The key to blame: Gradual typing meets cryptography

Jeremy Siek
Indiana University, USA
jsiek@indiana.edu

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

Abstract

We connect three ways to achieve relational parametricity: universal types, runtime type generation, and cryptographic sealing. We study a polymorphic blame calculus, $\lambda\mathsf{B}$, inspired by that of Ahmed, Findler, Siek, and Wadler (2011), that ties universal types to runtime type generation; and a cryptographic lambda calculus, $\lambda\mathsf{K}$, inspired by that of Pierce and Sumii (2000), that relies on cryptographic sealing. Our $\lambda\mathsf{B}$ calculus avoids the 'topsy turvy' aspects of Ahmed *et al*., who evaluate terms one would expect to be values, and leave as values terms one would expect to be evaluated. We present translations from $\lambda\mathsf{B}$ to $\lambda\mathsf{K}$ and back that we show to be simulations. We extract from $\lambda\mathsf{B}$ the subset $\lambda\mathsf{G}$ that corresponds to the polymorphic lambda calculus $\lambda\mathsf{F}$ of Girard (1972) and Reynolds (1974); $\lambda\mathsf{G}$ is also a subset of the system G studied by Neis, Dreyer, and Rossberg (2009). We present translations from $\lambda\mathsf{F}$ to $\lambda\mathsf{G}$ and back that we show to be fully abstract. Further, we shed light on the embedding given by Pierce and Sumii of $\lambda\mathsf{F}$ into $\lambda\mathsf{K}$, describing how it is related to the composition of our translations from $\lambda\mathsf{F}$ to $\lambda\mathsf{G}$ and $\lambda\mathsf{B}$ to $\lambda\mathsf{K}$, and that the conversions and casts of $\lambda\mathsf{B}$ relate to the $\mathcal{C}$ and $\mathcal{G}$ components of their embedding.

1. Introduction

Researchers have discovered three mechanisms for enforcing type abstraction: *universal types*, proposed by Girard (1972) and Reynolds (1974), provide a static type discipline; *cryptographic sealing*, proposed by Morris (1973), provides a runtime mechanism for restricting access to a value; and *runtime type generation*, proposed by Abadi et al. (1995), Sewell (2001), and Rossberg (2003), provides a runtime mechanism for generating new type names and controlling access to their underlying type. Variations on these three mechanisms have been studied by authors ranging from Ahmed to Zdancewic. (See Section 6.)

In particular, Pierce and Sumii (2000) presented a cryptographic lambda calculus, and demonstrated that encryption and decryption relate to universal types by describing an embedding of polymorphic lambda calculus into their calculus. They conjectured that their embedding was fully abstract which remains open to this day.

Findler and Felleisen (2002) introduced contracts and blame, which informed a flowering of work on the use of gradual types to integrate dynamic and static typing. (Again, see Section 6.)

In particular, Ahmed et al. (2011) presented a polymorphic blame calculus designed to satisfy the relational parametricity of Reynolds (1983) despite allowing casts of untyped code to polymorphic type. However, their system is 'topsy turvy' in that it evaluates things one would expect to be values, and leaves as values things one would expect to be evaluated. Namely, it evaluates under type abstractions and takes some type generators to be values. The authors recognise these issues, but plea that their formalism forces these undesirable choices.

Here we present a new formulation of the polymorphic blame calculus, based on three constructions, runtime type generation, conversions, and casts. Conversions correspond to similar features previously studied by Grossman et al. (2000), Rossberg (2003), and Vytiniotis et al. (2005), while Matthews and Ahmed (2008) studied a boundary feature with aspects of both conversions and casts. Our calulus puts things to rights, in that type abstractions are values (adopting the value polymorphism restriction of Wright (1995) and Pitts (1998)), and type generators are always evaluated.

Our paper makes the following contributions.

- We introduce the polymorphic blame calculus, $\lambda\mathsf{B}$, inspired by that of Ahmed et al. (2011). Our system is based on runtime type generation, conversion, and casts, and does not suffer from the 'topsy turvy' problems of the previous work. Our system satisfies the usual properties of type safety and blame safety. (Section 2.)
- We introduce the cryptographic lambda calculus, $\lambda\mathsf{K}$, inspired by that of Pierce and Sumii (2000). Where they give a denotational semantics, we give a small-step operational semantics and type safety proof. We give translations from $\lambda\mathsf{B}$ to $\lambda\mathsf{K}$ and back again, and show they are simulations. The translations are surprising. (They surprised us, at least!) A name generator translates to *two* key generators, where one key is used for translating conversions (which never fail) and one key is used for translating casts (which may fail). Conversely, a key translates to a *one* name generator encapsulated by an existential type, with encoding and decoding each requiring both a conversion and a cast. (Section 3.)
- We introduce a polymorphic lambda calculus, $\lambda\mathsf{F}$, which is identical to that of Girard (1972) and Reynolds (1974), save that following Wright (1995) and Pitts (1998), bodies of type abstractions are restricted to values. We also introduce a polymorphic lambda calculus with runtime type generation, $\lambda\mathsf{G}$, which is the subset of $\lambda\mathsf{B}$ suitable as a target for translation from $\lambda\mathsf{F}$, and is also a subset of the system G studied by Neis et al. (2009). We give translations from $\lambda\mathsf{F}$ to $\lambda\mathsf{G}$ and back, and show that they are bisimulations and fully abstract. (Section 4.)
- We consider the embedding given by Pierce and Sumii of $\lambda\mathsf{F}$ into $\lambda\mathsf{K}$, and discuss how it relates to the composition of our translations from $\lambda\mathsf{F}$ to $\lambda\mathsf{G}$ and $\lambda\mathsf{B}$ to $\lambda\mathsf{K}$. The components $\mathcal{C}$ and $\mathcal{G}$ of their embedding correspond to two different constructs of $\lambda\mathsf{B}$, conversions and casts, offering insight into the structure of the translation. (Section 5.)

Figure 1 summarises the calculi and translations between them. Section 6 discusses related work and Section 7 concludes. Examples and proofs appear in the supplemental material.

$$\lambda\mathsf{F} \rightleftarrows \lambda\mathsf{G} \subseteq \lambda\mathsf{B} \rightleftarrows \lambda\mathsf{K}$$

Figure 1. Calculi and transformations

2. Polymorphic blame calculus, $\lambda\mathsf{B}$

2.1 Casts

Blame calculus integrates typed and untyped code. One scenario is that we begin with untyped code to which we wish to add types. Here is a program assembling untyped components.

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

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

Untyped code is surrounded by ceiling brackets, $\lceil \cdot \rceil$. Following a slogan of Dana Scott (Statman 1991; Harper 2013) we treat "untyped as unityped": untyped code is typed code where every term has the dynamic type $\star$. By convention, we add $\star$ to the name of untyped components.

It is trivial to rewrite a three-line program, 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.

A *cast* has the form

$$M : A \overset{+\ell}{\Longrightarrow} B$$

where M is a term of type A and the cast as a whole has type B. Here ℓ is a *blame label* used to ascribe fault if the cast fails. Writing both source and target types eases the formalism; a practical language might use more compact notation. We will also need casts of the form

$$M : A \overset{-\ell}{\Longrightarrow} B$$

which may arise as a result of reducing higher-order casts.

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

$$\begin{array}{l}\texttt{let}\ \mathit{inc}^\star = \lceil \lambda x.\, x+1 \rceil\ \texttt{in}\\ \texttt{let}\ \mathit{two} = (\Lambda X.\, \lambda f{:}X{\to}X.\, \lambda x{:}X.\, f\,(f\,x))\ \texttt{in}\\ \texttt{let}\ \mathit{two}^\star = (\mathit{two} : \forall X.(X{\to}X){\to}X{\to}X \Longrightarrow^{+\ell} \star)\ \texttt{in}\\ \lceil \mathit{two}^\star\ \mathit{inc}^\star\ 0 \rceil\end{array} \qquad (1)$$

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

$$\lceil \mathit{two}^\star\ \mathit{inc}^\star\ 0 \rceil \qquad \text{by} \qquad \lceil \mathit{two}^\star\ 0\ \mathit{inc}^\star \rceil \qquad (2)$$

it now evaluates to $\texttt{blame}\ {-\ell}$. Blaming $-\ell$ indicates fault lies with the *context containing* the cast.

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

$$\begin{array}{l}\texttt{let}\ \mathit{inc} = (\lambda x{:}\texttt{num}.\, x+1)\ \texttt{in}\\ \texttt{let}\ \mathit{two}^\star = \lceil \lambda f.\, \lambda x.\, f\,(f\,x) \rceil\ \texttt{in}\\ \texttt{let}\ \mathit{two} = (\mathit{two}^\star : \star \Longrightarrow^{+\ell} \forall X.(X{\to}X){\to}X{\to}X)\ \texttt{in}\\ \mathit{two}\ \texttt{num}\ \mathit{inc}\ 0\end{array} \qquad (3)$$

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

$$\lceil \lambda f.\, \lambda x.\, f\,(f\,x) \rceil \qquad \text{by} \qquad \lceil \lambda f.\, \lambda x.\, 2 \rceil \qquad (4)$$

it now evaluates to $\texttt{blame}\ {+\ell}$. Blaming $+\ell$ indicates fault lies with the *term contained* in the cast.

We introduce a precision ordering on types, written $A <:_n B$, where $\star$ is the least precise type, $A <:_n \star$ for all types A. The *blame safety* property assures that blame never falls on the more precisely typed side of a cast: so $M : A \Longrightarrow^{+\ell} B$ never blames $+\ell$ if $A <:_n B$ and never blames $-\ell$ if $B <:_n A$. In other words, "Well-typed programs can't be blamed" (Wadler and Findler 2009).

