Wright [1995] and Pitts [1998b], one might expect the bodies of type abstractions to be restricted to values. Ahmed et al. [2011] mention that this would be desirable, but it is impossible in their system due to their formulation of reductions for casts:

$$v : A \to B \overset{p}{\Longrightarrow} A' \to B' \longrightarrow \lambda(x{:}A').\, v\,(x : A' \overset{-p}{\Longrightarrow} A) : B \overset{p}{\Longrightarrow} B' \tag{5}$$

$$v : A \overset{p}{\Longrightarrow} \forall X.B \longrightarrow \Lambda X.\, (v : A \overset{p}{\Longrightarrow} B) \tag{6}$$

The right-hand side of the second rule is a type abstraction with a body that is an arbitrary cast, and hence not a value. One might take any type abstraction to be a value, regardless of whether its body is a value, but this would lead to a violation of parametricity. For instance, parametricity requires that there should be no values of type $\forall X.X$, but the term $\Lambda X.\,\texttt{blame}\,+\ell$ has that type! To avoid the problem, they evaluate underneath type abstractions. That in turn leads to a problem with ν binders, so instead of generating new names globally, they push ν binders inside of values:

$$\nu X{:=}A.\, \lambda(y{:}B).\, e \longrightarrow \lambda(y{:}B[X{:=}A]).\, \nu X{:=}A.\, e \tag{7}$$

$$\nu X{:=}A.\, \Lambda X.\, v \longrightarrow \Lambda X.\, \nu X{:=}A.\, v \tag{8}$$

Rule 7 causes ν binders to be retained in function values, where one might expect the ν binder to be evaluated immediately.

Here we propose a simple way to avoid these convolutions, taking inspiration from the alternate approach to function casts that can be made space-efficient [Siek and Wadler 2010]. In that approach, casts between function type are values and the following reduction rule handles the application of a cast-wrapped function.

$$(v : A \to B \overset{p}{\Longrightarrow} A' \to B')\, v' \longrightarrow v\,(v' : A' \overset{-p}{\Longrightarrow} A) : B \overset{p}{\Longrightarrow} B' \tag{9}$$

The same approach can be applied to polymorphic values. Declare casts of the form $(v : A \overset{p}{\Longrightarrow} \forall X.B)$ to be values and add a reduction rule for applying a cast-wrapped polymorphic value to a type. The following are the two reduction rules for applying polymorphic values.

$$\Sigma \triangleright (\Lambda X.\, v)\,[B] \longrightarrow \Sigma, \alpha{:=}B \triangleright v[\alpha/X] : A[\alpha/X] \overset{+\alpha}{\Longrightarrow} A[B/X] \tag{10}$$

$$\Sigma \triangleright (v : A \overset{p}{\Longrightarrow} \forall X.A')\,[B] \longrightarrow \Sigma, \alpha{:=}B \triangleright v : A \overset{p}{\Longrightarrow} A'[\alpha/X] \overset{+\alpha}{\Longrightarrow} A'[B/X] \tag{11}$$

In place of type abstraction on the right-hand side of reduction rule (6), here in reduction rule (11) we have application on the left-hand side. Reductions no longer insert casts inside of type abstractions so their bodies can be restricted to values, as desired. This rules out any need to evaluate under type abstractions. In particular, $\Lambda X.\,\texttt{blame}\,+\ell$ is no longer a valid term, since $\texttt{blame}\,+\ell$ is not a value.

With the removal of evaluation under type abstractions, we are free to immediately place generated names in a global store, forgoing the use of ν binders. Let Σ range over name stores whose entries take the form $\alpha{:=}B$ and write $\Sigma \triangleright e$ for a configuration that pairs a name store with an expression.

3 POLYMORPHIC BLAME CALCULUS

Now that we solved these design challenges, we can proceed with the formal presentation of the polymorphic blame calculus (λB). The syntax is defined in Figure 2. Let A and B range over types,

Conversion Labels	ϕ	::=	$+\alpha \mid -\alpha$
Blame Labels	p, q	::=	$+\ell \mid -\ell$
Base Types	ι	::=	$\mathsf{int} \mid \mathsf{bool}$
Types	A, B	::=	$\iota \mid A \times B \mid A \to B \mid \forall X.\, A \mid X \mid \alpha \mid \star$
Ground Types	G, H	::=	$\iota \mid \star \times \star \mid \star \to \star \mid \alpha$
Operations	$\circledast$	::=	$+ \mid - \mid * \mid \ldots$
Expressions	e	::=	$n \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid e \circledast e \mid x \mid$ $\lambda(x:A).\, e \mid e\ e \mid \Lambda X.\, v \mid e\ [B] \mid \langle e,e\rangle \mid \pi_1\, e \mid$ $\pi_2\, e \mid (e : A \overset{\phi}{\Longrightarrow} B) \mid (e : A \overset{p}{\Longrightarrow} B) \mid \mathsf{blame}\ p$
Values	v	::=	$n \mid \mathsf{true} \mid \mathsf{false} \mid \lambda(x:A).\, e \mid \Lambda X.\, v \mid \langle v,v\rangle \mid$ $(v : A \to B \overset{\phi}{\Longrightarrow} A' \to B') \mid (v : \forall X.\, A \overset{\phi}{\Longrightarrow} \forall X.\, B) \mid$ $(v : A \overset{-\alpha}{\Longrightarrow} \alpha) \mid (v : A \to B \overset{p}{\Longrightarrow} A' \to B') \mid$ $(v : A \overset{p}{\Longrightarrow} \forall X.\, B) \mid (v : G \overset{p}{\Longrightarrow} \star)$
Type-Name Stores	Σ	::=	$\cdot \mid \Sigma, \alpha{:=}A$
Type Environments	Δ	::=	$\cdot \mid \Delta, X$
Environments	Γ	::=	$\cdot \mid \Gamma, x : A$
Evaluation Contexts	E	::=	$[\cdot] \mid E \circledast e \mid v \circledast E \mid \mathsf{if}\ E\ \mathsf{then}\ e\ \mathsf{else}\ e \mid E\ e \mid v\ E \mid$ $E\ [A] \mid \langle E,e\rangle \mid \langle v,E\rangle \mid (E : A \overset{\phi}{\Longrightarrow} B) \mid (E : A \overset{p}{\Longrightarrow} B)$

Fig. 2. Syntax

which are either base types ι, pair types $A \times B$, function types $A \to B$, universal types $\forall X.B$, type variables X, type names α, or the dynamic type $\star$. Let ι include integers int and Booleans bool. We do not include existential types here but plan to investigate both the encoding into universal types and directly adding them to the language. The ground types are those types that can be directly injected into type $\star$, and include the base types, the function type $\star \to \star$, the pair type $\star \times \star$, and type names α. The other types can be cast to $\star$, but such casts factor through the ground types. Let e range over expressions, which are either integers n, the Booleans true and false, variables x, operator applications $e \circledast e$, function abstraction $\lambda(x{:}A).\, e$, function application $e_1\, e_2$, type abstraction $\Lambda X.\, v$, type application $e\ [A]$, pairs $\langle e,e\rangle$, projections $\pi_1\, e$ and $\pi_2\, e$, casts $(e : A \overset{p}{\Longrightarrow} B)$, conversions $(e : A \overset{\phi}{\Longrightarrow} B)$, or errors $\mathtt{blame}\, p$. The body of a type abstraction must be a value.

The *conversion* form $(e : A \overset{\phi}{\Longrightarrow} B)$ is used to make explicit the conversion between a type name and the type it is bound to (in Σ). Let ϕ range over *conversion labels* of the form $+\alpha$ or $-\alpha$.

Let v range over values, which include integers, Booleans, function abstractions, pairs of values, and type abstractions. A value whose type is a name is a conversion of the form $(v : A \overset{-\alpha}{\Longrightarrow} \alpha)$. In addition, conversions and casts to a function or a universal type are values, which reduce when applied to a value or type, respectively.

We write $B[A/X]$ (or $B[A/\alpha]$) for the capture-avoiding substitution of type A for X (or α) in type B. We abbreviate a sequence of conversions in the same way as casts and similarly for combinations of casts and conversions.

Static Semantics. The static semantics for types is defined in Figure 3. Let Δ range over type variable contexts, which are lists of X's. Let Σ range over name stores, which are lists of binding of type names to types $\alpha{:=}A$. Write $\vdash \Sigma$ to indicate that Σ is a well-formed name store and $\Sigma; \Delta \vdash \Gamma$ for well-formed type context. Write $\Sigma; \Delta \vdash A$ for well-formed types.

Type-Name Store Well-Formedness $\boxed{\vdash \Sigma}$

$$\frac{}{\vdash \cdot} \qquad \frac{\alpha \notin \Sigma \qquad \Sigma; \cdot \vdash A}{\vdash \Sigma, \alpha{:=}A}$$

Type Well-Formedness $\boxed{\Sigma; \Delta \vdash A}$ where $\vdash \Sigma$

$$\frac{\vdash \Sigma \qquad X \in \Delta}{\Sigma; \Delta \vdash X} \qquad \frac{\vdash \Sigma \qquad \alpha{:=}A \in \Sigma}{\Sigma; \Delta \vdash \alpha} \qquad \frac{\vdash \Sigma}{\Sigma; \Delta \vdash \iota} \qquad \frac{\vdash \Sigma}{\Sigma; \Delta \vdash \star} \qquad \frac{\Sigma; \Delta \vdash A \qquad \Sigma; \Delta \vdash B}{\Sigma; \Delta \vdash A \times B}$$

$$\frac{\Sigma; \Delta \vdash A \qquad \Sigma; \Delta \vdash B}{\Sigma; \Delta \vdash A \to B} \qquad \frac{\Sigma; \Delta, X \vdash A}{\Sigma; \Delta \vdash \forall X.\, A}$$

Convertibility $\boxed{\Sigma; \Delta \vdash A \prec^{\phi} B}$ where $\Sigma; \Delta \vdash A$, $\Sigma; \Delta \vdash B$, and $FTN(\phi) \in \Sigma$

$$\frac{\vdash \Sigma}{\Sigma; \Delta \vdash \iota \prec^{\phi} \iota} \qquad \frac{\Sigma; \Delta \vdash A \prec^{\phi} A' \qquad \Sigma; \Delta \vdash B \prec^{\phi} B'}{\Sigma; \Delta \vdash A \times B \prec^{\phi} A' \times B'} \qquad \frac{\Sigma; \Delta \vdash A' \prec^{-\phi} A \qquad \Sigma; \Delta \vdash B \prec^{\phi} B'}{\Sigma; \Delta \vdash A \to B \prec^{\phi} A' \to B'}$$

$$\frac{\Sigma; \Delta, X \vdash A \prec^{\phi} B}{\Sigma; \Delta \vdash \forall X.\, A \prec^{\phi} \forall X.\, B} \qquad \frac{\vdash \Sigma \qquad \alpha{:=}A \in \Sigma}{\Sigma; \Delta \vdash \alpha \prec^{+\alpha} A} \qquad \frac{\vdash \Sigma \qquad \alpha{:=}A \in \Sigma}{\Sigma; \Delta \vdash A \prec^{-\alpha} \alpha}$$

$$\frac{\vdash \Sigma \qquad \alpha{:=}A \in \Sigma \qquad \alpha \notin \phi}{\Sigma; \Delta \vdash \alpha \prec^{\phi} \alpha} \qquad \frac{\vdash \Sigma \qquad X \in \Delta}{\Sigma; \Delta \vdash X \prec^{\phi} X} \qquad \frac{\vdash \Sigma}{\Sigma; \Delta \vdash \star \prec^{\phi} \star}$$

Label Negation

$$-(+\alpha) \stackrel{\text{def}}{=} -\alpha \qquad -(-\alpha) \stackrel{\text{def}}{=} +\alpha$$

Compatibility $\boxed{\Sigma; \Delta \vdash A \prec B}$ where $\Sigma; \Delta \vdash A$ and $\Sigma; \Delta \vdash B$

$$\frac{\vdash \Sigma}{\Sigma; \Delta \vdash \iota \prec \iota} \qquad \frac{\Sigma; \Delta \vdash A \prec A' \qquad \Sigma; \Delta \vdash B \prec B'}{\Sigma; \Delta \vdash A \times B \prec A' \times B'} \qquad \frac{\Sigma; \Delta \vdash A' \prec A \qquad \Sigma; \Delta \vdash B \prec B'}{\Sigma; \Delta \vdash A \to B \prec A' \to B'}$$

$$\frac{\Sigma; \Delta, X \vdash A \prec B \qquad X \notin A}{\Sigma; \Delta \vdash A \prec \forall X.\, B} \qquad \frac{\Sigma; \Delta \vdash A[\star/X] \prec B}{\Sigma; \Delta \vdash \forall X.\, A \prec B} \qquad \frac{\vdash \Sigma \qquad \alpha \in \Sigma}{\Sigma; \Delta \vdash \alpha \prec \alpha} \qquad \frac{\vdash \Sigma \qquad X \in \Delta}{\Sigma; \Delta \vdash X \prec X}$$

$$\frac{\Sigma; \Delta \vdash A}{\Sigma; \Delta \vdash A \prec \star} \qquad \frac{\Sigma; \Delta \vdash A}{\Sigma; \Delta \vdash \star \prec A}$$