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

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

where m, n, o are fresh blame labels.

Casting a number to $\star$ and back to $\texttt{num}$ acts as the identity.

$$(2 : \texttt{num} \overset{+\ell}{\Longrightarrow} \star) : \star \overset{+m}{\Longrightarrow} \texttt{num} \longrightarrow 2$$

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

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

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

$$\begin{array}{l}(\lceil \lambda x.\, x+1 \rceil : \star \overset{+\ell}{\Longrightarrow} \texttt{num}{\to}\texttt{num})\ 2\\ \longrightarrow^* (\lambda x{:}\star.\, \lceil x+1 \rceil)\,(2 : \texttt{num} \overset{-\ell}{\Longrightarrow} \star) : \star \overset{+\ell}{\Longrightarrow} \texttt{num}\\ \longrightarrow^* \lceil 3 \rceil\end{array}$$

The blame label is negated on the contravariant cast.

2.2 Generators and conversion

A fundamental semantic property of polymorphic types is *relational parametricity*, introduced by Reynolds (1983) and popularised by Wadler (1989b) under the slogan "Theorems for free". For instance, any function of type $\forall X.X \to X$ must either be the identity function or the undefined function that ignores its argument and always loops or always yield blame. Our system is designed to guarantee relational parametricity, even for untyped code cast to a polymorphic type.

In particular, we have that

$$\begin{array}{l}\texttt{let}\ \mathit{id}^\star = \lceil \lambda x.\, x \rceil\ \texttt{in}\\ \texttt{let}\ \mathit{id} = (\mathit{id}^\star : \star \Longrightarrow^{+\ell} \forall X.X{\to}X)\ \texttt{in}\\ \mathit{id}\ \texttt{num}\ 1\end{array} \qquad (5)$$

evaluates to $1 : \texttt{num}$. If in (5) we replace

$$\lceil \lambda x.\, x \rceil \qquad \text{by} \qquad \lceil \lambda x.\, 2 \rceil \qquad (6)$$

it now evaluates to $\texttt{blame}\ {+\ell}$, because the term contained in the cast to $\forall X.X{\to}X$ did not behave parametrically. Similarly, if in the above we replace

$$\lceil \lambda x.\, x \rceil \qquad \text{by} \qquad \lceil \lambda x.\, x+1 \rceil \qquad (7)$$

it evaluates to $\texttt{blame}\,{+m}$, where m is introduced by the translation of $\lceil \lambda x.\, x+1 \rceil$ at the end of Section 2.1, indicating an attempt to access a value with abstract type as if it were a $\texttt{num}$.

The traditional way to reduce type application is by substitution. This cannot work in our case! To see why, consider reducing (5) by substituting $\texttt{num}$ for X, yielding

$$(\lceil \lambda x.\, x \rceil : \star \overset{+\ell}{\Longrightarrow} \texttt{num} \to \texttt{num})\ 1$$

which in turn evaluates to $1 : \texttt{num}$, satisfying parametricity. But if in the above we replace $\lceil \lambda x.\, x \rceil$ by $\lceil \lambda x.\, 2 \rceil$ or by $\lceil \lambda x.\, x+1 \rceil$ it now evaluates to $2 : \texttt{num}$, violating parametricity!

To solve this problem, we introduce two constructs, *generators* and *conversions*, and use them to rephrase type application.

A type name *generator* of the form

$$\nu X{:=}A.N$$

introduces a fresh name X associated with type A in the scope of term N. The type of N is also the type of the generator as a whole, and it must not contain X, since X is not meaningful outside the scope of the generator.

Within the scope of the generator, we can convert between name X and its associated type A. A *conversion* has the form

$$M : B \overset{+X}{\Longrightarrow} B[X{:=}A]$$

where M is a term of type B and the conversion as a whole has type $B[X{:=}A]$. Conversions differ from casts in that they never fail, and are decorated with a name rather a blame label. We will also need conversions of the form

$$M : B[X{:=}A] \stackrel{-X}{\Longrightarrow} B$$

which may arise as a result of reducing higher-order conversions.

We rewrite type applications by defining

$$L\,A \stackrel{\text{def}}{=} \nu X{:=}A.(L\,X : B \stackrel{+X}{\Longrightarrow} B[X{:=}A]) \quad \text{where } L : \forall X.B$$

which generates a fresh name X associated with type A, forms the type application $L\,X$, and converts the result from type B to type $B[X{:=}A]$. Here we exploit the "anti-Barendregt" convention of Pitts (2011) to choose the bound name in the universal type to be the same as the name introduced by the generator. The name generator separates name X from type A, which we need to ensure parametricity, while conversion brings the two together again; without conversion, the type application would have the wrong type, B rather than $B[X{:=}A]$.

Returning to example (5), the subterm $id\,\texttt{num}\,1$ is replaced by

$$(\nu X{:=}\texttt{num}.(id\,X : X{\to}X \stackrel{+X}{\Longrightarrow} \texttt{num}{\to}\texttt{num}))\,1$$

which eventually reduces to

$$X{:=}\texttt{num} \rhd (\lceil \lambda x.\,x \rceil : \star \stackrel{+\ell}{\Longrightarrow} X{\to}X \stackrel{+X}{\Longrightarrow} \texttt{num}{\to}\texttt{num})\,1$$

which in turn evaluates to $1 : \texttt{num}$, satisfying parametricity. And if we replace $\lceil \lambda x.\,x \rceil$ by $\lceil \lambda x.\,2 \rceil$ or by $\lceil \lambda x.\,x{+}1 \rceil$ it now evaluates to $\texttt{blame}\,{+\ell}$ or $\texttt{blame}\,{+m}$, again satisfying parametricity. The role of $\rhd$ and the reduction rules are discussed below, and reductions for examples (1)–(7) appear in the supplemental material.

2.3 Righting the topsy-turvy system

The material in the previous two sections corresponds largely to the approach taken in Ahmed et al. (2011). There are some minor differences in terminology and approach. In that paper, what we call 'casts' are called 'dynamic casts' and what we call 'conversions' are called 'static casts'. Notationally, we use $+\ell$ and $-\ell$ where they use ℓ and $\bar{\ell}$. In their work, conversions may be implicit or explicit, while we require conversions to be explicit since they play an important role in the translation to cryptographic lambda calculus and in the relation to the work of Pierce and Sumii (2000).

More significantly, as noted in the introduction, the system 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 (1998), one might expect the bodies of type abstractions to be restricted to values. They mention that this would be desirable, but it is impossible in their system due to their formulation of reductions for casts:

$$V : A{\to}B \stackrel{p}{\Longrightarrow} C{\to}D \longrightarrow \lambda w{:}C.\,V\,(w : C \stackrel{-p}{\Longrightarrow} A) : B \stackrel{p}{\Longrightarrow} D$$
$$V : A \stackrel{p}{\Longrightarrow} \forall X.B \longrightarrow \Lambda X.\,V : A \stackrel{p}{\Longrightarrow} B$$

The right-hand side of the second rule is a type abstraction with a body that is a 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 require evaluation underneath type abstractions. That in turn leads to a problem with generators, so instead of generating new names globally, they push generators inside of values:

$$\nu X{:=}A.c \longrightarrow c$$
$$\nu X{:=}A.\lambda y{:}B.\,N \longrightarrow \lambda y{:}B[X{:=}A].\,\nu X{:=}A.N$$
$$\nu X{:=}A.\Lambda X.\,V \longrightarrow \Lambda X.\,\nu X{:=}A.V$$

The second rule causes generators to be retained in function values, where one might expect the generator to be evaluated immediately.

It turns out, there is a simple way to avoid these convolutions. We reformulate the reductions for casts as follows:

$$(V : A{\to}B \stackrel{p}{\Longrightarrow} C{\to}D)\,W \longrightarrow V\,(W : C \stackrel{-p}{\Longrightarrow} A) : B \stackrel{p}{\Longrightarrow} D$$
$$(V : A \stackrel{p}{\Longrightarrow} \forall X.B)\,X \longrightarrow V : A \stackrel{p}{\Longrightarrow} B$$

In place of abstraction on the right we have application on the left. As a result, we are free to restrict the body of a type abstraction to be a value, 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. In turn, this permits more straightforward evaluation of generators. We let Σ range over name stores with entries of the form $X{:=}A$, and we write $\Sigma \rhd M$ for a configuration that pairs a name store with a term. The three reduction rules for generators given above are replaced by one rule:

$$\Sigma \rhd \nu X{:=}A.N \longrightarrow \Sigma,\ X{:=}A \rhd N$$

We choose X to be a fresh name not already in Σ. Generated names are kept in a global store, as in Neis et al. (2009, 2011), and generators are evaluated rather than retained in function values.

A drawback of the new formulation is that abstraction is no longer the only way to form values of function and universal type. In addition to considering function and type abstractions as values, we must also consider casts to function and universal types as values, and further we must do the same for conversions as well. However, other considerations also push in this direction; in particular, the space-efficient treatment of casts in Siek et al. (2015) also requires that casts to function type be taken as values. On balance, the solution we provide here seems simpler and more in line with other developments than that given by Ahmed et al. (2011).

2.4 Types and Terms

We now begin the formal development. Figure 2 presents the type rules of λB.

Let A, B, C, D range over types, which are either base type ι, function type $A \to B$, universal type $\forall X.B$, name X, or the dynamic type $\star$. Let ι range over base types, which include numbers $\texttt{num}$. Let G, H range over *ground types*, which are either base type ι, the function type $\star \to \star$, or name X. Casts to and from $\star$ factor through the ground types. Write $X \not\in A$ to indicate that name X does not occur free in type A.

Let ℓ, m, n, o range over simple labels, let Q range over *conversion labels* of the form $+X$ or $-X$, and let p, q range over *blame labels* of the form $+\ell$ or $-\ell$. Define involutions $-Q$ and $-p$ by

$$-(+X) = -X \qquad -(+\ell) = -\ell$$
$$-(-X) = +X \qquad -(-\ell) = +\ell$$

so that $-(-Q) = Q$ and $-(-p) = p$.

Let L, M, N range over terms, which are either constant c, variable x, operator application $op(\vec{M})$ where op is a operator and $\vec{M}$ is a sequence of zero or more terms, function abstraction $\lambda x{:}A.\,N$, function application $L\,M$, type abstraction $\Lambda X.\,V$, type application $L\,X$, name generator $\nu X{:=}A.N$, conversion $M : A \Longrightarrow^{Q} B$, cast $M : A \Longrightarrow^{p} B$, or blame $\texttt{blame}\,p$. In a type abstraction the body of the abstraction must be a value, and in a type application the type must be a name.

Let Γ range over type contexts, which are lists of hypotheses of the forms $X{:}\texttt{tp}$, $X{:=}A$, and $x{:}A$. Write $\Gamma\ \texttt{wf}$ to indicate that Γ is a well-formed type context. Write $X{:}\texttt{tp} \in \Gamma$ to indicate that $\Gamma\ \texttt{wf}$ and $X{:}\texttt{tp}$ appears in Γ, and similarly for $X{:=}A$ in Γ and $x{:}A \in \Gamma$.

Write $\Gamma \vdash A \prec^{Q} B$ to indicate that in context Γ types A and B are *convertible* under Q. Conversions must be between convertible types, and the rules for convertibility ensure that reductions preserve convertibility. Under hypothesis $X{:=}A$, judgements

Syntax

$$\begin{array}{rclrcl}
A, B, C, D & ::= & \iota \mid A \to B \mid \forall X.B \mid X \mid \star & \iota & ::= & \mathtt{num} \mid \cdots \\
G, H & ::= & \iota \mid \star \to \star \mid X & Q & ::= & +X \mid -X \\
\Gamma & ::= & \bullet \mid \Gamma,\ X{:}\mathtt{tp} \mid \Gamma,\ X{:=}A \mid \Gamma,\ x{:}A & p, q & ::= & +\ell \mid -\ell
\end{array}$$

$$L, M, N ::= c \mid op(\vec{M}) \mid x \mid \lambda x{:}A.\, N \mid L\, M \mid \Lambda X.\, V \mid L\, X \mid \nu X{:=}A.N \mid M : A \overset{Q}{\Longrightarrow} B \mid M : A \overset{p}{\Longrightarrow} B \mid \mathtt{blame}\, p$$

Contexts $\boxed{\Gamma\ \mathtt{wf}}$

$$\frac{}{\bullet\ \mathtt{wf}} \qquad \frac{\Gamma\ \mathtt{wf} \quad X \notin \Gamma}{\Gamma,\ X : \mathtt{tp}\ \mathtt{wf}} \qquad \frac{\Gamma\ \mathtt{wf} \quad X \notin \Gamma \quad \Gamma \vdash A : \mathtt{tp}}{\Gamma,\ X{:=}A\ \mathtt{wf}} \qquad \frac{\Gamma\ \mathtt{wf} \quad x \notin \Gamma \quad \Gamma \vdash A : \mathtt{tp}}{\Gamma,\ x : A\ \mathtt{wf}}$$

Types $\boxed{\Gamma \vdash A : \mathtt{tp}}$

$$\frac{\Gamma\ \mathtt{wf}}{\Gamma \vdash \iota : \mathtt{tp}} \qquad \frac{\Gamma \vdash A : \mathtt{tp} \quad \Gamma \vdash B : \mathtt{tp}}{\Gamma \vdash A \to B : \mathtt{tp}} \qquad \frac{\Gamma,\ X{:}\mathtt{tp} \vdash B : \mathtt{tp}}{\Gamma \vdash \forall X.B : \mathtt{tp}} \qquad \frac{X{:=}A \in \Gamma}{\Gamma \vdash X : \mathtt{tp}} \qquad \frac{X{:}\mathtt{tp} \in \Gamma}{\Gamma \vdash X : \mathtt{tp}} \qquad \frac{\Gamma\ \mathtt{wf}}{\Gamma \vdash \star : \mathtt{tp}}$$

Convertible $\boxed{\Gamma \vdash A \prec^{Q} B}$

$$\frac{\Gamma\ \mathtt{wf}}{\Gamma \vdash \iota \prec^{Q} \iota} \qquad \frac{\Gamma \vdash C \prec^{-Q} A \quad \Gamma \vdash B \prec^{Q} D}{\Gamma \vdash A \to B \prec^{Q} C \to D} \qquad \frac{\Gamma,\ X{:}\mathtt{tp} \vdash A \prec^{Q} B \quad X \notin Q}{\Gamma \vdash \forall X.A \prec^{Q} \forall X.B}$$

$$\frac{X{:=}A \in \Gamma}{\Gamma \vdash X \prec^{+X} A} \qquad \frac{X{:=}A \in \Gamma}{\Gamma \vdash A \prec^{-X} X} \qquad \frac{X : \mathtt{tp} \in \Gamma \quad X \notin Q}{\Gamma \vdash X \prec^{Q} X} \qquad \frac{\Gamma\ \mathtt{wf}}{\Gamma \vdash \star \prec^{Q} \star}$$

Compatible $\boxed{\Gamma \vdash A \prec B}$

$$\frac{\Gamma\ \mathtt{wf}}{\Gamma \vdash \iota \prec \iota} \qquad \frac{\Gamma \vdash C \prec A \quad \Gamma \vdash B \prec D}{\Gamma \vdash A \to B \prec C \to D} \qquad \frac{\Gamma,\ X{:}\mathtt{tp} \vdash A \prec B \quad X \notin A}{\Gamma \vdash A \prec \forall X.B} \qquad \frac{\Gamma \vdash A[X{:=}\star] \prec B}{\Gamma \vdash \forall X.A \prec B}$$

$$\frac{\Gamma\ \mathtt{wf} \quad X{:}\mathtt{tp} \in \Gamma}{\Gamma \vdash X \prec X} \qquad \frac{\Gamma \vdash A : \mathtt{tp}}{\Gamma \vdash A \prec \star} \qquad \frac{\Gamma \vdash A : \mathtt{tp}}{\Gamma \vdash \star \prec A}$$

Terms $\boxed{\Gamma \vdash M : A}$

$$\frac{\Gamma\ \mathtt{wf}}{\Gamma \vdash c : \mathsf{type}(c)} \qquad \frac{x{:}A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash \vec{M} : \vec{A} \quad \Gamma \vdash B : \mathtt{tp} \quad op : \vec{A} \to B}{\Gamma \vdash op(\vec{M}) : B}$$

$$\frac{\Gamma \vdash A : \mathtt{tp} \quad \Gamma,\ x{:}A \vdash N : B}{\Gamma \vdash (\lambda x{:}A.\, N) : A \to B} \qquad \frac{\Gamma \vdash L : A \to B \quad \Gamma \vdash M : A}{\Gamma \vdash (L\, M) : B} \qquad \frac{\Gamma,\ X{:}\mathtt{tp} \vdash V : B}{\Gamma \vdash (\Lambda X.\, V) : \forall X.B} \qquad \frac{\Gamma \vdash L : \forall X.B \quad X{:=}A \in \Gamma}{\Gamma \vdash (L\, X) : B}$$

$$\frac{\Gamma \vdash A : \mathtt{tp} \quad \Gamma,\ X{:=}A \vdash N : B \quad X \notin B}{\Gamma \vdash (\nu X{:=}A.N) : B} \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash A \prec^{Q} B}{\Gamma \vdash (M : A \overset{Q}{\Longrightarrow} B) : B} \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash A \prec B}{\Gamma \vdash (M : A \overset{p}{\Longrightarrow} B) : B} \qquad \frac{\Gamma \vdash A : \mathtt{tp}}{\Gamma \vdash \mathtt{blame}\, p : A}$$

Figure 2. Polymorphic blame calculus, λB (types)

$B \prec^{+X} C$ and $C \prec^{-X} B$ hold iff $C = B[X{:=}A]$. Further, $A \prec^{Q} B$ iff $B \prec^{-Q} A$. Write $X \notin Q$ if Q is not $+X$ or $-X$. Convertability is almost reflexive, in that $A \prec^{Q} A$ if $X \notin Q$ for every free name X in A.

Write $\Gamma \vdash A \prec B$ to indicate that in context Γ types A and B are *compatible*. Casts must be between compatible types, and the rules for compatibility ensure that reductions preserve compatibility. For every type B we have both $\forall X.B \prec B[X{:=}\star]$ and $B[X{:=}\star] \prec \forall X.B$. However, compatiblity is not symmetric since $\forall X.B \prec B[X{:=}A]$ for all A, but $B[X{:=}A] \not\prec \forall X.B$ when $A \neq \star$. Compatibility is reflexive but not transitive, since we always have $A \prec \star$ and $\star \prec B$, even when $A \not\prec B$.

Write $\Gamma \vdash M : A$ to indicate that in context Γ term M has type A. Typing for constants, operators, function abstraction, function application, type abstraction, and type application is standard. Each constant c has type $\mathsf{type}(c)$. Typing for name generators, casts, and conversions is as discussed previously, and $\mathtt{blame}\, p$ has type A for every valid type A.