Fig. 3. Type-Level Static Semantics

Write $\Sigma; \Delta \vdash A \prec^{\phi} B$ to indicate that in context $\Sigma; \Delta$ types A and B are *convertible* under ϕ. Conversions must be between convertible types, and the rules for convertibility ensure that reductions involving conversions preserve convertibility. That is, when a conversion $(e : A \stackrel{\phi}{\Longrightarrow} B)$ occurs on the right-hand side of a reduction rule, we can deduce that $\Sigma; \Delta \vdash A \prec^{\phi} B$ from the fact that the conversions of the left-hand side of the rule were well typed. If $\alpha{:=}A \in \Sigma$, judgments $\Sigma; \Delta \vdash B \prec^{+\alpha} B'$ and $\Sigma; \Delta \vdash B' \prec^{-\alpha} B$ hold iff $B' = B[A/\alpha]$. Further, $\Sigma; \Delta \vdash A \prec^{\phi} B$ iff $\Sigma; \Delta \vdash B \prec^{-\phi} A$. Write $\alpha \notin \phi$ if

Environment Well-Formedness $\boxed{\Sigma;\Delta\vdash\Gamma}$ where $\vdash\Sigma$

$$\frac{\vdash\Sigma}{\Sigma;\Delta\vdash\cdot}\qquad\frac{\Sigma;\Delta\vdash\Gamma\qquad\Sigma;\Delta\vdash A}{\Sigma;\Delta\vdash\Gamma,x:A}$$

Well-typed Expressions $\boxed{\Sigma;\Delta;\Gamma\vdash e:A}$ where $\Sigma;\Delta\vdash\Gamma$ and $\Sigma;\Delta\vdash A$

$$\frac{\Sigma;\Delta;\Gamma\vdash e:\mathsf{bool}\qquad\Sigma;\Delta;\Gamma\vdash e_1:A\qquad\Sigma;\Delta;\Gamma\vdash e_2:A}{\Sigma;\Delta;\Gamma\vdash\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2:A}\qquad\frac{\Sigma;\Delta\vdash\Gamma}{\Sigma;\Delta;\Gamma\vdash\mathsf{true}:\mathsf{bool}}\qquad\frac{\Sigma;\Delta\vdash\Gamma}{\Sigma;\Delta;\Gamma\vdash\mathsf{false}:\mathsf{bool}}$$

$$\frac{\Sigma;\Delta\vdash\Gamma}{\Sigma;\Delta;\Gamma\vdash n:\mathsf{int}}\qquad\frac{\Sigma;\Delta;\Gamma\vdash e:\mathsf{int}\qquad\Sigma;\Delta;\Gamma\vdash e':\mathsf{int}}{\Sigma;\Delta;\Gamma\vdash e\circledast e':\mathsf{int}}\qquad\frac{\Sigma;\Delta;\Gamma\vdash e_1:A\qquad\Sigma;\Delta;\Gamma\vdash e_2:B}{\Sigma;\Delta;\Gamma\vdash\langle e_1,e_2\rangle:A\times B}$$

$$\frac{\Sigma;\Delta;\Gamma\vdash e:A\times B}{\Sigma;\Delta;\Gamma\vdash\pi_1\,e:A}\qquad\frac{\Sigma;\Delta;\Gamma\vdash e:A\times B}{\Sigma;\Delta;\Gamma\vdash\pi_2\,e:B}$$

$$\frac{\Sigma;\Delta\vdash\Gamma\qquad\Gamma(x)=A}{\Sigma;\Delta;\Gamma\vdash x:A}\qquad\frac{\Sigma;\Delta;\Gamma,x:A\vdash e:B}{\Sigma;\Delta;\Gamma\vdash\lambda(x:A).\,e:A\to B}\qquad\frac{\Sigma;\Delta;\Gamma\vdash e:B\to A\qquad\Sigma;\Delta;\Gamma\vdash e':B}{\Sigma;\Delta;\Gamma\vdash e\ e':A}$$

$$\frac{\Sigma;\Delta,X;\Gamma\vdash v:A\qquad\Sigma;\Delta\vdash\Gamma}{\Sigma;\Delta;\Gamma\vdash\Lambda X.v:\forall X.A}\qquad\frac{\Sigma;\Delta;\Gamma\vdash e:\forall X.A\qquad\Sigma;\Delta\vdash B}{\Sigma;\Delta;\Gamma\vdash e\ [B]:A[B/X]}\qquad\frac{\Sigma;\Delta;\Gamma\vdash e:A\qquad\Sigma;\Delta\vdash A\prec^{\phi}B}{\Sigma;\Delta;\Gamma\vdash(e:A\overset{\phi}{\Longrightarrow}B):B}$$

$$\frac{\Sigma;\Delta;\Gamma\vdash e:A\qquad\Sigma;\Delta\vdash A\prec B}{\Sigma;\Delta;\Gamma\vdash(e:A\overset{p}{\Longrightarrow}B):B}\qquad\frac{\Sigma;\Delta\vdash\Gamma\qquad\Sigma;\Delta\vdash A}{\Sigma;\Delta;\Gamma\vdash\mathsf{blame}\,p\ :A}$$

Fig. 4. Expression-Level Static Semantics

ϕ is not $+\alpha$ or $-\alpha$. Convertability is almost reflexive: in that $\Sigma;\Delta\vdash A\prec^{\phi}A$ if $\alpha\notin\phi$ for every free name α in A. Figure 3 defines negation $-\phi$ to flip the sign of a conversion label.

Write $\Sigma;\Delta\vdash A\prec B$ to indicate that in context $\Sigma;\Delta$ types A and B are *compatible*. The intuition is that two types are compatible if it is possible for a cast from one to the other to succeed. The type $\star$ is compatible with any type (on either side). A universal type $\forall X.A$ is compatible with a type B if instantiating it with $\star$ yields a compatible type, that is, if $A[\star/X]$ is compatible with B. On the other hand, a type A is compatible with a universal type $\forall X.B$ if A is compatible with B while holding X abstract, that is, if $\Sigma;\Delta,X\vdash A\prec B$. The compatibility relation is reflexive but not symmetric because of the treatment of universal types. The rules for compatibility ensure that reductions involving casts preserve compatibility.

The static semantics for expressions is defined in Figure 4. Let Γ range over type contexts, which are lists of hypotheses of the form $x{:}A$. Write $\Sigma;\Delta;\Gamma\vdash e:A$ to indicate that in context $\Sigma;\Delta;\Gamma$ expression e has type A. Typing for constants, operators, function abstraction, function application, type abstraction, and type application is standard. A conversion $(e:A\overset{\phi}{\Longrightarrow}B)$ has type B if expression e has type A and $\Sigma;\Delta\vdash A\prec^{\phi}B$. Similarly, a cast $(e:A\overset{p}{\Longrightarrow}B)$ has type B if expression e has type A and $\Sigma;\Delta\vdash A\prec B$.

Dynamic Semantics. The dynamic semantics of λB is defined in Figure 5. Write $e\longrightarrow e'$ for reduction of expressions. Reduction for operators is standard: each operator application $n\circledast n'$

Reduction of Expressions $\boxed{e \longrightarrow e'}$

$$
\begin{array}{rcl}
n \circledast n' & \longrightarrow & [\![\circledast]\!]\,(n,n') \\
\text{if true then } e_1 \text{ else } e_2 & \longrightarrow & e_1 \\
\text{if false then } e_1 \text{ else } e_2 & \longrightarrow & e_2 \\
(\lambda(x:A).\,e)\ v & \longrightarrow & e[v/x] \\
\pi_1\,\langle v_1,v_2\rangle & \longrightarrow & v_1 \\
\pi_2\,\langle v_1,v_2\rangle & \longrightarrow & v_2 \\
(v : \iota \overset{\phi}{\Longrightarrow} \iota) & \longrightarrow & v \\
(\langle v_1,v_2\rangle : A\times B \overset{\phi}{\Longrightarrow} A'\times B') & \longrightarrow & \langle (v_1 : A \overset{\phi}{\Longrightarrow} B),(v_2 : A' \overset{\phi}{\Longrightarrow} B')\rangle \\
(v : A\to B \overset{\phi}{\Longrightarrow} A'\to B')\ v' & \longrightarrow & (v\ (v' : A' \overset{-\phi}{\Longrightarrow} A) : B \overset{\phi}{\Longrightarrow} B') \\
(v : \alpha \overset{\phi}{\Longrightarrow} \alpha) & \longrightarrow & v \quad \text{if } \alpha \notin \phi \\
((v : A \overset{-\alpha}{\Longrightarrow} \alpha) : \alpha \overset{+\alpha}{\Longrightarrow} A) & \longrightarrow & v \\
(v : \star \overset{\phi}{\Longrightarrow} \star) & \longrightarrow & v \\
(v : \iota \overset{p}{\Longrightarrow} \iota) & \longrightarrow & v \\
(\langle v_1,v_2\rangle : A\times B \overset{p}{\Longrightarrow} A'\times B') & \longrightarrow & \langle (v_1 : A \overset{p}{\Longrightarrow} B),(v_2 : A' \overset{p}{\Longrightarrow} B')\rangle \\
(v : A\to B \overset{p}{\Longrightarrow} A'\to B')\ v' & \longrightarrow & (v\ (v' : A' \overset{-p}{\Longrightarrow} A) : B \overset{p}{\Longrightarrow} B') \\
(v : \forall X.\,A \overset{p}{\Longrightarrow} B) & \longrightarrow & (v\ [\star] : A[\star/X] \overset{p}{\Longrightarrow} B) \quad \text{if } B \neq \forall Y.\,B' \text{ for any } Y, B' \\
(v : \alpha \overset{p}{\Longrightarrow} \alpha) & \longrightarrow & v \\
(v : \star \overset{p}{\Longrightarrow} \star) & \longrightarrow & v \\
((v : G \overset{p}{\Longrightarrow} \star) : \star \overset{q}{\Longrightarrow} G) & \longrightarrow & v \\
((v : G \overset{p}{\Longrightarrow} \star) : \star \overset{q}{\Longrightarrow} H) & \longrightarrow & \text{blame } q \quad \text{if } G \neq H \\
(v : A \overset{p}{\Longrightarrow} \star) & \longrightarrow & ((v : A \overset{p}{\Longrightarrow} G) : G \overset{p}{\Longrightarrow} \star) \quad \text{if } A \sim G, A \neq G, A \neq \star \\
(v : \star \overset{p}{\Longrightarrow} A) & \longrightarrow & ((v : \star \overset{p}{\Longrightarrow} G) : G \overset{p}{\Longrightarrow} A) \quad \text{if } A \sim G, A \neq G, A \neq \star
\end{array}
$$

Reduction of Configurations $\boxed{\Sigma \triangleright e \longmapsto \Sigma' \triangleright e'}$

$$\frac{e \longrightarrow e'}{\Sigma \triangleright E[e] \longmapsto \Sigma \triangleright E[e']} \qquad \frac{\Sigma \triangleright e \longmapsto \Sigma' \triangleright e'}{\Sigma \triangleright E[e] \longmapsto \Sigma' \triangleright E[e']} \qquad \frac{}{\Sigma \triangleright E[\text{blame } p\,] \longmapsto \Sigma \triangleright \text{blame } p}$$

$$\frac{\Sigma;(\cdot,X);\ \cdot \vdash v : A \qquad \alpha \notin \text{dom}(\Sigma)}{\Sigma \triangleright (\Lambda X.\,v)\ [B] \longmapsto \Sigma, \alpha{:=}B \triangleright (v[\alpha/X] : A[\alpha/X] \overset{+\alpha}{\Longrightarrow} A[B/X])}$$

$$\frac{\alpha \notin \text{dom}(\Sigma)}{\Sigma \triangleright (v : A \overset{p}{\Longrightarrow} \forall X.\,A')\ [B] \longmapsto \Sigma, \alpha{:=}B \triangleright ((v : A \overset{p}{\Longrightarrow} A'[\alpha/X]) : A'[\alpha/X] \overset{+\alpha}{\Longrightarrow} A'[B/X])}$$

$$\frac{\alpha \notin \text{dom}(\Sigma)}{\Sigma \triangleright (v : \forall X.\,A \overset{\phi}{\Longrightarrow} \forall X.\,A')\ [B] \longmapsto \Sigma, \alpha{:=}B \triangleright ((v\ [\alpha] : A[\alpha/X] \overset{\phi}{\Longrightarrow} A'[\alpha/X]) : A'[\alpha/X] \overset{+\alpha}{\Longrightarrow} A'[B/X])}$$

Fig. 5. Dynamic Semantics

is specified by a total meaning function $[\![\circledast]\!]\,(n,n')$ that preserves types. Reduction for function application is standard. Reduction for type application is non-standard and discussed below in the reduction of configurations.