Every well-typed term with no subterms of the form $\mathtt{blame}\, p$ has a unique type. We also have the following.

Lemma 1 (Well-formedness)**.**

- *If* $\Gamma \vdash A : \mathtt{tp}$ *then* $\Gamma\ \mathtt{wf}$.
- *If* $\Gamma \vdash A \prec^{Q} B$ *then* $\Gamma \vdash A : \mathtt{tp}$ *and* $\Gamma \vdash B : \mathtt{tp}$.
- *If* $\Gamma \vdash A \prec B$ *then* $\Gamma \vdash A : \mathtt{tp}$ *and* $\Gamma \vdash B : \mathtt{tp}$.
- *If* $\Gamma \vdash M : A$ *then* $\Gamma \vdash A : \mathtt{tp}$.

If $\Gamma\ \mathtt{wf}$ it is easy to verify that $\Gamma \vdash A \prec^{Q} B$ and $\Gamma \vdash A \prec B$ imply $\Gamma \vdash A : \mathtt{tp}$ and $\Gamma \vdash B : \mathtt{tp}$, and that $\Gamma \vdash M : A$ implies $\Gamma \vdash A : \mathtt{tp}$.

Untyped lambda calculus embeds into blame calculus.

$$\begin{aligned}
\lceil c \rceil &= c : \mathsf{type}(c) \overset{+\ell}{\Longrightarrow} \star \\
\lceil op(\vec{M}\,) \rceil &= op(\lceil \vec{M} \rceil : \vec{\star} \overset{+\ell}{\Longrightarrow} \vec{A}) : B \overset{+\ell}{\Longrightarrow} \star, \quad \text{if } op : \vec{A} {\to} B \\
\lceil x \rceil &= x \\
\lceil \lambda x.\, N \rceil &= (\lambda x{:}\star\, .\, \lceil N \rceil) : \star{\to}\star \overset{+\ell}{\Longrightarrow} \star \\
\lceil L\, M \rceil &= (\lceil L \rceil : \star \overset{+\ell}{\Longrightarrow} \star{\to}\star)\, \lceil M \rceil
\end{aligned}$$

The embedding introduces fresh blame labels for each cast.

We abbreviate a sequence of casts in the obvious way,

$$M : A \overset{p}{\Longrightarrow} B \overset{q}{\Longrightarrow} C \overset{\text{def}}{=} (M : A \overset{p}{\Longrightarrow} B) : B \overset{q}{\Longrightarrow} C,$$

and similarly for combinations of conversions and casts.

2.5 Reductions

Figure 3 presents the reduction rules of the polymorphic blame calculus.

Let V, W range over values, which include constants, function abstractions, and type abstractions. A value whose type is a name is a conversion of the form $V : A \Longrightarrow^{-X} X$, and a value of dynamic type is a cast of the form $V : G \Longrightarrow^{p} \star$. In addition, conversions or casts to a function or a universal type are values, which reduce when applied to a value or name, respectively. A cast $V : A \Longrightarrow^{p} \forall X.B$ is a value only if A is not a universal type.

Let Σ range over stores, which are type contexts consisting only of hypotheses $X{:=}A$. Let $\mathcal{E}$ range over evaluation contexts, which are standard. Write $\Sigma \triangleright M$ for a *configuration*, used in reduction to ensure each name allocated is fresh, and to record the association of names with types, which will help to formulate type preservation.

Write $M \longmapsto N$ for reduction of terms. Reductions for operators are standard. Each operator op is specified by a total meaning function $[\![op]\!]$ that preserves types: if $op : \vec{A} \to B$ and $\vec{V} : \vec{A}$, then $[\![op]\!](\vec{V})$ is a value W such that $W : B$. Reduction for function and type applications is also standard, save that type applications are restricted to names, with substitution implemented via the anti-Barendregt convention.

Conversions are reduced as follows. Conversion from a base type, name, or the dynamic type to the same type is the identity; a name converts to itself only if the conversion label refers to a different name. 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, when applied to a name, reduces to a conversion between the instances of the universal types. In scope of hypothesis $X{:=}A$, conversion from A to X and back reduces to the identity.

Casts are reduced as follows. A cast from a base type, name, or the dynamic type to the same type is the identity. A cast between two function types, when applied to a value, reduces to a contravariant cast on the domain (negating the label) and a covariant cast on the range. A cast to universal type, when applied to a name, reduces to a cast to the instance of the universal type. A cast from universal type reduces by instantiating the value at type $\star$. The last rule is equivalent to

$$V : \forall X.A \overset{p}{\Longrightarrow} B \longmapsto \nu X{:=}\star.(V\, X : A \overset{+X}{\Longrightarrow} A[X{:=}\star] \overset{p}{\Longrightarrow} B)$$

by the definition of type application. Given a cast from universal type to universal type, the rules may apply in in either order, as both orders yield the same result.

Finally we discuss casts to and from $\star$. Say A is *uniquely groundable*, and write $\mathsf{ug}(A)$, if $A \neq \star$, and $A \neq G'$ for any ground type G', and $A \neq \forall X.A'$ for any A'. If $\mathsf{ug}(A)$ then there is a unique G such that $A \prec G$ and $G \prec A$. (In fact, if $\mathsf{ug}(A)$ then $A = A' {\to} B'$ for some A' and B', and if $\mathsf{ug}(A)$ and $A \prec G$ or $G \prec A$ then $G = \star{\to}\star$. But our formulation adapts if we permit other ground types, such as $\star \times \star$.) If $\mathsf{ug}(A)$ and $A \prec G$, then a cast from A to $\star$ reduces to a cast from A to G followed by a cast from G to $\star$, and similarly for a cast from $\star$ to A. Casts from a ground type G to type $\star$ and back to type G reduce to the identity (a successful cast), while casts from ground type G to type $\star$ and back to a different ground type H allocate blame to the label of the outer cast (an unsuccessful cast).

Write $\Sigma \triangleright M \longrightarrow \Sigma' \triangleright M'$ for reduction of configurations. Reduction is closed under evaluation contexts, evaluating blame terminates reduction, and evaluating a generator adds a fresh name to the configuration. In general, if $\longrightarrow$ is a reduction relation, write $\longrightarrow^{?}$ for its reflexive closure and $\longrightarrow^{*}$ for its reflexive and transitive closure. Detailed reductions for examples (1)–(7) from Sections 2.1 and 2.2 appear in the supplemental material.

2.6 Type safety

The usual type safety properties hold. The canonical forms lemma needs to be adjusted to account for conversions and casts that are taken as values.

Lemma 2 (Canonical forms)**.** *If $V : A$ then either*

- $V = c'$ *and* $A = \iota'$
- $V = \lambda x : A'.\, N'$ *and* $A = A' \to B'$
- $V = V' : A'{\to}B' \Longrightarrow^{Q'} C'{\to}D'$ *and* $A = C'{\to}D'$
- $V = V' : A'{\to}B' \Longrightarrow^{p'} C'{\to}D'$ *and* $A = C'{\to}D'$
- $V = \Lambda X.\, V'$ *and* $A = \forall X.B'$
- $V = V' : \forall X.A' \Longrightarrow^{Q'} \forall X.B'$ *and* $A = \forall X.B'$
- $V = V' : A' \Longrightarrow^{p'} \forall X.B'$ *and* $A = \forall X.B'$
- $V = V' : G' \Longrightarrow^{p'} \star$ *and* $A = \star$
- $V = V' : A' \Longrightarrow^{-X'} X'$ *and* $A = X'$

where all primed variables are existentially quantified.

We have the usual preservation and progress results.

Proposition 3 (Type safety)**.**

1. *If $\Sigma \vdash M : A$ and $\Sigma \triangleright M \longrightarrow \Sigma' \triangleright M'$ then $\Sigma' \vdash M' : A$.*
2. *If $\Sigma \vdash M : A$ then either*
   - $M = V'$
   - $M = \mathtt{blame}\, p'$
   - $\Sigma \triangleright M \longrightarrow \Sigma' \triangleright M'$

   where all primed variables are existentially quantified.

Convertibility and compatibility exactly ensure that reductions of conversions and casts preserve typability. For instance, the compatibility rule

$$\frac{C \prec^{-Q} A \qquad B \prec^{Q} D}{A \to B \prec^{Q} C \to D}$$

ensures for the reduction

$$(V : A{\to}B \overset{Q}{\Longrightarrow} C{\to}D)\, W \longmapsto V\,(W : C \overset{-Q}{\Longrightarrow} A) : B \overset{Q}{\Longrightarrow} D$$

that if the conversion on the left is well-typed then the two conversions on the right are also well-typed. Similarly for each convertibility and compatibility rule and the corresponding reduction for conversions or casts.

2.7 Blame safety

Preservation and progress on their own are relatively weak, in that they permit blame to arise. We now consider under what circumstances we can guarantee absence of blame.

Blame safety is defined in terms of three relations. Positive and negative subtyping, $A <:^{+} B$ and $A <:^{-} B$, characterise when a cast $A \Longrightarrow^{p} B$ can never result in blaming p or $-p$, respectively. Naive subtyping, $A <:_{n} B$, characterises when type A is more *precise* than type B. The three relations are as given by Ahmed et al. (2011), and are presented in the supplemental material.

As noted by Ahmed et al. (2011), positive, negative, and naive subtyping are closely related.