Conversions are reduced as follows. Conversion from a base type to itself is the identity. Conversion between two function types, when applied to a value, reduces to a contravariant conversion on the domain (negating the label) and a covariant conversion on the range. Conversion between two universal types are discussed below. A type name reduces to itself only if the conversion label refers to a different name. Two mirror image conversions, from A to α and back, reduce to the identity.

Many of the reductions for casts are similar to those for conversion, but some are different. A cast from universal type to a non-universal type reduces to an instantiation of the polymorphic value at type $\star$. A cast from ground type G to $\star$ back to G is the identity whereas a cast from G to $\star$ to a different ground type H raises blame. A cast from a non-ground type to or from $\star$ factors through its unique ground type.

Write $\Sigma \triangleright e \longmapsto \Sigma' \triangleright e'$ for reduction of configurations. In addition to handling reduction under an evaluation context, the reduction of configurations also handles the reductions that involve type application, which use the name store Σ. The application of a type abstraction generates a fresh name α and binds it to the instantiating type B in the store. Then α is substituted for the abstraction's variable X in the body. Finally, a conversion is wrapped around the body to mediate between its type $A[\alpha/X]$ and the type that the context expects it to have, which is $A[B/X]$ [2]. The application of a cast whose target is a universal type $\forall X.A'$ also generates a fresh name α and binds it to the instantiating type B. The α is substituted for X in A' and again, a conversion is inserted to mediate between $A'[\alpha/X]$ and $A'[B/X]$. Finally, the application of a conversion between $\forall X.A$ and $\forall X.A'$ involves generating α and binding it to B, then instantiating v at α, and substituting α for X in the conversion. As in the other cases, a conversion is inserted to mediate between $A'[\alpha/X]$ and $A'[B/X]$.

In general, if $\longrightarrow$ is a reduction relation, write $\longrightarrow^*$ for its reflexive and transitive closure. We write $\Sigma \triangleright e\Downarrow$ when a configuration terminates with a value—that is, as shorthand for $\exists\Sigma', v.\ \Sigma \triangleright e \longmapsto^* \Sigma' \triangleright v$

3.1 Contextual Equivalence

Two open expressions e_1 and e_2 have the same behavior, formally, when they are *contextually equivalent* [Morris 1968], that is, when substituting one for the other in the context of a larger program does not change the result of the program. As is common, we define contextual equivalence, below, in terms of contextual approximation. We write $\Sigma;\Delta;\Gamma \vdash e_1 \preceq^{ctx} e_2 : A$ to say that e_2 mimics the behavior of e_1 at type A in the context of Σ, Δ, and Γ. Then we write $\Sigma;\Delta;\Gamma \vdash e_1 \approx^{ctx} e_2 : A$ to say e_1 and e_2 are contextually equivalent, that is, they approximate each other. The definition of contextual equivalence requires the context C to be well-typed, written $\vdash C : (\Sigma;\Delta;\Gamma \vdash A) \rightsquigarrow (\Sigma';\Delta';\Gamma' \vdash A')$, which is defined in the accompanying technical report [Ahmed et al. 2017]. The well-known problem with contextual equivalence is that it is difficult to reason about all possible contexts. The advantage of logical relations is that they provide a means for proving the contextual equivalence of two expressions that only requires reasoning about the two expressions.

[2] The reduction rule for type abstraction obtains A, the type of the body, by referring to the type system, but an implementation would not do that. Instead, we recommend recording the type A during compilation and attaching it to the type abstraction so that it is available in constant time.

Definition 3.1. Contextual Approximation and Equivalence

$$\begin{aligned}
\Sigma;\Delta;\Gamma \vdash e_1 \preceq^{ctx} e_2 : A \quad &\stackrel{\text{def}}{=} \quad \Sigma;\Delta;\Gamma \vdash e_1 : A \ \wedge\ \Sigma;\Delta;\Gamma \vdash e_2 : A \ \wedge \\
&\quad \forall C, \Sigma', B.\ \vdash C : (\Sigma;\Delta;\Gamma \vdash A) \rightsquigarrow (\Sigma';\cdot;\cdot \vdash B) \implies \\
&\quad\quad (\Sigma' \triangleright C[e_1]\Downarrow \implies \Sigma' \triangleright C[e_2]\Downarrow) \wedge \\
&\quad\quad (\exists \Sigma_1.\ \Sigma' \triangleright C[e_1] \longmapsto^* \Sigma_1 \triangleright \text{blame } p \implies \\
&\quad\quad\quad \exists \Sigma_2.\ \Sigma' \triangleright C[e_2] \longmapsto^* \Sigma_2 \triangleright \text{blame } p\ ) \\
\Sigma;\Delta;\Gamma \vdash e_1 \approx^{ctx} e_2 : A \quad &\stackrel{\text{def}}{=} \quad \Sigma;\Delta;\Gamma \vdash e_1 \preceq^{ctx} e_2 : A \ \wedge\ \Sigma;\Delta;\Gamma \vdash e_2 \preceq^{ctx} e_1 : A
\end{aligned}$$

4 LOGICAL RELATION AND PARAMETRICITY

In this section, we present our logical relation for λB and prove the Fundamental Property of the logical relation, which for this language can also be referred to as the Parametricity Theorem. We show that our logical relation is sound with respect to contextual equivalence—that is, if two programs are logically related then they are contextually equivalent—which justifies the use of the logical relation for proving contextual equivalence of programs.

4.1 Main Ideas of the Logical Relation

To establish parametricity for λB, we construct a Kripke logical relation which is a logical relation indexed by *possible worlds*. Kripke logical relations are needed when reasoning about properties that depend on certain conditions regarding the state of the computation. The basic idea, then, is to track prevailing conditions in the current world. Thus, our logical relation for closed terms $\mathcal{E}\,[\![A]\!]$ relates two expressions e_1 and e_2 in a world W. In contrast, the logical relation for λF (Figure 1) did not require worlds because λF is a pure language. The world W keeps track of three ingredients which we now discuss in turn.

First, a world W contains a natural number j that, intuitively, represents the number of steps left in the computation. In essence, we need *step-indexed* logical relations because λB contains the dynamic type $\star$ which, semantically, behaves like the following type D encoded using recursive types and tagged sums:

$$\mathsf{D} = \mu X.\ \iota \ +\ (X \times X) \ +\ (X \to X) \ +\ (\alpha_1 \ +\ \ldots \ +\ \alpha_n)$$

where $\alpha_1, \ldots, \alpha_n$ are the set of type names that have been generated thus far. So unlike λF, the logical relation cannot be defined simply by induction on the indexing type. Instead we make use of induction on the step index to ensure that the logical relation is well-founded. This is by now a standard technique used when defining logical relations for untyped languages or languages with dynamic type [Acar et al. 2008; Matthews and Ahmed 2008].

Second, a world W keeps track of the (current) type-name stores Σ_1 and Σ_2 under which the expressions e_1 and e_2, respectively, should be type checked and evaluated.

Finally, for any *related* type names α in the two programs, W keeps track of the relational interpretation R that should be used to relate values of type α. Intuitively, what makes these type names "related" is that they were generated (earlier in the computation) when both programs performed some type application that we wish to relate. The placement of a map from type names to relational interpretations in worlds is a significant departure from the logical relation for λF (Figure 1), where instead the ρ maps type variables to their relational interpretation. This change is the result of moving from static enforcement of parametricity to dynamic enforcement with RTG.[3]

[3]Technically, due to the nondeterministic nature of type-name generation, those two type applications could have generated different type names α_1 and α_2, with some type variable X replaced by α_1 in one program and with α_2 in the other. But since we have an infinite supply of type names, we can assume without loss of generality that we can always generate the same fresh name for both programs when using the logical relation to reason about equivalence. This assumption reduces

$$
\begin{array}{lcl}
\mathrm{Atom}_n\,[A_1,A_2] & = & \{(W,e_1,e_2) \mid W.j < n \ \wedge\ W \in \mathrm{World}_n \ \wedge \\
& & \quad W.\Sigma_1;\cdot;\cdot \vdash e_1 : A_1 \ \wedge\ W.\Sigma_2;\cdot;\cdot \vdash e_2 : A_2\} \\
\mathrm{Atom}^{\mathrm{val}}_n\,[A_1,A_2] & = & \{(W,v_1,v_2) \in \mathrm{Atom}_n\,[A_1,A_2]\,\} \\
\mathrm{Rel}_n\,[A_1,A_2] & = & \{R \subseteq \mathrm{Atom}^{\mathrm{val}}_n\,[A_1,A_2] \mid \forall (W,v_1,v_2) \in R.\ \forall W' \sqsupseteq W.\ (W',v_1,v_2) \in R\} \\
\mathrm{World}_n & = & \{(j,\Sigma_1,\Sigma_2,\kappa) \in \mathrm{Nat} \times \mathrm{TNStore} \times \mathrm{TNStore} \times (\mathrm{TName} \xrightarrow{\mathrm{fin}} \mathrm{Rel}_j) \mid \\
& & \quad j < n \ \wedge\ \vdash \Sigma_1 \ \wedge\ \vdash \Sigma_2 \ \wedge\ \forall \alpha \in \mathrm{dom}(\kappa).\ \kappa(\alpha) \in \mathrm{Rel}_j\,[\Sigma_1(\alpha),\Sigma_2(\alpha)]\,\} \\
\mathrm{Atom}\,[A]\,\rho & = & \bigcup\limits_{n \geq 0} \{(W,e_1,e_2) \in \mathrm{Atom}_n\,[\rho(A),\rho(A)]\,\} \\
\mathrm{World} & = & \bigcup\limits_{n \geq 0} \mathrm{World}_n
\end{array}
$$

$$
\begin{array}{lcl}
\lfloor R \rfloor_n & = & \{(W,e_1,e_2) \in R \mid W.j < n\} \\
\lfloor \kappa \rfloor_n & = & \{\alpha \mapsto \lfloor R \rfloor_n \mid \kappa(\alpha) = R\} \\
\\
W' \sqsupseteq W & \stackrel{\mathrm{def}}{=} & W'.j \leq W.j \ \wedge\ W'.\Sigma_1 \supseteq W.\Sigma_1 \ \wedge\ W'.\Sigma_2 \supseteq W.\Sigma_2 \ \wedge \\
& & \quad W'.\kappa \ \sqsupseteq\ \lfloor W.\kappa \rfloor_{W'.j} \ \wedge\ W, W' \in \mathrm{World} \\
\kappa' \sqsupseteq \kappa & \stackrel{\mathrm{def}}{=} & \forall \alpha \in \mathrm{dom}(\kappa).\ \kappa'(\alpha) = \kappa(\alpha) \\
W' \sqsupseteq_n W & \stackrel{\mathrm{def}}{=} & W'.j + n = W.j \ \wedge\ W' \sqsupseteq W \\
\\
\blacktriangleright R & = & \{(W,e_1,e_2) \mid W.j > 0 \implies (\blacktriangleright W,e_1,e_2) \in R\} \\
\blacktriangleright(j+1,\Sigma_1,\Sigma_2,\kappa) & \stackrel{\mathrm{def}}{=} & (j,\Sigma_1,\Sigma_2,\lfloor \kappa \rfloor_j) \\
\\
W \boxplus (\alpha,B_1,B_2,R) & \stackrel{\mathrm{def}}{=} & (W.j, W.\Sigma_1, \alpha{:=}B_1, W.\Sigma_2, \alpha{:=}B_2, W.\kappa\,[\alpha \mapsto R])
\end{array}
$$

Fig. 6. Logical Relation: Auxiliary Definitions

An essential part of Kripke logical relations is a world extension relation $W' \sqsupseteq W$ that specifies constraints on how the current world W may evolve into a future world W'—intuitively, capturing how the properties of the state evolve from W to W' as the computation progresses. In our logical relation, we allow the step index to *decrease over time* as steps are used up by the computation, while allowing type-name stores and the finite map from type names to relational interpretations to *grow over time* as new type names are generated (ensuring that we don't forget information about existing type names). Intuitively, the semantic treatment of dynamically generated type names is similar to that of dynamically allocated mutable references, which is why the underlying structure of our logical relation resembles that of step-indexed Kripke logical relations for ML-style mutable references [Ahmed et al. 2010, 2003, 2009; Ahmed 2004].