Proposition 4 (Factoring)**.** *$A <:_{n} B$ iff $A <:^{+} B$ and $B <:^{-} A$.*

Say that a cast $A \Longrightarrow^{q} B$ is *safe* for p if $p = q$ and $A <:^{+} B$ or $p = -q$ and $A <:^{-} B$ or $p \neq q$ and $p \neq -q$. A term M is safe for

Syntax

$$\begin{array}{rl} V, W ::= & c \mid \lambda x{:}A.\, N \mid \Lambda X.\, V \mid V : A \to B \overset{Q}{\Longrightarrow} C \to D \mid V : \forall X.A \overset{Q}{\Longrightarrow} \forall X.B \mid V : A \overset{-X}{\Longrightarrow} X \mid \\ & V : A \to B \overset{p}{\Longrightarrow} C \to D \mid V : A \overset{p}{\Longrightarrow} \forall X.B \mid V : G \overset{p}{\Longrightarrow} \star \\ \Sigma ::= & \bullet \mid \Sigma,\ X{:=}A \\ \mathcal{E} ::= & \square \mid op(\vec{V}, \mathcal{E}, \vec{M}) \mid \mathcal{E}\, M \mid V\, \mathcal{E} \mid \mathcal{E}\, X \mid \mathcal{E} : A \overset{Q}{\Longrightarrow} B \mid \mathcal{E} : A \overset{p}{\Longrightarrow} B \end{array}$$

Reduction $\boxed{M \longmapsto N}$

$$\begin{array}{rcl} op(\vec{V}) & \longmapsto & [\![op]\!](\vec{V}) \\ (\lambda x{:}A.\, N)\, V & \longmapsto & N[x{:=}V] \\ (\Lambda X.\, V)\, X & \longmapsto & V \\ \\ V : \iota \overset{Q}{\Longrightarrow} \iota & \longmapsto & V \\ (V : A \to B \overset{Q}{\Longrightarrow} C \to D)\, W & \longmapsto & V\, (W : C \overset{-Q}{\Longrightarrow} A) : B \overset{Q}{\Longrightarrow} D \\ (V : \forall X.A \overset{Q}{\Longrightarrow} \forall X.B)\, X & \longmapsto & V\, X : A \overset{Q}{\Longrightarrow} B \quad \text{if } X \notin Q \\ V : X \overset{Q}{\Longrightarrow} X & \longmapsto & V \quad \text{if } X \notin Q \\ V : A \overset{-X}{\Longrightarrow} X \overset{+X}{\Longrightarrow} A & \longmapsto & V \\ V : \star \overset{Q}{\Longrightarrow} \star & \longmapsto & V \end{array}$$

$$\begin{array}{rcl} V : \iota \overset{p}{\Longrightarrow} \iota & \longmapsto & V \\ (V : A \to B \overset{p}{\Longrightarrow} C \to D)\, W & \longmapsto & V\, (W : C \overset{-p}{\Longrightarrow} A) : B \overset{p}{\Longrightarrow} D \\ (V : A \overset{p}{\Longrightarrow} \forall X.B)\, X & \longmapsto & V : A \overset{p}{\Longrightarrow} B \\ V : \forall X.A \overset{p}{\Longrightarrow} B & \longmapsto & (V\, \star) : A[X{:=}\star] \overset{p}{\Longrightarrow} B \\ V : X \overset{p}{\Longrightarrow} X & \longmapsto & V \\ V : \star \overset{p}{\Longrightarrow} \star & \longmapsto & V \\ V : A \overset{p}{\Longrightarrow} \star & \longmapsto & V : A \overset{p}{\Longrightarrow} G \overset{p}{\Longrightarrow} \star \quad \text{if } \mathsf{ug}(A), A \prec G \\ V : \star \overset{p}{\Longrightarrow} A & \longmapsto & V : \star \overset{p}{\Longrightarrow} G \overset{p}{\Longrightarrow} A \quad \text{if } \mathsf{ug}(A), G \prec A \\ V : G \overset{p}{\Longrightarrow} \star \overset{q}{\Longrightarrow} G & \longmapsto & V \\ V : G \overset{p}{\Longrightarrow} \star \overset{q}{\Longrightarrow} H & \longmapsto & \mathtt{blame}\, q \quad \text{if } G \neq H \end{array}$$

Reduction of configurations $\boxed{\Sigma \rhd M \longrightarrow \Sigma' \rhd N}$

$$\frac{M \longmapsto N}{\Sigma \rhd \mathcal{E}[M] \longrightarrow \Sigma \rhd \mathcal{E}[N]} \qquad \frac{}{\Sigma \rhd \mathcal{E}[\mathtt{blame}\, p] \longrightarrow \Sigma \rhd \mathtt{blame}\, p} \qquad \frac{X \notin \Sigma}{\Sigma \rhd \mathcal{E}[\nu X{:=}A.N] \longrightarrow \Sigma,\ X{:=}A \rhd \mathcal{E}[N]}$$

Figure 3. Polymorphic blame calculus, λB (reductions)

Syntax

$$\begin{array}{rl} A, B, C, D ::= & \iota \mid A \to B \mid \mathtt{key}\langle A \rangle \mid \mathtt{bits} \\ L, M, N ::= & c \mid op(\vec{M}) \mid x \mid \lambda x{:}A.\, N \mid L\, M \mid \mathtt{blame}\, p \mid \kappa \mid \mathtt{new}\langle A \rangle \mid \lfloor M \rfloor_N \mid \lceil L \rceil^p_N \\ V, W ::= & c \mid \lambda x{:}A.\, N \mid \kappa \mid \lfloor V \rfloor_\kappa \\ \Gamma ::= & \bullet \mid \Gamma,\ x{:}A \mid \Gamma,\ \kappa{:}\mathtt{key}\langle A \rangle \\ \Delta ::= & \bullet \mid \Delta,\ \kappa{:}\mathtt{key}\langle A \rangle \\ \mathcal{E} ::= & \square \mid op(\vec{V}, \mathcal{E}, \vec{M}) \mid \mathcal{E}\, M \mid V\, \mathcal{E} \mid \lfloor \mathcal{E} \rfloor_N \mid \lfloor V \rfloor_{\mathcal{E}} \mid \lceil \mathcal{E} \rceil^p_N \mid \lceil V \rceil^p_{\mathcal{E}} \end{array}$$

Terms $\boxed{\Gamma \vdash M : A}$

$$\frac{}{\mathtt{new}\langle A \rangle : \mathtt{key}\langle A \rangle} \qquad \frac{M : A \qquad N : \mathtt{key}\langle A \rangle}{\lfloor M \rfloor_N : \mathtt{bits}} \qquad \frac{L : \mathtt{bits} \qquad N : \mathtt{key}\langle A \rangle}{\lceil L \rceil^p_N : A}$$

Reduction $\boxed{M \longmapsto N}$

$$\begin{array}{rclcrcl} op(\vec{V}) & \longmapsto & [\![op]\!](\vec{V}) & \qquad & \lceil \lfloor V \rfloor_\kappa \rceil^p_\kappa & \longmapsto & V \\ (\lambda x{:}A.\, N)\, V & \longmapsto & N[x{:=}V] & & \lceil \lfloor V \rfloor_\kappa \rceil^p_{\kappa'} & \longmapsto & \mathtt{blame}\, p \quad \text{if } \kappa \neq \kappa' \end{array}$$

Reduction of configurations $\boxed{\Delta \rhd M \longrightarrow \Delta' \rhd M'}$

$$\frac{M \longrightarrow N}{\Delta \rhd \mathcal{E}[M] \longrightarrow \Delta \rhd \mathcal{E}[N]} \qquad \frac{}{\Delta \rhd \mathcal{E}[\mathtt{blame}\, p] \longrightarrow \Delta \rhd \mathtt{blame}\, p} \qquad \frac{\kappa \notin \Delta}{\Delta \rhd \mathcal{E}[\mathtt{new}\langle A \rangle] \longrightarrow \Delta,\ \kappa : \mathtt{key}\langle A \rangle \rhd \mathcal{E}[\kappa]}$$

Figure 4. Cryptographic lambda calculus, λK

p, written M safe p, if every cast within it is safe for p. We have variants of preservation and progress.

Proposition 5 (Blame safety)**.**

1. *If M safe p and $\Sigma \rhd M \longrightarrow \Sigma' \rhd M'$ then M' safe p.*
2. *If M safe p then $\Sigma \rhd M \not\longrightarrow \Sigma' \rhd \mathtt{blame}\, p$.*

A corollary of Propositions 4 and 5 is that if $A <:_n B$ then a cast $A \Longrightarrow^p B$ is safe for p, and a cast $B \Longrightarrow^p A$ is safe for $-p$. Hence, if a cast between a less-precise and a more-precise type fails, blame always falls on the less-precise side of the cast.

Types $\boxed{\llparenthesis A \rrparenthesis^{\mathsf{BK}}}$

$$\begin{aligned}
\llparenthesis \iota \rrparenthesis &= \iota \\
\llparenthesis A \to B \rrparenthesis &= \llparenthesis A \rrparenthesis \to \llparenthesis B \rrparenthesis \\
\llparenthesis \forall X. B \rrparenthesis &= \mathtt{key}\langle\mathtt{bits}\rangle \to \llparenthesis B \rrparenthesis \\
\llparenthesis X \rrparenthesis &= \mathtt{bits} \\
\llparenthesis \star \rrparenthesis &= \mathtt{bits}
\end{aligned}$$

Ground types $\boxed{\llparenthesis G \rrparenthesis_R^{\mathsf{BK}}}$

$$\begin{aligned}
\llparenthesis \iota \rrparenthesis_R &= \kappa_\iota \\
\llparenthesis \star \to \star \rrparenthesis_R &= \kappa_{\star\to\star} \\
\llparenthesis X \rrparenthesis_R &= k && \text{if } X \mapsto (j,k) \in R
\end{aligned}$$

Terms $\boxed{\llparenthesis M \rrparenthesis_R^{\mathsf{BK}}}$

$$\begin{aligned}
\llparenthesis c \rrparenthesis_R &= c \\
\llparenthesis x \rrparenthesis_R &= x \\
\llparenthesis \lambda x{:}A.\, N \rrparenthesis_R &= \lambda x : \llparenthesis A \rrparenthesis.\, \llparenthesis N \rrparenthesis_R \\
\llparenthesis \Lambda X.\, V \rrparenthesis_R &= \lambda k : \mathtt{key}\langle\mathtt{bits}\rangle.\, \llparenthesis V \rrparenthesis_{R,\, X\mapsto(\bullet,k)}
\end{aligned}
\qquad
\begin{aligned}
\llparenthesis op(\vec{M}) \rrparenthesis_R &= op(\llparenthesis \vec{M} \rrparenthesis_R) \\
\llparenthesis \mathtt{blame}\, p \rrparenthesis_R &= \mathtt{blame}\, p \\
\llparenthesis L\, M \rrparenthesis_R &= \llparenthesis L \rrparenthesis_R\, \llparenthesis M \rrparenthesis_R \\
\llparenthesis L\, X \rrparenthesis_R &= \llparenthesis L \rrparenthesis_R\, k \qquad \text{if } (X \mapsto (j,k)) \in R
\end{aligned}$$

$$\begin{aligned}
\llparenthesis \nu X{:=}A.N \rrparenthesis_R &= \mathtt{let}\, j = \mathtt{new}\langle \llparenthesis A \rrparenthesis \rangle\ \mathtt{in}\ \mathtt{let}\, k = \mathtt{new}\langle\mathtt{bits}\rangle\ \mathtt{in}\ \llparenthesis N \rrparenthesis_{R,\, X\mapsto(j,k)} \\
\llparenthesis M : A \xRightarrow{Q} B \rrparenthesis_R &= \llparenthesis M \rrparenthesis_R \mathbin{@} \llparenthesis A \xRightarrow{Q} B \rrparenthesis_R \\
\llparenthesis M : A \xRightarrow{p} B \rrparenthesis_R &= \llparenthesis M \rrparenthesis_R \mathbin{@} \llparenthesis A \xRightarrow{p} B \rrparenthesis_R
\end{aligned}$$

Conversions $\boxed{\llparenthesis A \xRightarrow{Q} B \rrparenthesis_R^{\mathsf{BK}}}$

$$\begin{aligned}
\llparenthesis \iota \xRightarrow{Q} \iota \rrparenthesis_R &= \overline{\lambda} v{:}\iota.\, v \\
\llparenthesis X \xRightarrow{Q} X \rrparenthesis_R &= \overline{\lambda} v{:}\mathtt{bits}.\, v && \text{if } X \notin Q \\
\llparenthesis \star \xRightarrow{Q} \star \rrparenthesis_R &= \overline{\lambda} v{:}\mathtt{bits}.\, v
\end{aligned}
\qquad
\begin{aligned}
\llparenthesis A \xRightarrow{-X} X \rrparenthesis_R &= \overline{\lambda} v{:}\llparenthesis A \rrparenthesis.\, \lfloor v \rfloor_j && \text{if } (X \mapsto (j,k)) \in R \\
\llparenthesis X \xRightarrow{+X} A \rrparenthesis_R &= \overline{\lambda} v{:}\mathtt{bits}.\, \lceil v \rceil_j^{\bullet} && \text{if } (X \mapsto (j,k)) \in R
\end{aligned}$$

$$\begin{aligned}
\llparenthesis A \to B \xRightarrow{Q} C \to D \rrparenthesis_R &= \overline{\lambda} v{:}\llparenthesis A \to B \rrparenthesis.\, \lambda w{:}\llparenthesis C \rrparenthesis.\, (v\, (w \mathbin{@} \llparenthesis C \xRightarrow{-Q} A \rrparenthesis_R)) \mathbin{@} \llparenthesis B \xRightarrow{Q} D \rrparenthesis_R \\
\llparenthesis \forall X.A \xRightarrow{Q} \forall X.B \rrparenthesis_R &= \overline{\lambda} v{:}\mathtt{key}\langle\mathtt{bits}\rangle{\to}\llparenthesis A \rrparenthesis.\, \lambda k{:}\mathtt{key}\langle\mathtt{bits}\rangle.\, (v\, k) \mathbin{@} \llparenthesis A \xRightarrow{Q} B \rrparenthesis_{R,\, X\mapsto(\bullet,k)} \qquad \text{if } X \notin Q
\end{aligned}$$

Casts $\boxed{\llparenthesis A \xRightarrow{p} B \rrparenthesis_R^{\mathsf{BK}}}$

$$\begin{aligned}
\llparenthesis \iota \xRightarrow{p} \iota \rrparenthesis_R &= \overline{\lambda} v{:}\iota.\, v \\
\llparenthesis X \xRightarrow{p} X \rrparenthesis_R &= \overline{\lambda} v{:}\mathtt{bits}.\, v \\
\llparenthesis \star \xRightarrow{p} \star \rrparenthesis_R &= \overline{\lambda} v{:}\mathtt{bits}.\, v
\end{aligned}
\qquad
\begin{aligned}
\llparenthesis G \xRightarrow{p} \star \rrparenthesis_R &= \overline{\lambda} v{:}\llparenthesis G \rrparenthesis.\, \lfloor v \rfloor_k && \text{if } k = \llparenthesis G \rrparenthesis_R \\
\llparenthesis \star \xRightarrow{p} G \rrparenthesis_R &= \overline{\lambda} v{:}\mathtt{bits}.\, \lceil v \rceil_k^p && \text{if } k = \llparenthesis G \rrparenthesis_R \\
\llparenthesis A \xRightarrow{p} \star \rrparenthesis_R &= \overline{\lambda} v{:}\llparenthesis A \rrparenthesis.\, (v \mathbin{@} \llparenthesis A \xRightarrow{p} G \rrparenthesis_R) \mathbin{@} \llparenthesis G \xRightarrow{p} \star \rrparenthesis_R && \text{if } \mathsf{ug}(A), A \prec G \\
\llparenthesis \star \xRightarrow{p} A \rrparenthesis_R &= \overline{\lambda} v{:}\mathtt{bits}.\, (v \mathbin{@} \llparenthesis \star \xRightarrow{p} G \rrparenthesis_R) \mathbin{@} \llparenthesis G \xRightarrow{p} A \rrparenthesis_R && \text{if } \mathsf{ug}(A), G \prec A
\end{aligned}$$

$$\begin{aligned}
\llparenthesis A \to B \xRightarrow{p} C \to D \rrparenthesis_R &= \overline{\lambda} v{:}\llparenthesis A \to B \rrparenthesis.\, \overline{\lambda} w{:}\llparenthesis C \rrparenthesis.\, (v\, (w \mathbin{@} \llparenthesis C \xRightarrow{-p} A \rrparenthesis_R)) \mathbin{@} \llparenthesis B \xRightarrow{p} D \rrparenthesis_R \\
\llparenthesis A \xRightarrow{p} \forall X.B \rrparenthesis_R &= \overline{\lambda} v{:}\llparenthesis A \rrparenthesis.\, \lambda k{:}\mathtt{key}\langle\mathtt{bits}\rangle.\, v \mathbin{@} \llparenthesis A \xRightarrow{p} B \rrparenthesis_{R,\, X\mapsto(\bullet,k)} \\
\llparenthesis \forall X.A \xRightarrow{p} B \rrparenthesis_R &= \overline{\lambda} v{:}\mathtt{key}\langle\mathtt{bits}\rangle{\to}\llparenthesis A \rrparenthesis.\, \mathtt{let}\, j = \mathtt{new}\langle\mathtt{bits}\rangle\ \mathtt{in}\ \mathtt{let}\, k = \mathtt{new}\langle\mathtt{bits}\rangle\ \mathtt{in} \\
&\qquad ((v\, k) \mathbin{@} \llparenthesis A \xRightarrow{+X} A[X{:=}\star] \rrparenthesis_{R, X\mapsto(j,k)}) \mathbin{@} \llparenthesis A[X{:=}\star] \xRightarrow{p} B \rrparenthesis_R
\end{aligned}$$

Figure 5. Translation from λB to λK

3. Cryptographic lambda calculus, λK

Figure 4 presents the cryptographic lambda calculus, λK.

Our calculus is inspired by Pierce and Sumii (2000), but makes some small modifications for ease of comparison with our system. We restrict the body of a type abstraction to be a value, and a failed decryption raises blame on a given label rather than invoking arbitrary code.

The calculus modifies the simply-typed lambda calculus by adding a type $\mathtt{key}\langle A\rangle$ of cryptographic keys, and a type `bits` of encoded values, and operations to allocate a new key, encode a value under a key, and decode a value under a key. An attempt to decode a value with the wrong key yields blame.

Let A, B, C, D range over types, which are either base type ι, function type $A \to B$, key type $\mathtt{key}\langle A\rangle$ that encodes or decodes values of type A, and the type `bits` of encoded values.

Let κ range over key constants. Let L, M, N range over terms, which are either a constant, an operator, a function abstraction, a function application, or blame, as before, or a key constant, or terms to allocate a fresh key or encrypt or decrypt a value by a given key.