4.2 Logical Relation, Formally Defined

The logical relation for λB is defined in Figures 6 and 7.

Preliminaries. We start by specifying the semantic objects used in the construction of the logical relation (Figure 6, top). We explain the basic properties of these semantic objects and then discuss details related to step indexing.

clutter in our logical relation, as otherwise we would have to keep track of a bijection between a subset of the existing type names of both programs as Neis et al. [2011] and Ahmed et al. [2011] do.

We construct relational interpretations of types as sets of *atoms* of the form (W, e_1, e_2) where worlds describe the assumptions under which the pair of expressions are related. $\mathrm{Atom}_n\,[A_1, A_2]$ requires that e_1 and e_2 have the types A_1 and A_2 under the type-name stores Σ_1 and Σ_2 from the world W, and that they have no free type or term variables.

Worlds W are 4-tuples of the form $(j, \Sigma_1, \Sigma_2, \kappa)$. (We use dot notation $W.j$, $W.\Sigma_1$, $W.\Sigma_2$, $W.\kappa$ to project out the relevant components of world W.) Here j is a natural number representing the step index, Σ_1 and Σ_2 are the type-name stores under which the terms being related are type checked and evaluated, and κ is a finite map from type names α to *admissible* relations R that relate values of type $\Sigma_1(\alpha)$ and $\Sigma_2(\alpha)$.[4]

$\mathrm{Rel}_n\,[A_1, A_2]$ specifies the set of admissible relations as relations R that satisfy *monotonicity* (or closure) under world extension: if R relates v_1 and v_2 in a world W, then R must relate v_1 and v_2 in any future world of W. Monotonicity ensures that when we extend the world, associating new type names with types and relations, we do not lose or modify information associated with old type names.

Since relations contain worlds and worlds contain relations (in the codomain of κ), we cannot naïvely construct a set-theoretic model based on the above intentions. As explained by Ahmed [2004] and Ahmed et al. [2009], who present step-indexed logical relations for ML-style mutable references, a naïve construction would have an inconsistent cardinality. To eliminate the inconsistency, we stratify both worlds and relations with a step index. We require that an n-level world $W \in \mathrm{World}_n$ contains only step indices $j < n$ and interpretations κ that map type names to j-level relations $R \in \mathrm{Rel}_j\,[A_1, A_2]$; and the latter may only contain j-level atoms which contain j-level worlds $W \in \mathrm{World}_j$. Our world structure is similar to that of Ahmed [2004] because dynamically generated type names are analogous to the dynamically allocated locations of mutable references.

Approximation, World Extension, and Later. The bottom half of Figure 6 presents definitions of n-approximation $\lfloor\cdot\rfloor_n$, world extension ($\sqsupseteq$) and the "later" operation ($\blacktriangleright$). For a set of atoms R, we define the n-approximation of the set (written $\lfloor R \rfloor_n$) as the subset of its elements whose step indices are strictly smaller than n. For any interpretation κ mapping type names to relational interpretations, we define n-approximation (written $\lfloor \kappa \rfloor_n$) by applying $\lfloor\cdot\rfloor_n$ to all relations in the codomain of κ.

A world W' extends W (written $W' \sqsupseteq W$) if W' has the same or fewer steps left as W; if the type-name stores in W' are supersets of the corresponding stores in W; and if $W'.\kappa$ remembers the relational interpretations associated with each $\alpha \in \mathrm{dom}(W.\kappa)$ up to approximation $W'.j$. Note that this definition allows W' to provide interpretations for a superset of the type names that W interprets. Finally, we write $W' \sqsupseteq_n W$ when W' extends W and we consume exactly n steps to get from W to W'.

Given a world W with a positive step index, we use $\blacktriangleright W$ (pronounced "later W") to lower the step index of the world—and of the κ in the world—by one. We lift this notion to relational interpretations R as follows: we say $\blacktriangleright R$ relates two expressions in world W if R relates the expressions in $\blacktriangleright W$ whenever W has a positive step index.

Relational Interpretation of Types. Figure 7 presents the definition of the logical relation intended to capture contextual approximation. The value relation $\mathcal{V}\,[\![A]\!]\,\rho$ relates *closed* values—i.e., values with no free type or term variables, though they may have free type names. The expression relation $\mathcal{E}\,[\![A]\!]\,\rho$ relates similarly *closed* expressions. The value and expression relations are parameterized by a type substitution ρ that maps free type variables X in A to type names α. Note that, unlike existing logical relations for polymorphic languages—such as the one shown in Section 2 or the

[4]The notation $\Sigma(\alpha)$ looks up the type associated with α in Σ. If Σ contains $\alpha := A$, then $\Sigma(\alpha) = A$.

$$
\begin{array}{lcl}
\mathcal{V}\,[\![\mathsf{int}]\!]\,\rho & = & \{(W,n,n) \in \mathrm{Atom}\,[\mathsf{int}]\,\rho\} \\
\mathcal{V}\,[\![\mathsf{bool}]\!]\,\rho & = & \{(W,b,b) \in \mathrm{Atom}\,[\mathsf{bool}]\,\rho\} \\
\mathcal{V}\,[\![A \times B]\!]\,\rho & = & \{(W,\langle v_1,v_2\rangle,\langle v_1',v_2'\rangle) \in \mathrm{Atom}\,[A\times B]\,\rho \mid \\
 & & \quad (W,v_1,v_2) \in \mathcal{V}\,[\![A]\!]\,\rho \ \wedge\ (W,v_1',v_2') \in \mathcal{V}\,[\![B]\!]\,\rho\} \\
\mathcal{V}\,[\![A \to B]\!]\,\rho & = & \{(W,v_{f1},v_{f2}) \in \mathrm{Atom}\,[A\to B]\,\rho \mid \\
 & & \quad \forall W' \sqsupseteq W.\ \forall v_1,v_2.\ (W',v_1,v_2) \in \mathcal{V}\,[\![A]\!]\,\rho \implies \\
 & & \quad\quad (W', v_{f1}\ v_1, v_{f2}\ v_2) \in \mathcal{E}\,[\![B]\!]\,\rho\} \\
\mathcal{V}\,[\![\forall X.\,A]\!]\,\rho & = & \{(W,v_{f1},v_{f2}) \in \mathrm{Atom}\,[\forall X.\,A]\,\rho \mid \\
 & & \quad \forall W' \sqsupseteq W.\ \forall B_1,B_2,R.\ \forall e_1,e_2.\ \forall \alpha. \\
 & & \quad\quad W'.\Sigma_1;\cdot \vdash B_1 \ \wedge\ W'.\Sigma_2;\cdot \vdash B_2 \ \wedge\ R \in \mathrm{Rel}_{W'.j}\,[B_1,B_2] \ \wedge \\
 & & \quad\quad W'.\Sigma_1 \triangleright v_{f1}\,[B_1] \longmapsto W'.\Sigma_1, \alpha{:=}B_1 \triangleright (e_1 : \rho(A)[\alpha/X] \overset{+\alpha}{\Longrightarrow} \rho(A)[B_1/X]) \ \wedge \\
 & & \quad\quad W'.\Sigma_2 \triangleright v_{f2}\,[B_2] \longmapsto W'.\Sigma_2, \alpha{:=}B_2 \triangleright (e_2 : \rho(A)[\alpha/X] \overset{+\alpha}{\Longrightarrow} \rho(A)[B_2/X]) \\
 & & \quad\quad\quad \implies (W' \boxplus (\alpha,B_1,B_2,R), e_1, e_2) \in \blacktriangleright\mathcal{E}\,[\![A]\!]\,\rho[X \mapsto \alpha]\} \\
\mathcal{V}\,[\![X]\!]\,\rho & = & \{(W,(v_1 : A_1 \overset{-\alpha}{\Longrightarrow} \alpha),(v_2 : A_2 \overset{-\alpha}{\Longrightarrow} \alpha)) \in \mathrm{Atom}\,[X]\,\rho \mid (W,v_1,v_2) \in \blacktriangleright W.\kappa\,(\alpha)\} \\
\mathcal{V}\,[\![\alpha]\!]\,\rho & = & \{(W,(v_1 : A_1 \overset{-\alpha}{\Longrightarrow} \alpha),(v_2 : A_2 \overset{-\alpha}{\Longrightarrow} \alpha)) \in \mathrm{Atom}\,[\alpha]\,\emptyset \mid (W,v_1,v_2) \in \blacktriangleright W.\kappa\,(\alpha)\} \\
\mathcal{V}\,[\![\star]\!]\,\rho & = & \{(W,(v : \iota \overset{p}{\Longrightarrow} \star),(v : \iota \overset{p}{\Longrightarrow} \star)) \in \mathrm{Atom}\,[\star]\,\emptyset\} \\
 & \cup & \{(W,(v_1 : \star\to\star \overset{p}{\Longrightarrow} \star),(v_2 : \star\to\star \overset{p}{\Longrightarrow} \star)) \in \mathrm{Atom}\,[\star]\,\emptyset \mid \\
 & & \quad (W,v_1,v_2) \in \blacktriangleright\mathcal{V}\,[\![\star\to\star]\!]\,\rho\} \\
 & \cup & \{(W,(v_1 : \alpha \overset{p}{\Longrightarrow} \star),(v_2 : \alpha \overset{p}{\Longrightarrow} \star)) \in \mathrm{Atom}\,[\star]\,\emptyset \mid \\
 & & \quad v_1 = (v_1' : A_1 \overset{-\alpha}{\Longrightarrow} \alpha) \ \wedge\ v_2 = (v_2' : A_2 \overset{-\alpha}{\Longrightarrow} \alpha) \ \wedge\ (W,v_1',v_2') \in \blacktriangleright W.\kappa\,(\alpha)\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{E}\,[\![A]\!]\,\rho & = & \{(W,e_1,e_2) \in \mathrm{Atom}\,[A]\,\rho \mid \forall j < W.j. \\
 & & \quad (\forall \Sigma_1, v_1.\ W.\Sigma_1 \triangleright e_1 \longrightarrow^j \Sigma_1 \triangleright v_1 \implies \\
 & & \quad\quad \exists W', \Sigma_2, v_2.\ W.\Sigma_2 \triangleright e_2 \longrightarrow^* \Sigma_2 \triangleright v_2 \ \wedge\ W' \sqsupseteq_j W \ \wedge \\
 & & \quad\quad W'.\Sigma_1 = \Sigma_1 \ \wedge\ W'.\Sigma_2 = \Sigma_2 \ \wedge\ (W',v_1,v_2) \in \mathcal{V}\,[\![A]\!]\,\rho) \ \wedge \\
 & & \quad (\forall \Sigma_1, p.\ W.\Sigma_1 \triangleright e_1 \longrightarrow^j \Sigma_1 \triangleright \mathrm{blame}\,p \implies \exists \Sigma_2.\ W.\Sigma_2 \triangleright e_2 \longrightarrow^* \Sigma_2 \triangleright \mathrm{blame}\,p)\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{S}\,[\![\cdot]\!] & = & \mathrm{World} \\
\mathcal{S}\,[\![\Sigma, \alpha{:=}A]\!] & = & \mathcal{S}\,[\![\Sigma]\!] \cap \{W \in \mathrm{World} \mid W.\Sigma_1(\alpha) = A \ \wedge\ W.\Sigma_2(\alpha) = A \ \wedge \\
 & & \qquad\qquad \vdash W.\Sigma_1 \ \wedge\ \vdash W.\Sigma_2 \ \wedge\ W.\kappa\,(\alpha) = \lfloor \mathcal{V}\,[\![A]\!]\,\emptyset \rfloor_{W.j}\} \\
\mathcal{D}\,[\![\cdot]\!] & = & \{(W,\emptyset) \mid W \in \mathrm{World}\} \\
\mathcal{D}\,[\![\Delta, X]\!] & = & \{(W,\rho[X \mapsto \alpha]) \mid (W,\rho) \in \mathcal{D}\,[\![\Delta]\!] \ \wedge\ \alpha \in \mathrm{dom}(W.\kappa)\} \\
\mathcal{G}\,[\![\cdot]\!]\,\rho & = & \{(W,\emptyset) \mid W \in \mathrm{World}\} \\
\mathcal{G}\,[\![\Gamma, x : A]\!]\,\rho & = & \{(W,\gamma\,[x \mapsto (v_1,v_2)]) \mid (W,\gamma) \in \mathcal{G}\,[\![\Gamma]\!]\,\rho \ \wedge\ (W,v_1,v_2) \in \mathcal{V}\,[\![A]\!]\,\rho\}
\end{array}
$$

$$
\begin{array}{lcl}
\Sigma;\Delta;\Gamma \vdash e_1 \preceq e_2 : A & \overset{\mathrm{def}}{=} & \Sigma;\Delta;\Gamma \vdash e_1 : A \ \wedge\ \Sigma;\Delta;\Gamma \vdash e_2 : A \ \wedge\ \forall W,\rho,\gamma. \\
 & & (W \in \mathcal{S}\,[\![\Sigma]\!] \ \wedge\ (W,\rho) \in \mathcal{D}\,[\![\Delta]\!] \ \wedge\ (W,\gamma) \in \mathcal{G}\,[\![\Gamma]\!]\,\rho) \\
 & & \quad \implies (W, \rho(\gamma_1(e_1)), \rho(\gamma_2(e_2))) \in \mathcal{E}\,[\![A]\!]\,\rho \\
\Sigma;\Delta;\Gamma \vdash e_1 \approx e_2 : A & \overset{\mathrm{def}}{=} & \Sigma;\Delta;\Gamma \vdash e_1 \preceq e_2 : A \wedge \Sigma;\Delta;\Gamma \vdash e_2 \preceq e_1 : A
\end{array}
$$

Fig. 7. Logical Relation

related work by Neis et al. [2011] and Ahmed et al. [2011]—ρ does *not* map type variables X to relational interpretations R. Instead ρ specifies a mapping from type variables X to type names α and we look up the relational interpretation for α (which is the same as the relational interpretation we wish to use for X) in the world W. We return to this point below when we discuss the value relation for universal types.

The relation $\mathcal{V}\,[\![A]\!]\,\rho$ specifies when closed values v_1 and v_2 are related in a world W. It requires that v_1 and v_2 have type $\rho(A)$ under the type-name stores $W.\Sigma_1$ and $W.\Sigma_2$, respectively. The definition of $\mathcal{V}\,[\![A]\!]\,\rho$ is straightforward for base types and pairs. Two values are related at a base type ι in any world W as long as they are equal. Two pairs are related at the type $A \times B$ in a world W if the first components of the pairs are related at type A in W and the second components are related at type B in W.

Functions v_{f1} and v_{f2} are related at $A \to B$ in world W if, in any future world W' that extends W, given two arguments v_1 and v_2 related at the argument type A in W', the functions applied to the arguments are related expressions at the result type B in world W'. The extension to a future world W' is essential for proving monotonicity of $\mathcal{V}\,[\![A \to B]\!]\,\rho$ under world extension. Note that in λB, the values v_{f1} and v_{f2} may each be a function abstraction or a conversion or cast from a value to a function type.

The definition of $\mathcal{V}\,[\![\forall X.\, A]\!]\,\rho$ is made more complex by several factors. Note that it must relate values v_{f1} and v_{f2}, which may each be a type abstraction or a conversion or cast from a value to a universal type. $\mathcal{V}\,[\![\forall X.\, A]\!]\,\rho$ says that v_{f1} and v_{f2} are related in world W if, in any future world W' that extends W, given two types B_1 and B_2 that are well formed under the type-name stores from W', and given an admissible relation $R \in \mathrm{Rel}\,[B_1, B_2]$ that is good for the remaining $W'.j$ steps, the type applications $v_{f1}\,[B_1]$ and $v_{f2}\,[B_2]$ must be related. The unusual aspect of the definition is how we specify relatedness of the aforementioned type applications.

Unlike Ahmed et al. [2011] and Neis et al. [2011], who require that the type applications $v_{f1}\,[B_1]$ and $v_{f2}\,[B_2]$ are related in $\mathcal{E}\,[\![A]\!]$ with an environment ρ that maps X to (B_1, B_2, R), in a world that extends W' to record (essentially) that X maps to the type name α and the triple (B_1, B_2, R), we avoid duplicating the information (B_1, B_2, R) in both ρ and the world. Instead we rely on the indirection provided by type names. That is, we let ρ specify simply that X maps to type name α and store the types and relational interpretation for α in the world (written $W' \boxplus (\alpha, B_1, B_2, R)$, which is defined at the bottom of Figure 6).

But having made the above design decision, we have a bit of a problem regarding how to specify relatedness of the type applications $v_{f1}\,[B_1]$ and $v_{f2}\,[B_2]$ in the definition of $\mathcal{V}\,[\![\forall X.\, A]\!]\,\rho$. We proceed by noting that the operational semantics guarantees that regardless of whether the values v_{fi} are type abstractions or conversions or casts to universal type, a type application $\Sigma \triangleright v_{fi}\,[B_i]$ steps to a configuration of the form $\Sigma, \alpha{:=}B_i \triangleright (e_i : \rho(A)[\alpha/X] \overset{+\alpha}{\Longrightarrow} \rho(A)[B_i/X])$. Thus, $\mathcal{V}\,[\![\forall X.\, A]\!]\,\rho$ requires that the subexpressions e_1 and e_2 be related in $\blacktriangleright\mathcal{E}\,[\![A]\!]\,\rho[X \mapsto \alpha]$ in world $W' \boxplus (\alpha, B_1, B_2, R)$. (At the end of Section 4.2, we explain why the dynamic semantics of type application in λB precludes a more standard logical-relation design where ρ maps type variables X to triples (B_1, B_2, R).)

We use $\blacktriangleright\mathcal{E}\,[\![A]\!]$ because we must ensure that we go down a step to avoid circularities due to impredicative polymorphism (as pointed out in prior work [Ahmed 2006, 2004]). Specifically, consider what happens if B_1 and B_2 happen to be the type $\rho(\forall X.\, A)$—i.e., types that are not strictly smaller than the type whose relational interpretation we wish to define. Then, to ensure that our definition of $\mathcal{V}\,[\![\forall X.\, A]\!]\,\rho$ at world W is well founded, when we relate e_1 and e_2 we must use only the $(W.j - 1)$-approximation of R (where $W.j \geq W'.j$). This is justified because type application consumes a reduction step.

The relation $\mathcal{V}\,[\![X]\!]\,\rho$ relates values of type $\rho(X) = \alpha$. Hence, the values must be of the form $(v_i : A_i \overset{-\alpha}{\Longrightarrow} \alpha)$ for $i \in \{1, 2\}$. These values are related if the underlying v_1 and v_2 are related by the relational interpretation for α (i.e., $W.\kappa(\alpha)$) for one fewer step.

The relation $\mathcal{V}\,[\![\alpha]\!]\,\rho$ also relates values of type α. Hence, the values must again be of the form $(v_i : A_i \overset{-\alpha}{\Longrightarrow} \alpha)$ for $i \in \{1, 2\}$. As above, these values are related if the underlying v_1 and v_2 are related by $W.\kappa(\alpha)$ for one fewer step.

The relation $\mathcal{V}\,[\![\star]\!]\,\rho$ relates values of the form $(v_1 : G \overset{p}{\Longrightarrow} \star)$ and $(v_2 : G \overset{p}{\Longrightarrow} \star)$ in world W, where both values have the same ground type G. There are three cases to consider. When G is a base type, v_1 and v_2 must be equal. When G is $\star \to \star$, v_1 and v_2 must be related at the type $\star \to \star$ in world W for one fewer step. Since our definition of the value relation at $\star$ relies on the value relation for the bigger type $\star \to \star$, it is imperative that we go down a step to ensure that the logical relation is well founded. Going down a step is justified since extracting the values v_i from $(v_i : \star \to \star \overset{p}{\Longrightarrow} \star)$ consumes a reduction step. Finally, when G is a type name α, the values v_1 and v_2 must be of the form $(v_i' : A_i \overset{-\alpha}{\Longrightarrow} \alpha)$ and v_1' and v_2' are related by the relational interpretation for α (i.e., $W.\kappa(\alpha)$) for one fewer step.

The expression relation $\mathcal{E}\,[\![A]\!]\,\rho$ is similar to prior step-indexed Kripke logical relations (in particular, Matthews and Ahmed [2008]). It says that two expressions e_1 and e_2 are related in world W if whenever e_1 evaluates to a value under $W.\Sigma_1$ in fewer than $W.j$ steps, then e_2 evaluates to a value under $W.\Sigma_2$ and the resulting values and stores are related in some future world W'; and whenever e_1 evaluates to blame p under $W.\Sigma_1$, again in fewer than $W.j$ steps, then e_2 evaluates to blame p under $W.\Sigma_2$.

The remaining definitions in Figure 7 specify the logical relation for open terms. First, $\mathcal{S}\,[\![\Sigma]\!]$ specifies worlds W that satisfy Σ. It says that the type-name stores in W should be well formed, and for every $\alpha := A \in \Sigma$, the type-name stores in W should map α to A and the interpretation $W.\kappa$ should map α to the $W.j$-approximation of $\mathcal{V}\,[\![A]\!]\,\emptyset$. Next, $\mathcal{D}\,[\![\Delta]\!]$ says that a type substitution ρ satisfies Δ in a world W if it maps all the type variables in Δ to type names α that are associated with a relational interpretation in $W.\kappa$. Finally, we let the metavariable γ range over relational value substitutions, i.e., finite maps from variables to pairs of values. $\mathcal{G}\,[\![\Gamma]\!]\,\rho$ says that γ satisfies Γ in world W if every variable in $x \in \mathrm{dom}(\Gamma)$ is mapped to pairs of values related in $\mathcal{V}\,[\![\Gamma(x)]\!]\,\rho$ in W.

The logical approximation relation on open terms $\Sigma\,;\Delta\,;\Gamma \vdash e_1 \preceq e_2 : A$ says that given a world W that satisfies Σ, a type substitution ρ that maps type variables in Δ to type names in W, and a relational value substitution γ that maps variables to pairs of values related in W, the terms $\rho(\gamma_1(e_1))$ and $\rho(\gamma_2(e_2))$ are related in $\mathcal{E}\,[\![A]\!]\,\rho$ at world W. The logical equivalence relation $\Sigma\,;\Delta\,;\Gamma \vdash e_1 \approx e_2 : A$ requires that e_1 logically approximates e_2 and vice versa.

Complications in the Interpretation of Universal Types. Consider defining the logical relation in the usual way with value interpretations parameterized by an environment ρ that maps type variables X to triples (B_1, B_2, R) that record the types B_1 and B_2 that X was instantiated with and the relational interpretation R that should be used to relate values at the type X. Then our interpretation of type variables and universal types would be as follows:

$$\begin{array}{lcl}
\mathcal{V}\,[\![X]\!]\,\rho & = & R \quad \text{where } \rho(X) = (B_1, B_2, R) \\
\mathcal{V}\,[\![\forall X\,.\,A]\!]\,\rho & = & \{(W, v_1, v_2) \in \mathrm{Atom}\,[\forall X\,.\,A]\,\rho \mid \forall W' \sqsupseteq W.\ \forall \alpha, B_1, B_2, R. \\
 & & \qquad W'.\Sigma_1;\cdot \vdash B_1 \ \wedge\ W'.\Sigma_2;\cdot \vdash B_2 \ \wedge\ R \in \mathrm{Rel}_{W'.j}\,[B_1, B_2] \implies \\
 & & \qquad (W' \boxplus (\alpha, B_1, B_2, R), v_1[B_1], v_2[B_2]) \in \mathcal{E}\,[\![A]\!]\,\rho[X \mapsto (B_1, B_2, R)]\}
\end{array}$$

With the above definitions, assuming a standard definition of the open term relation, we fail to prove the compatibility lemma for type abstraction. That lemma says: if $\Sigma\,;\Delta, X;\Gamma \vdash v_1 \preceq v_2 : A$ then $\Sigma\,;\Delta\,;\Gamma \vdash \Lambda X\,.\,v_1 \preceq \Lambda X\,.\,v_2 : \forall X\,.\,A$.