Types $\boxed{(\!|A|\!)^{\mathsf{KB}}}$

$$(\!|\iota|\!) = \iota \qquad (\!|A \to B|\!) = (\!|A|\!) \to (\!|B|\!) \qquad (\!|\mathtt{key}\langle A\rangle|\!) = \exists X.((\!|A|\!) \to X) \times (X \to (\!|A|\!)) \qquad (\!|\mathtt{bits}|\!) = \star$$

Terms $\boxed{(\!|M|\!)^{\mathsf{KB}}}$

$$\begin{aligned} (\!|c|\!) &= c & (\!|op(\vec{M})|\!) &= op((\!|\vec{M}'|\!)) \\ (\!|x|\!) &= x & (\!|\mathtt{blame}\,p|\!) &= \mathtt{blame}\,p \\ (\!|\lambda x{:}A.\,N|\!) &= \lambda x : (\!|A|\!).\,(\!|N|\!) & (\!|L\,M|\!) &= (\!|L|\!)\,(\!|M|\!) \end{aligned}$$

$$\begin{aligned} (\!|\mathtt{new}\langle A\rangle|\!) &= \nu X{:=}(\!|A|\!).\mathtt{pack}\,(X, (\lambda v.\,v : (\!|A|\!) \xRightarrow{-X} X, \lambda w.\,w : X \xRightarrow{+X} (\!|A|\!)))\;\mathtt{as}\;\exists X.((\!|A|\!){\to}X) \times (X{\to}(\!|A|\!)) \\ (\!|\lfloor M\rfloor_N|\!) &= \mathtt{let}\,v = (\!|M|\!)\;\mathtt{in}\;\mathtt{unpack}\,(X,y) = (\!|N|\!)\;\mathtt{in}\;\mathtt{let}\,(f,g) = y\;\mathtt{in}\,(f\,v) : X \xRightarrow{\bullet} \star \\ (\!|\lceil L\rceil^p_N|\!) &= \mathtt{let}\,w = (\!|L|\!)\;\mathtt{in}\;\mathtt{unpack}\,(X,y) = (\!|N|\!)\;\mathtt{in}\;\mathtt{let}\,(f,g) = y\;\mathtt{in}\;g\,(w : \star \xRightarrow{p} X) \end{aligned}$$

Figure 6. Translation from λK to λB

Let Γ range over type contexts, which associate variables and key constants with types, and let Δ range over key stores, which are type contexts that bind only key constants. Unlike the polymorphic blame calculus, where types later in the store may contain names declared earlier in the store, the types in a key store are independent of the keys declared in the store.

Write $\vdash A : \mathtt{tp}$ to indicate that A is a valid type, and $\Gamma \vdash M : A$ to indicate that M is a term of type A under context Γ. The formation rules for contexts and types are straightforward and omitted. The formation rules for constant, operator, function abstraction, function application, and blame are as before, and are also omitted. We give formation rules for the three new term forms. Term $\mathtt{new}\langle A\rangle$ allocates a fresh key of type A, term $\lfloor M\rfloor_N$ encrypts M of type A by key N of type $\mathtt{key}\langle A\rangle$ yielding a value of type $\mathtt{bits}$, and term $\lceil L\rceil^p_N$ decrypts L of type $\mathtt{bits}$ by key N of type $\mathtt{key}\langle A\rangle$ yielding a value of type A or possibly failing and blaming p.

Let V, W range over values, which are constants, functions, key constants, or encoded values. Write $M \longmapsto N$ for a single reduction step. Reduction for operators and function application is standard. Encrypting and decrypting on the same key is the identity, while encrypting and decrypting on different keys yields blame.

Write $\Delta \triangleright M \longrightarrow \Delta' \triangleright M'$ for reduction of configurations. Reduction is closed under evaluation contexts, evaluating blame terminates reduction, and evaluating a key generator adds a fresh key to the configuration, returning that key.

The canonical forms and type safety results are straightforward.

Lemma 6 (Canonical forms)**.** *If $V : A$ then either*

- $V = c'$ *and* $A = \iota'$
- $V = \lambda x : A'.\,N'$ *and* $A = A' \to B'$
- $V = \kappa'$ *and* $A = \mathtt{key}\langle A'\rangle$
- $V = \lfloor V'\rfloor_{\kappa'}$ *and* $A = \mathtt{bits}$

where all primed variables are existentially quantified.

Proposition 7 (Type safety)**.**

1. *If* $\Delta \vdash M : A$ *and* $\Delta \triangleright M \longrightarrow \Delta' \triangleright M'$ *then* $\Delta' \vdash M' : A$.
2. *If* $\Delta \triangleright M : A$ *then either*
   - $M = V'$
   - $M = \mathtt{blame}\,p'$
   - $\Sigma \triangleright M \longrightarrow \Sigma' \triangleright M'$

   where all primed variables are existentially quantified.

Unlike λB, the type system of λK does not provide guarantees regarding the absence of blame. It ensures that a matching encode and decode have keys of the same type, but not the same key.

3.1 Translation of λB to λK

Figure 5 presents the translation from polymorphic blame calculus λB to cryptographic lambda calculus λK.

We extend λK with postfix application and let bindings:

$$\begin{aligned} M \mathbin{\mathtt{@}} L &\stackrel{\text{def}}{=} L\,M \\ \mathtt{let}\,x = M\;\mathtt{in}\,N &\stackrel{\text{def}}{=} (\lambda x{:}A.\,N)\,M \qquad \text{where } M : A \end{aligned}$$

Write $(\!|A|\!)$ for the translation of types. A base type translates to itself. A function type $A \to B$ translates homomorphically to $(\!|A|\!) \to (\!|B|\!)$. A universal type $\forall X.B$ translates to a function $\mathtt{key}\langle\mathtt{bits}\rangle \to (\!|B|\!)$, which accepts a key (used for performing casts in the body of the type abstraction), and returns a value of type $(\!|B|\!)$. Every name translates to the type $\mathtt{bits}$ of encrypted values. The dynamic type $\star$ also translates to the type $\mathtt{bits}$.

The translation associates each name X in the source with a pair of variables j and k bound to keys in the target. If X is introducted by a generator, with $X{:=}A$, then $j : \mathtt{key}\langle(\!|A|\!)\rangle$ and $k : \mathtt{key}\langle\mathtt{bits}\rangle$ in the target. Key j is used for conversions from A to X and back, while key k is used for casts from X to $\star$ and back. The translation of terms, conversions, and casts is parameterised by a relation R consisting of associations $X \mapsto (j,k)$. If X is introduced by a type abstraction, with $X : \mathtt{tp}$, then it participates in no conversions, so we write $X \mapsto (\bullet, k)$.

To each ground type G we associate a key used for casts from G to $\star$ and back, written $(\!|G|\!)_R$, where R is a relation from names to pairs of keys as described above. We assume that at the top level we have allocated keys

$$\kappa_\iota : \mathtt{key}\langle\iota\rangle \qquad \text{and} \qquad \kappa_{\star\to\star} : \mathtt{key}\langle\mathtt{bits} \to \mathtt{bits}\rangle$$

for this purpose. The only other ground type is a name, which is translated according to the relation R. We pick the second key of the pair, which is the one used for casts.

Write $(\!|M|\!)_R$ for the translation of terms, $(\!|A \Longrightarrow^Q B|\!)_R$ for the translation of conversions, and $(\!|A \Longrightarrow^p B|\!)_R$ for the translation of casts, where R is a relation from names to pairs of keys as described above. The translation inserts administrative λ's in the sense of Plotkin (1975), which are marked with an overline, as in $\overline{\lambda}$. The translation implicitly performs β-reduction whenever it produces a redex involving an adminstrative λ, which simplifies the simulation between λB and λK (Proposition 9).

Terms are translated as follows. Constants, operators, variables, function abstractions, function applications, and blame translate homomorphically. A type abstraction $\Lambda X.\,V$ translates to a function abstraction that accepts a key k and recursively translates the body V, where the relation R is extended by an association $X \mapsto (\bullet, k)$. The first key is written $\bullet$ because it is irrelevant (no conversions on X happen in the body of the abstraction), and the second is the parameter k passed to the function (used for casts on X in the body of the abstraction). Correspondingly, a type application $L\,X$ translates to an application of the translation of L to the key k, where the association $X \mapsto (j, k)$ occurs in relation R. A generator $\nu X{:=}A.N$ translates to a term that binds j and k to fresh keys of the appropriate type, and recursively translates the body N where the relation R is extended by $X \mapsto (j, k)$. Conversions and casts are translated by recursively translating the term and then applying the translation of the conversion or cast.

A conversion between name X and its corresponding type A is performed by encrypting or decrypting with the relevant key from the relation. The blame label for decryption is irrelevant, since a decryption corresponding to a conversion never fails. Most of the translation rules for conversions corresponds to the reduction rules for conversions. A conversion between two universal types translates to a term that accepts a function over keys and a key, applies the function to the key, and converts instances of the types at a fresh name. To preserve the invariant that every name mentioned in a term appears in the relation, the relation is extended by the fresh name, even though the key associated with it is not used in the translation. Indeed, in the translation of a conversion with label $Q = +X$ or $Q = -X$, although we pass the entire relation R (to maintain the correspondence with casts), the only relevant key is j in the association $X \mapsto (j, k) \in R$, since that is the only key required to translate the conversion.