Next, we show the proof attempt for a simpler version of that lemma which says: if $\cdot;\cdot, X;\ \cdot \vdash v_1 \preceq v_2 : A$ then $\cdot;\ \cdot\ ;\ \cdot \vdash \Lambda X\,.\,v_1 \preceq \Lambda X\,.\,v_2 : \forall X\,.\,A$. Given W that satisfies the empty type-name

store, it suffices to show that $(W, \Lambda X.v_1, \Lambda X.v_2) \in \mathcal{V}\,[\![\forall X.\,A]\!]\,\emptyset$. Unrolling the definition given above, we assume appropriate W', α, B_1, B_2, and R and need to show

$$(W' \boxplus (\alpha, B_1, B_2, R), (\Lambda X.v_1)\,[B_1], (\Lambda X.v_2)\,[B_2]) \in \mathcal{E}\,[\![A]\!]\,\emptyset[X \mapsto (B_1, B_2, R)]$$

or equivalently (after taking a step),

$$\left(W' \boxplus (\alpha, B_1, B_2, R),\ \begin{array}{l}(v_1[\alpha/X] : A[\alpha/X] \stackrel{+\alpha}{\Longrightarrow} A[B_1/X]),\\ (v_2[\alpha/X] : A[\alpha/X] \stackrel{+\alpha}{\Longrightarrow} A[B_2/X])\end{array}\right) \in \blacktriangleright\mathcal{E}\,[\![A]\!]\,\rho[X \mapsto (B_1, B_2, R)]$$

But instantiating the premise $\cdot;\cdot, X;\ \cdot \vdash v_1 \preceq v_2 : A$ with $W' \boxplus (\alpha, B_1, B_2, R)$ and $\emptyset[X \mapsto (B_1, B_2, R)]$ gives us:

$$(W' \boxplus (\alpha, B_1, B_2, R), v_1[B_1/X], v_2[B_2/X]) \in \mathcal{E}\,[\![A]\!]\,\rho[X \mapsto (B_1, B_2, R)]$$

and we have no way of completing the proof.

The problem is that the logical relation for open terms substitutes the B_i for X while the reduction rule for type application substitutes a fresh type name α for X and then performs a conversion from $A[\alpha/X]$ to $A[B_i/X]$. To fix this mismatch, we would at a minimum need to make ρ a mapping from type variables X to (α, B_1, B_2, R) so we can keep track of the type name generated for each X. In addition, the logical relation for open terms $\Sigma;\Delta;\Gamma \vdash e_1 \preceq e_2 : A$ would have to be modified so that once we have picked a world W that satisfies Σ; a ρ that maps all X_i in Δ to $(\alpha_i, B_{i1}, B_{i2}, R_i)$, where the α_i are fresh type names with respect to the type-name stores in W; and a substitution γ that satisfies Γ, we would require that the two terms being related by $\mathcal{E}\,[\![A]\!]\,\rho$ (for $j \in \{1, 2\}$) be of the form:

$$\begin{array}{l}((e_j[\alpha_1/X_1]\ldots[\alpha_n/X_n] : A[\alpha_1/X_1]\ldots[\alpha_n/X_n] \stackrel{+\alpha_1}{\Longrightarrow} A[B_{1j}/X_1][\alpha_2/X_2]\ldots[\alpha_n/X_n])\ \ldots\\ \quad : A[B_{1j}/X_1]\ldots[B_{(n-1)j}/X_{n-1}][\alpha_n/X_n] \stackrel{+\alpha_n}{\Longrightarrow} A[B_{1j}/X_1]\ldots[B_{nj}/X_n])\end{array}$$

Clearly, the logical relation design we just sketched out is more complicated than the logical relation we present in this paper. The lesson is that every logical relation must "listen" to the (static and dynamic semantics of the) language at hand. The design of λB presents different challenges in the design of the logical relation compared to the related work by Neis et al. [2011] and Ahmed et al. [2011] even though all three use Kripke logical relations (which is to be expected for any language with RTG). We give a detailed comparison with the aforementioned related work in Section 6.

4.3 Parametricity and Soundness of the Logical Relation

In this section, we summarize the proofs for the Fundamental Property and soundness of the logical relation with respect to contextual equivalence. The full proofs are in the accompanying technical report [Ahmed et al. 2017].

The Fundamental Property is that any well-typed term of λB is related to itself by the logical relation.

Theorem 4.1 (Fundamental Property / Parametricity). *If* $\Sigma;\Delta;\Gamma \vdash e : A$, *then* $\Sigma;\Delta;\Gamma \vdash e \preceq e : A$.

The proof of the fundamental property is by induction on the derivation of $\Sigma;\Delta;\Gamma \vdash e : A$, relying on a compatibility lemma for each kind of expression in λB. The compatibility lemmas for conversion, type application, and cast are interesting.

For conversion, the proof boils down to showing that if e_1 and e_2 are related, applying a conversion to them, either positively or negatively, yields related expressions:

$$\begin{aligned}
&\Sigma;\Delta \vdash A \prec^{+\alpha} B \;\text{ and }\; (W, e_1, e_2) \in \mathcal{E}\,[\![A]\!]\,\rho \\
&\quad \text{implies } (W, (e_1 : \rho(A) \overset{+\alpha}{\Longrightarrow} \rho(B)), (e_2 : \rho(A) \overset{+\alpha}{\Longrightarrow} \rho(B))) \in \mathcal{E}\,[\![B]\!]\,\rho \\
&\Sigma;\Delta \vdash B \prec^{-\alpha} A \;\text{ and }\; \wedge\,(W, e_1, e_2) \in \mathcal{E}\,[\![B]\!]\,\rho \\
&\quad \text{implies } (W, (e_1 : \rho(B) \overset{-\alpha}{\Longrightarrow} \rho(A)), (e_2 : \rho(B) \overset{-\alpha}{\Longrightarrow} \rho(A))) \in \mathcal{E}\,[\![A]\!]\,\rho
\end{aligned}$$

We prove these two statements simultaneously by induction on the size of A, using numerous technical lemmas (e.g., anti-reduction, monotonicity, and compositionality).

The proof of compatibility for type application requires that we show

$$(W, v_1, v_2) \in \mathcal{V}\,[\![\forall X.\, A]\!]\,\rho \quad \text{implies} \quad (W, v_1\,[\rho(B)], v_1\,[\rho(B)]) \in \mathcal{E}\,[\![A[B/X]]\!]\,\rho$$

The proof of this does not require induction, but it uses the above property of conversion as well as many of the same technical lemmas.

The proof of compatibility for casts boils down to showing that if some expressions e_1 and e_2 are related, casting them yields related expressions:

$$\begin{aligned}
&\Sigma;\Delta \vdash A \prec B \;\text{ and }\; (W, e_1, e_2) \in \mathcal{E}\,[\![A]\!]\,\rho \\
&\quad \text{implies } (W, e_1 : \rho(A) \overset{p}{\Longrightarrow} \rho(B), e_2 : \rho(A) \overset{p}{\Longrightarrow} \rho(B)) \in \mathcal{E}\,[\![B]\!]\,\rho
\end{aligned}$$

We prove this by induction on the step index in W and the derivation of $\Sigma;\Delta \vdash A \prec B$. The induction on the step index is needed for the cases that cast to and from $\star$. We rely on the above property of type application for the cast from $\forall X.A'$ to $\star$.

Next we discuss the proof that the logical relation is sound with respect to contextual approximation (and therefore contextual equivalence).

Theorem 4.2 (Soundness w.r.t. Contextual Approximation).
If $\Sigma;\Delta;\Gamma \vdash e_1 \preceq e_2 : A$ *then* $\Sigma;\Delta;\Gamma \vdash e_1 \preceq^{ctx} e_2 : A$.

The proof follows the usual route of going through congruence and adequacy.

Lemma 4.3 (Congruence). *If* $\Sigma;\Delta;\Gamma \vdash e_1 \approx e_2 : A$ *and* $\vdash C : (\Sigma;\Delta;\Gamma \vdash A) \rightsquigarrow (\Sigma';\Delta';\Gamma' \vdash B)$ *then* $\Sigma';\Delta';\Gamma' \vdash C[e_1] \approx C[e_2] : B$.

Congruence is proved by induction on the typing derivation for C, using a weakening lemma for cases where C is empty, and the compatibility lemmas for all other cases.

Lemma 4.4 (Adequacy). *If* $\Sigma;\cdot;\cdot \vdash e_1 \approx e_2 : A$ *then* $\Sigma \triangleright e_1\Downarrow$ *if and only if* $\Sigma \triangleright e_2\Downarrow$.

Adequacy follows easily from the definition of the logical relation.

5 FREE THEOREM EXAMPLES

We consider two examples of obtaining theorems for free from parametricity. The first example, about the K combinator, comes from Ahmed et al. [2011]. The second example comes from Wadler [1989], and revisits our discussion of *swap* from Section 2.

We write $\Sigma \vdash e : A$ as shorthand for $\Sigma;\cdot;\cdot \vdash e : A$.

5.1 K Combinator

Ahmed et al. [2011] discuss the K combinator to motivate the design of the polymorphic lambda calculus, so it is illustrative to return to this example. On the left we have the untyped K combinator

and on the right we have the untyped K combinator again, but with a typo!

$$K^\star = \lceil \lambda(x).\,\lambda(y).\,x \rceil \qquad badK^\star = \lceil \lambda(x).\,\lambda(y).\,y \rceil$$
$$K = (K^\star : \star \overset{\ell_1}{\Longrightarrow} \forall X.\forall Y.X \to Y \to X) \qquad badK = (badK^\star : \star \overset{\ell_2}{\Longrightarrow} \forall X.\forall Y.X \to Y \to X)$$

Let us consider the behavior of these two combinators once they are cast to the polymorphic type that one would expect for the K combinator, that is, $\forall X.\forall Y.X \to Y \to X$. As explained by Ahmed et al. [2011], parametricity should tell us that a value of type $\forall X.\forall Y.X \to Y \to X$ is a polymorphic function that returns its first argument or is a function that diverges or a function that raises blame. Indeed, the above expression K is an example of the former and $badK$ is an example of the later. Furthermore, with our parametricity result in hand we can prove that in general, values of type $\forall X.\forall Y.X \to Y \to X$ behave as expected.

Theorem 5.1 (Free Theorem: K-Combinator).
If $\Sigma \vdash v : \forall X.\,\forall Y.\,X \to Y \to X$, $\Sigma \vdash v_1 : A$, *and* $\Sigma \vdash v_2 : B$, *then either*

(1) $\Sigma \triangleright v\ [A]\ [B]\ v_1\ v_2 \longmapsto^*\ \Sigma' \triangleright v_1'$ *and* $v_1' \approx^{ctx} v_1$, *for some* Σ', v_1', *or*
(2) $\Sigma \triangleright v\ [A]\ [B]\ v_1\ v_2 \Uparrow$, *or*
(3) $\Sigma \triangleright v\ [A]\ [B]\ v_1\ v_2 \longmapsto^*\ \Sigma' \triangleright \mathsf{blame}\, p$, *for some* Σ', p.

Proof (sketch)

(The full proof is in the accompanying technical report [Ahmed et al. 2017]). Here we give the high points.) We focus on case (1) where $v\ [A]\ [B]$ reduces to a value. Let $e = v\ [A]\ [B]\ v_1\ v_2$. We need to show that $v_1' \approx^{ctx} v_1$.
By the Fundamental Property (Theorem 4.1), we have

$$(W_0, v, v) \in \mathcal{V}\,[\![\forall X.\forall Y.X \to Y \to X]\!]\,\emptyset \tag{12}$$

The type applications reduce as follows.

$$\Sigma \triangleright v\ [A]\ [B] \longmapsto^*\ \Sigma' \triangleright e_1 : \alpha \to \beta \to \alpha \overset{+\alpha}{\Longrightarrow} A \to \beta \to A \overset{+\beta}{\Longrightarrow} A \to B \to A \tag{13}$$

where $\Sigma' = \Sigma, \alpha{:=}A, \beta{:=}B$. Let e' be the right-hand side of the above reduction. Let v_1'' be the result of normalizing $v_1 : A \overset{-\beta}{\Longrightarrow} A$, so we have $v_1'' \approx^{ctx} v_1$. From (12) and (13) we have

$$(W_1, e_1, e_1) \in \mathcal{E}\,[\![\alpha \to \beta \to \alpha]\!]\,\emptyset \tag{14}$$

with $W_1.\Sigma_1(\alpha) = W_1.\Sigma_2(\alpha) = A$, $W_1.\Sigma_1(\beta) = W_1.\Sigma_2(\beta) = B$, $W_1.\kappa(\alpha) = R_X$, and $W_1.\kappa(\beta) = R_Y$, where we choose

$$R_X = \{(W, v_1'', v_1'') \in \mathrm{Atom}^{\mathrm{val}}_{W_0.j}\,[A, A]\} \text{ and } R_Y = \{(W, v_2, v_2) \in \mathrm{Atom}^{\mathrm{val}}_{W_0.j}\,[B, B]\}. \tag{15}$$

Next, we have the reduction

$$\Sigma' \triangleright e'\ v_1\ v_2 \longmapsto^*\ \Sigma' \triangleright e_1\ (v_1'' : A \overset{-\alpha}{\Longrightarrow} \alpha)\ (v_2 : B \overset{-\beta}{\Longrightarrow} \beta).$$

From (15) we have

$$(W_2, v_1'' : A \overset{-\alpha}{\Longrightarrow} \alpha, v_1'' : A \overset{-\alpha}{\Longrightarrow} \alpha) \in \mathcal{V}\,[\![\alpha]\!]\,\emptyset \text{ and}$$
$$(W_2, v_2 : B \overset{-\beta}{\Longrightarrow} \beta, v_2 : B \overset{-\beta}{\Longrightarrow} \beta) \in \mathcal{V}\,[\![\beta]\!]\,\emptyset.$$

So from (14) we have that $e_1\ (v_1'' : A \overset{-\alpha}{\Longrightarrow} \alpha)\ (v_2 : B \overset{-\beta}{\Longrightarrow} \beta)$ reduces to a value $v_r : A \overset{-\alpha}{\Longrightarrow} \alpha$ and $(W_3, v_r, v_r) \in W_3.\kappa(\alpha)$ for some $W_3 \sqsupseteq W_1$. So $(W_3, v_r, v_r) \in R_X$ and therefore $v_r = v_1''$.

Finally, we have the reduction

$$(v_1'' : A \overset{-\alpha}{\Longrightarrow} \alpha \overset{+\alpha}{\Longrightarrow} A \overset{+\beta}{\Longrightarrow} A) \longrightarrow (v_1'' : A \overset{+\beta}{\Longrightarrow} A) \longrightarrow^* v_1' \text{ and } v_1' \approx^{ctx} v_1'' \approx^{ctx} v_1.$$

□

5.2 Rearrangements Commute with Map

Wadler [1989]'s introductory example of a free theorem showed that any function r of type $\forall X.\mathrm{List}\, X \to \mathrm{List}\, X$ (r is for "rearrangement") commutes with the *map* function. That is, for any function $f : A \to B$ and list $l : \mathrm{List}\, A$,

$$map\, f\ (r\ [A]\ l) = r\ [B]\ (map\, f\ l).$$

Here we derive the free theorem for rearrangements on pairs instead of lists. That is, any function r of type $\forall X.X \times X \to X \times X$ commutes with the pair-mapping operation: $map\, f\, \langle v_1, v_2\rangle = \langle f\ v_1, f\ v_2\rangle$. Note that functions of type $\forall X.X \times X \to X \times X$ include untyped functions that have been cast to this type, such as

$$swap^\star = \lceil \lambda(p).\, \langle \pi_2\, p, \pi_1\, p\rangle \rceil : \star \overset{\ell}{\Longrightarrow} \forall X.X \times X \to X \times X$$

$$id^\star = \lceil \lambda(x).\, x \rceil : \star \overset{\ell}{\Longrightarrow} \forall X.X \times X \to X \times X$$

$$bad^\star = \lceil \lambda(p).\, \langle \pi_1\, p + \pi_2\, p, 42\rangle \rceil : \star \overset{\ell}{\Longrightarrow} \forall X.X \times X \to X \times X$$

We write $e{\downarrow}$ for the value resulting from normalizing the expression e, if it exists.

Theorem 5.2 (Free Theorem: Rearrangement).
Suppose $\Sigma \vdash r : \forall X.\, X \times X \to X \times X$, $\Sigma \vdash f : A \to B$, $\Sigma \vdash v : A \times A$, $\Sigma \triangleright f\ (\pi_1\, v){\Downarrow}$, *and* $\Sigma \triangleright f\ (\pi_2\, v){\Downarrow}$. *Then* $\Sigma;\ \cdot\ ;\ \cdot \vdash map\, f\ (r\ [A]\ v) \approx^{ctx} r\ [B]\ (map\, f\ v) : B \times B$.

Proof (sketch)

(The full proof is in the accompanying technical report [Ahmed et al. 2017].) We focus on the case where r terminates with a value. The type applications reduce as follows

$$r\ [A] \longrightarrow r' : \alpha \times \alpha \to \alpha \times \alpha \overset{+\alpha}{\Longrightarrow} A \times A \to A \times A \qquad \text{and}$$

$$r\ [B] \longrightarrow r' : \alpha \times \alpha \to \alpha \times \alpha \overset{+\alpha}{\Longrightarrow} B \times B \to B \times B.$$

By the Fundamental Property (Theorem 4.1), we have

$$(W_0, r, r) \in \mathcal{V}\ [\![\forall X.X \times X \to X \times X]\!]\ \emptyset \qquad \text{and therefore}$$

$$(W_1, r', r') \in \mathcal{E}\ [\![\alpha \times \alpha \to \alpha \times \alpha]\!]\ \emptyset \tag{16}$$

in a world W_1 with $W_1.\Sigma_1(\alpha) = A$, $W_1.\Sigma_2(\alpha) = B$, and $W_1.\kappa(\alpha) = R_X$. We choose

$$R_X = \{(W, v', f\ v'{\downarrow}) \in \mathrm{Atom}^{\mathrm{val}}_{W'.j}\ [A, B] \mid \Sigma \vdash v' : A\}. \tag{17}$$

Let $v = \langle v_1, v_2\rangle$. We have

$$(W_1, v_1 : A \overset{-\alpha}{\Longrightarrow} \alpha, f\, v_1{\downarrow} : B \overset{-\alpha}{\Longrightarrow} \alpha) \in \mathcal{V}\ [\![\alpha]\!]\ \emptyset \quad \text{and}$$

$$(W_1, v_2 : A \overset{-\alpha}{\Longrightarrow} \alpha, f\, v_2{\downarrow} : B \overset{-\alpha}{\Longrightarrow} \alpha) \in \mathcal{V}\ [\![\alpha]\!]\ \emptyset.$$

Let $p = \langle v_1 : A \overset{-\alpha}{\Longrightarrow} \alpha, v_2 : A \overset{-\alpha}{\Longrightarrow} \alpha\rangle$ and $p' = \langle f\, v_1{\downarrow} : B \overset{-\alpha}{\Longrightarrow} \alpha, f\, v_2{\downarrow} : B \overset{-\alpha}{\Longrightarrow} \alpha\rangle$.
So $(W_1, p, p') \in \mathcal{V}\ [\![\alpha \times \alpha]\!]\ \emptyset$ and from (16) we have $(W_2, r'\, p, r'\, p') \in \mathcal{E}\ [\![\alpha \times \alpha]\!]\ \emptyset$. We have

the reductions

$$r'\,p \longrightarrow^* \langle v_3 : A \overset{-\alpha}{\Longrightarrow} \alpha, v_4 : A \overset{-\alpha}{\Longrightarrow} \alpha\rangle \quad \text{and}$$
$$r'\,p' \longrightarrow^* \langle v_3' : B \overset{-\alpha}{\Longrightarrow} \alpha, v_4' : B \overset{-\alpha}{\Longrightarrow} \alpha\rangle$$

so $(W_3, v_3, v_3') \in \mathcal{V}\,[\![\alpha]\!]\,\emptyset$ and $(W_3, v_4, v_4') \in \mathcal{V}\,[\![\alpha]\!]\,\emptyset$. Therefore, from (17) we have $v_3' = f\,v_3{\downarrow}$ and $v_4' = f\,v_4{\downarrow}$. Next we have the reductions

$$\langle v_3 : A \overset{-\alpha}{\Longrightarrow} \alpha, v_4 : A \overset{-\alpha}{\Longrightarrow} \alpha\rangle : \alpha \times \alpha \overset{+\alpha}{\Longrightarrow} A \times A \longrightarrow^* \langle v_3, v_4\rangle \quad \text{and}$$
$$\langle f\,v_3{\downarrow} : B \overset{-\alpha}{\Longrightarrow} \alpha, f\,v_4{\downarrow} : B \overset{-\alpha}{\Longrightarrow} \alpha\rangle : \alpha \times \alpha \overset{+\alpha}{\Longrightarrow} B \times B \longrightarrow^* \langle f\,v_3{\downarrow}, f\,v_4{\downarrow}\rangle.$$

To put this all together, we have

$$map\,f\,(r\,[A]\,v) \longrightarrow^* map\,f\,\langle v_3, v_4\rangle \longrightarrow^* \langle f\,v_3{\downarrow}, f\,v_4{\downarrow}\rangle \quad \text{and}$$
$$r\,[B]\,(map\,f\,v) \longrightarrow^* \langle f\,v_3{\downarrow}, f\,v_4{\downarrow}\rangle.$$

We conclude that $map\,f\,(r\,[A]\,\langle v_1, v_2\rangle) \approx^{ctx} r\,[B]\,\langle f\,v_1, f\,v_2\rangle$. □

6 RELATED WORK

The syntactic type abstraction of Grossman et al. [2000] inspired the conversion construct used in this paper. They use brackets to hide a host's knowledge that $X{=}A$ from a client; so their $[M]_h^B$ corresponds to our negative conversion $M : B[\alpha{:=}A] \overset{-\alpha}{\Longrightarrow} B$ and their $[M]_c^B$ corresponds to our positive conversion $M : B \overset{+\alpha}{\Longrightarrow} B[\alpha{:=}A]$. In their semantics, brackets push through λ, in contrast to our approach. Grossman et al. [2000] do not prove parametricity but they do prove a weaker value abstraction property.

Greenberg [2013] and Sekiyama et al. [2017] study a language that combines polymorphism and manifest contracts (aka. refinement types), but not type dynamic, and they prove parametricity. In their language parametricity is enforced statically (as in System F), so no dynamic enforcement is needed.

Sulzmann et al. [2007] study System F extended with type equality coercions, to serve as an intermediate language for Haskell. Their coercions have a similar purpose to the conversions of λB and their reduction rules push coercions through abstractions, similar to those of Ahmed et al. [2011] that we discussed in Section 2.4.

Non-parametric Polymorphism. Several language designs provide polymorphism but do not enforce parametricity. The argument in favor of this design choice is that it increases expressiveness. The languages include CAML [Leroy and Mauny 1991], Typed Racket [Tobin-Hochstadt and Felleisen 2006, 2008, 2010], and those generated by the Gradualizer [Cimini and Siek 2016, 2017].

Sealing for Parametricity. The use of sealing to enforce type abstraction goes back to Morris [1973], who described a language with a *Createseal* primitive that returns a pair of functions for sealing and unsealing, which respectively correspond to encryption and decryption. The distinction between sealing and RTG is that the seals are values, not types. Thus, sealing applies in untyped languages. Guha et al. [2007] use sealing in the design of polymorphic contracts for PLT Scheme.

Sumii and Pierce [2003] create a simply typed λ-calculus with cryptographic sealing. They define a logical relation, prove parametricity for the language, and present an embedding of System F into their calculus [Pierce and Sumii 2000]. Their logical relation uses possible worlds analogous to ours, but they connect relational interpretations to private keys instead of type names.

Pitts and Stark [1993] and Stark [1995] study a language with fresh name generation (akin to Scheme's `gensym`) and define a logical relation for reasoning about representation independence. It is straightforward to implement sealing using name generation and equality on names.

Takikawa et al. [2012] present a gradual type system for class-based code that uses sealing contracts to protect row-polymorphic functions on classes. Sealing is applied at the level of fields and methods, which hides abstracted behavior within an object. They present a formal model of gradually typed dynamic class composition with an operational semantics that carries extra information for contract monitoring and blame assignment, including ownership information for values—where a value's owners include any components that may affect the flow of that value. They prove type soundness and a property called *complete monitoring* [Dimoulas et al. 2012]. The latter ensures that sealing and unsealing are handled properly, in essence, ensuring row parametricity. Instead of using a logical relation, they use a standard progress-and-preservation-style proof, but at the cost of having to augment the operational semantics to track extra information such as ownership labels.

Runtime Type Generation. As discussed in Section 1, Matthews and Ahmed [2008] studied the integration of Scheme and ML by designing a multi-language that uses RTG to protect ML's polymorphic functions from Scheme's runtime type tests. Their multi-language contains boundaries that mediate between Scheme and ML code and are annotated with *conversion schemes* κ (which are essentially types). Boundaries have the form SM^{κ} (ML inside, Scheme outside: convert from κ to $\star$) and ${}^{\kappa}MS$ (Scheme inside, ML outside: convert from $\star$ to κ). Ahmed et al. [2011] later fixed their proof of parametricity, making cosmetic changes to the multi-language in the process. Their reduction rule for type application is as follows:

$$K \triangleright (\Lambda X.e)\ [B] \longrightarrow K, k \triangleright e[X := \langle k; B\rangle]$$

where K is a type-name store, k is a freshly generated type name, and $e[X := \langle k; B\rangle]$ denotes a *sealing substitution*, which replaces occurrences of X in the conversion schemes κ that appear on boundaries in e. Performing sealing substitution on conversion schemes provides two things: (1) local storage of type-name bindings, whereas in λB, type-name bindings reside in a global store; and (2) a means to insert many local conversions, whereas in λB, the type application rule performs one global conversion from $A[\alpha/X]$ to $A[B/X]$. Recall that at the end of Section 4.2, we discussed the ramifications of this global conversion on the design of the logical relation for λB. Due to the sealing substitutions and many-local-conversions aspects of the multi-language design, Ahmed et al. [2011] can parameterize $\mathcal{V}\ [\![A]\!]$ with a ρ that maps X to (B_1, B_2, R) without storing type names in ρ. A final distinction between the Scheme+ML multi-language and λB is that the former only allows casts to and from $\star$, while we support casts between many more (compatible) types, which makes our calculus more general.

In Section 1, we also discussed the work by Neis et al. [2009] who present a Kripke logical relation for G, a polymorphic language extended with Girard's non-parametric J operator, but without the type $\star$. Since their language is non-parametric, they have a conversion typing rule that essentially says that if $e : A$ and $A \approx A'$ under some type-name store Σ (where $A \approx A'$ allows all type names in A and A' to be replaced by the types they are bound to in Σ), then $e : A'$. In λB, we do not have such a conversion rule, which would break parametricity. These language distinctions lead to differences in the design of the logical relation as discussed in Section 4.2.

Finally, Igarashi et al. [2017] present work at the same conference that is complementary to this paper. They design a gradually typed variant of System F, called F_G, and prove that it satisfies the static part of the gradual guarantee [Siek et al. 2015] thanks to a careful definition of consistency for universal types. They give a dynamic semantics to F_G via translation to F_C, a calculus based

on the λB of this paper, though there are a few differences to note. They considered using the compatibility relation of this paper for F_G, but found that it broke conservativity of typing with respect to System F. So they instead develop a new consistency relation that solves that problem. As a result, they also use consistency in F_C instead of compatibility. A second difference is that Igarashi et al. [2017] differentiate between static and gradual type variables, building on the work of Garcia and Cimini [2015]. Only the gradual type variables require RTG; the static ones can be erased. The final difference between F_C and λB is that they replace conversions with the use of type environment application. Regarding the metatheory of F_C, they prove that it is type safe and they prove the blame theorem.

7 CONCLUSION

This paper settles a long-standing open question: can a language that integrates dynamic typing into a statically typed language with universal types satisfy parametricity? For a variant of the polymorphic blame calculus of Ahmed et al. [2011], we define a Kripke logical relation, prove the Fundamental Property (Theorem 4.1), and prove that the logical relation is sound with respect to contextual equivalence (Theorem 4.2). To demonstrate the utility of this parametricity result, we prove free theorems about the K combinator and about rearrangements.

Looking to the future, it would be interesting to see our parametricity result adapted to F_C of Igarashi et al. [2017]. We are also eager to see whether the dynamic aspect of the gradual guarantee holds for F_G, which requires challenging lemmas regarding F_C. Finally, we plan to investigate whether a coercion-based version of the polymorphic blame calculus can achieve space efficiency.

ACKNOWLEDGMENTS

This work was supported by EPSRC grant EP/K034413/1, by a Northeastern University Advanced Research and Creative Endeavor Award for Undergraduate Research, and by National Science Foundation grants CCF-1518844 and CCF-1453796. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. 1991. Dynamic Typing in a Statically Typed Language. *ACM Transactions on Programming Languages and Systems* 13, 2 (April 1991), 237–268.

M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. 1995. Dynamic typing in polymorphic languages. *Journal of Functional Programming* 5 (1995), 111–130.

Umut A. Acar, Amal Ahmed, and Matthias Blume. 2008. Imperative Self-Adjusting Computation. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*. 309–322.

Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *European Symposium on Programming (ESOP)*. 69–83.

Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic Foundations for Typed Assembly Languages. *ACM Transactions on Programming Languages and Systems* 32, 3 (March 2010), 1–67.

Amal Ahmed, Andrew W. Appel, and Roberto Virga. 2003. An Indexed Model of Impredicative Polymorphism and Mutable References. (Jan. 2003). Available at `http://www.cs.princeton.edu/~appel/papers/impred.pdf`.

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-Dependent Representation Independence. In *ACM Symposium on Principles of Programming Languages (POPL), Savannah, Georgia*.

Amal Ahmed, Robert Bruce Findler, Jeremy Siek, and Philip Wadler. 2011. Blame for All. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*. 201–214.

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. *Technical Report: Theorems for Free for Free: Parametricity, With and Without Types*. Technical Report. Perma.cc. https://perma.cc/L74G-6A8W

Amal Ahmed, Lindsey Kuper, and Jacob Matthews. 2011. Parametric polymorphism through run-time sealing, or, Theorems for low, low prices! (April 2011). Available at `http://www.ccs.neu.edu/home/amal/papers/paramseal-tr.pdf`.

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

Amal Ahmed, James T. Perconti, Jeremy G. Siek, and Philip Wadler. 2014. Blame for All (revised). (August 2014). draft.

Amal Jamil Ahmed. 2004. *Semantics of Types for Mutable State.* Ph.D. Dissertation. Princeton University.

Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. 2013. Gradual Typing for Smalltalk. *Science of Computer Programming* (Aug. 2013). Available online.

Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Lecture Notes in Computer Science, Vol. 8586. Springer Berlin Heidelberg, 257–281.

Gilad Bracha. 2011. *Optional Types in Dart.* Google.

Avik Chaudhuri. 2014. Flow: a static type checker for JavaScript. (2014). http://flowtype.org/

Matteo Cimini and Jeremy G. Siek. 2016. The Gradualizer: a methodology and algorithm for generating gradual type systems. In *Symposium on Principles of Programming Languages (POPL)*.

Matteo Cimini and Jeremy G. Siek. 2017. Automatically Generating the Dynamic Semantics of Gradually Typed Languages. In *Symposium on Principles of Programming Languages (POPL)*.

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *European Symposium on Programming (ESOP)*.

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

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, 303–315.

Jean-Yves Girard. 1972. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur.* Thèse de doctorat d'état. Université Paris VII, Paris, France.

Michael Greenberg. 2013. *Manifest Contracts.* Ph.D. Dissertation. University of Pennsylvania.

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2010. Contracts made manifest. In *ACM Symposium on Principles of Programming Languages (POPL), Madrid, Spain*. 353–364.

Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. 2006. Sage: Hybrid Checking for Flexible Specifications. In *Scheme and Functional Programming Workshop (Scheme)*. 93–104.

Dan Grossman, Greg Morrisett, and Steve Zdancewic. 2000. Syntactic Type Abstraction. *ACM Transactions on Programming Languages and Systems* 22, 6 (Nov. 2000), 1037–1080.

Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. 2007. Relationally-Parametric Polymorphic Contracts. In *Dynamic Languages Symposium (DLS)*. 29–40.

Robert Harper. 2013. *Practical Foundations for Programming Languages.* Cambridge University Press.

Anders Hejlsberg. 2010. C# 4.0 and beyond by Anders Hejlsberg. Microsoft Channel 9 Blog. (April 2010).

Anders Hejlsberg. 2012. Introducing TypeScript. Microsoft Channel 9 Blog. (2012).

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

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On Polymorphic Gradual Typing. *Proc. ACM Program. Lang.* 1, ICFP, Article 40 (Sept. 2017), 29 pages. DOI: http://dx.doi.org/10.1145/3110284

Lintaro Ina and Atsushi Igarashi. 2011. Gradual typing for generics. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA '11)*.

Xavier Leroy and Michel Mauny. 1991. *Dynamics in ML.* Springer Berlin Heidelberg, Berlin, Heidelberg, 406–426. DOI: http://dx.doi.org/10.1007/3540543961_20

Jacob Matthews and Amal Ahmed. 2008. Parametric polymorphism through run-time sealing, or, Theorems for low, low prices!. In *European Symposium on Programming (ESOP)*. 16–31.

Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-Language Programs. In *ACM Symposium on Principles of Programming Languages (POPL), Nice, France*. 3–10.

James H. Morris. 1968. *Lambda-calculus Models of Programming Languages.* Ph.D. Dissertation. MIT, Cambridge, MA, USA.

James H. Morris, Jr. 1973. Protection in Programming Languages. *Commun. ACM* 16, 1 (Jan. 1973), 15–21.

Georg Neis, Derek Dreyer, and Andreas Rossberg. 2009. Non-Parametric Parametricity. In *International Conference on Functional Programming (ICFP), Edinburgh, Scotland*. 135–148.

Georg Neis, Derek Dreyer, and Andreas Rossberg. 2011. Non-parametric parametricity. *Journal of Functional Programming* 21 (2011), 497–562.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. Dynamic Typing with Dependent Types. In *IFIP International Conference on Theoretical Computer Science*. 437–450.

Benjamin Pierce and Eijiro Sumii. 2000. Relating cryptography and polymorphism. (2000). www.cis.upenn.edu/~bcpierce/papers/infohide.ps Manuscript.

Andrew M. Pitts. 1998a. Existential Types: Logical Relations and Operational Equivalence. *Lecture Notes in Computer Science* 1443 (1998), 309–326.

Andrew M. Pitts. 1998b. *Existential types: Logical relations and operational equivalence.* Springer Berlin Heidelberg, Berlin, Heidelberg, 309–326. DOI: http://dx.doi.org/10.1007/BFb0055063

Andrew M. Pitts and Ian D. B. Stark. 1993. Observable Properties of Higher Order Functions That Dynamically Create Local Names, or What's New?. In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science (MFCS '93)*. Springer-Verlag, London, UK, UK, 122–141.

John C. Reynolds. 1974. Towards a Theory of Type Structure. In *Programming Symposium (LNCS)*, Vol. 19. Springer-Verlag, 408–425.

John C. Reynolds. 1983. Types, abstraction, and parametric polymorphism. *Information Processing* (1983), 513–523.

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

Andreas Rossberg. 2006. The Missing Link: Dynamic Components for ML. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*. ACM, New York, NY, USA, 99–110. DOI: http://dx.doi.org/10.1145/1159803.1159816

Taro Sekiyama, Atsushi Igarashi, and Michael Greenberg. 2017. Polymorphic Manifest Contracts, Revised and Resolved. *ACM Trans. Program. Lang. Syst.* 39, 1, Article 3 (Feb. 2017), 36 pages. DOI: http://dx.doi.org/10.1145/2994594

Peter Sewell. 2001. Modules, Abstract Types, and Distributed Versioning. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*. ACM, New York, NY, USA, 236–247.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop (Scheme)*. 81–92.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *SNAPL: Summit on Advances in Programming Languages (LIPIcs: Leibniz International Proceedings in Informatics)*.

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

Jeremy G. Siek and Philip Wadler. 2016. The key to blame: Gradual typing meets cryptography. (July 2016). http://homepages.inf.ed.ac.uk/wadler/topics/blame.html#blame-key (draft).

Ian Stark. 1995. *Names and Higher-Order Functions.* Ph.D. Dissertation. University of Cambridge.

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

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation.* ACM, New York, NY, USA, 53–66.

Eijiro Sumii and Benjamin Pierce. 2003. Logical relations for encryption. *J. Comput. Secur.* 11, 4 (July 2003), 521–554.

Nikhil Swamy, Cédric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin M. Bierman. 2014. Gradual typing embedded securely in JavaScript. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California.* 425–438.

Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual Typing for First-Class Classes. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*.

Satish Thatte. 1990. Quasi-static typing. In *ACM Symposium on Principles of Programming Languages (POPL)*. 367–381.

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *Dynamic Languages Symposium (DLS)*. 964–974.

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California.*

Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical Types for Untyped Languages. In *International Conference on Functional Programming (ICFP)*. ACM, 117–128.

Jonathan Turner. 2014. TypeScript and the Road to 2.0. (October 2014). https://blogs.msdn.microsoft.com/typescript/2014/10/22/typescript-and-the-road-to-2-0/

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

Philip Wadler. 1989. Theorems for free!. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*.

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

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