A cast from G to $\star$ or from $\star$ to G translates as encoding or decoding with the appropriate key. Most of the translation rules for casts corresponds to the reduction rules for casts. A cast from A to $\forall X.B$ translates to a term that accepts a value v of type $(\!|A|\!)$ and a key k, and recursively casts v from A to B. For the cast, the relation R is extended by an association $X \mapsto (\bullet, k)$, where the key for conversions is written $\bullet$ because it is irrelevant, and k is the key for casts. A cast from $\forall X.A$ to B translates to a term that accepts a function over keys and allocates two fresh keys, then applies the function at the appropriate key and recursively converts from A to $A[X{:=}\star]$ and then casts $A[X{:=}\star]$ to B. For the conversion, the relation is extended by the association $X \mapsto (j, k)$, where j is the key for conversions and k is the key for casts (though the latter is unused in the conversion), while for the cast the relation is not extended (since the name X does not appear in its source or target).

We translate stores as follows. Let Σ be a store. For each $X_i{:=}A_i$ in Σ where R contains the association $X_i \mapsto (j_i, k_i)$, let Δ have entries $j_i : \texttt{key}\langle(\!|A_i|\!)\rangle, k_i : \texttt{key}\langle\texttt{bits}\rangle$. Define $(\!|\Sigma|\!)_R = \Delta$. We write $\Sigma \sim R$ if for every association $X{:=}A$ in Σ there is an association $X \mapsto (j, k)$ in R.

The translation preserves types and is a simulation.

Proposition 8 (Type preservation, λB to λK)**.**
If $\Sigma \vdash M : A$ and $\Sigma \sim R$ then $(\!|\Sigma|\!)_R \vdash (\!|M|\!)_R : (\!|A|\!)$.

Proposition 9 (Simulation, λB to λK)**.**
Assume $\Sigma \vdash M : A$ and $\Sigma \sim R$. Then

1. *If $M = V$ then $(\!|M|\!)_R$ is a value.*
2. *If $\Sigma \triangleright M \longrightarrow_{\mathsf{B}} \Sigma' \triangleright M'$ then $(\!|\Sigma|\!)_R \triangleright (\!|M|\!)_R \longrightarrow^*_{\mathsf{K}} (\!|\Sigma'|\!)_{R'} \triangleright (\!|M'|\!)_{R'}$, for some R'.*

3.2 Translation of λK to λB

Figure 6 presents the translation from cryptographic lambda calculus λK to polymorphic blame calculus λB. We extend λB with products and existentials by the standard encodings:

$$\begin{aligned}
A \times B &\overset{\text{def}}{=} \forall Z.(A \to B \to Z) \to Z \\
(V, W) &\overset{\text{def}}{=} \Lambda Z.\, \lambda h : A{\to}B{\to}Z.\, z\, V\, W \\
\texttt{let}\,(x, y) = L\,\texttt{in}\,N &\overset{\text{def}}{=} L\,C\,(\lambda x : A.\, \lambda y : B.\, N) \\
&\qquad \text{if } L : A \times B \text{ and } N : C
\end{aligned}$$

$$\begin{aligned}
\exists X.B &\overset{\text{def}}{=} \forall Z.(\forall X.B \to Z) \to Z \\
\texttt{pack}\,(A, V)\,\texttt{as}\,\exists X.B &\overset{\text{def}}{=} \Lambda Z.\, \lambda h : \forall X.B{\to}Z.\, h\, A\, V \\
\texttt{unpack}\,(X, y) = L\,\texttt{in}\,N &\overset{\text{def}}{=} L\,C\,(\Lambda X.\, \lambda y : B.\, N) \\
&\qquad \text{if } L : \exists X.B \text{ and } N : C
\end{aligned}$$

We write $(\!|A|\!)$ for the translation of types. A base type translates to itself. A function type $A \to B$ translates homomorphically to $(\!|A|\!) \to (\!|B|\!)$. Type $\texttt{key}\langle A\rangle$ translates to the existential type

$$\exists X.((\!|A|\!) \to X) \times (X \to (\!|A|\!))$$

which corresponds to functions that encrypt and decrypt values of type A by converting them to existentially quantified type X. Type $\texttt{bits}$ translates to the dynamic type $\star$.

We write $(\!|M|\!)$ for the translation of terms. Constant, operator, variable, blame, function abstraction, and function application translate homomorphically. Term $\texttt{new}\langle A\rangle$ translates to a value of the existential type, which generates a fresh X associated with type A, and uses conversions from A to X and X to A for the encrypting and decrypting functions. Term $\lfloor M \rfloor_N$ unpacks the existential type X corresponding to the key, and combines the encrypting function from A to X with a cast from X to $\star$. Conversely, term $\lceil L \rceil^p_N$ unpacks the existential type X corresponding to the key, and combines a cast from $\star$ to X with the decrypting function from X to A. The encoding cast can never fail, so its blame label is irrelevant, while the decoding cast takes its blame label from the decryption term.

We might imagine an alternative design that avoids the use of existential types:

$$\begin{aligned}
(\!|\texttt{key}\langle A\rangle|\!) &= ((\!|A|\!) \to \star) \times (\star \to (\!|A|\!)) \\
(\!|\texttt{new}\langle A\rangle|\!) &= \nu X{:=}(\!|A|\!).(\, \lambda v.\ v : A \overset{-X}{\Longrightarrow} X \overset{\bullet}{\Longrightarrow} \star, \\
&\qquad\qquad \lambda w.\, w : \star \overset{q}{\Longrightarrow} X \overset{+X}{\Longrightarrow} A\,) \\
(\!|\lfloor M \rfloor_N|\!) &= \texttt{let}\ v = (\!|M|\!)\ \texttt{in}\,\texttt{let}\,(f, g) = (\!|N|\!)\ \texttt{in}\ f\, v \\
(\!|\lceil M \rceil^p_N|\!) &= \texttt{let}\ w = (\!|M|\!)\ \texttt{in}\,\texttt{let}\,(f, g) = (\!|N|\!)\ \texttt{in}\ g\, w
\end{aligned}$$

However, this design is not quite right: if a decryption fails it blames label q, which appears in the translation of key allocation, rather than the label p that appears on the decryption term.

We translate key stores as follows. Let Δ be a key store. For each $\kappa_i : \texttt{key}\langle A_i\rangle$ in Δ where $\nu X_i{:=}(\!|A_i|\!).N_{\mathsf{B}i} = (\!|\texttt{new}\langle A_i\rangle|\!)$, let Σ be the store consisting of all the associations $X_i{:=}(\!|A_i|\!)$, and let σ be a substitution consisting of all the replacements $\kappa_i \mapsto N_{\mathsf{B}i}$. Define $(\!|\Delta|\!) = (\Sigma, \sigma)$.

The translation preserves types and is a simulation.

Proposition 10 (Type preservation, λK to λB)**.**
If $\Delta \triangleright M : A$ and $(\!|\Delta|\!) = (\Sigma, \sigma)$, then $\Sigma \triangleright (\!|M|\!)\sigma : (\!|A|\!)$.

Proposition 11 (Simulation, λK to λB)**.**
Assume $\Delta \triangleright M : A$ and $(\!|\Delta|\!) = (\Sigma, \sigma)$. Then

1. *If $M = V$ then $\Sigma \triangleright (\!|M|\!)\sigma \longrightarrow^*_{\mathsf{B}} \Sigma \triangleright W$, for some W.*
2. *If $\Delta \triangleright M \longrightarrow_{\mathsf{K}} \Delta' \triangleright M'$ then $\Sigma \triangleright (\!|M|\!)\sigma \longrightarrow^*_{\mathsf{B}} \Sigma' \triangleright (\!|M'|\!)\sigma'$, where $(\!|\Delta'|\!) = (\Sigma', \sigma')$.*

Polymorphic lambda calculus, $\lambda\mathsf{F}$

$$A, B ::= \iota \mid A \to B \mid \forall X.B \mid X$$

$$L, M, N ::= c \mid op(\vec{M}) \mid x \mid \lambda x{:}A.\, N \mid L\, M \mid \Lambda X.\, V \mid L\, A$$

Translation of $\lambda\mathsf{F}$ to $\lambda\mathsf{G}$ $\boxed{(\!|M|\!)^{\mathsf{FG}}}$

$$(\!|c|\!) = c \qquad (\!|\lambda x : A.\, N|\!) = \lambda x : A.\, (\!|N|\!)$$
$$(\!|op(\vec{M})|\!) = op((\!|\vec{M}|\!)) \qquad (\!|L\, M|\!) = (\!|L|\!)\, (\!|M|\!)$$
$$(\!|x|\!) = x \qquad (\!|\Lambda X.\, V|\!) = \Lambda X.\, (\!|V|\!)$$
$$(\!|L\, A|\!) = \nu X{:=}A.((\!|L|\!)\, X : B \overset{+X}{\Longrightarrow} B[X{:=}A])$$
$$\text{where } L : \forall X.B$$

Polymorphic lambda calculus with generativity, $\lambda\mathsf{G}$

$$L, M, N ::= c \mid op(\vec{M}) \mid x \mid \lambda x{:}A.\, N \mid L\, M \mid \Lambda X.\, V \mid L\, X \mid \nu X{:=}A.N \mid M : A \overset{Q}{\Longrightarrow} B$$

Translation of $\lambda\mathsf{G}$ to $\lambda\mathsf{F}$ $\boxed{(\!|M|\!)^{\mathsf{GF}}}$

$$(\!|c|\!) = c \qquad (\!|\lambda x : A.\, N|\!) = \lambda x : A.\, (\!|N|\!)$$
$$(\!|op(\vec{M})|\!) = op((\!|\vec{M}|\!)) \qquad (\!|L\, M|\!) = (\!|L|\!)\, (\!|M|\!)$$
$$(\!|x|\!) = x \qquad (\!|\Lambda X.\, V|\!) = \Lambda X.\, (\!|V|\!)$$
$$(\!|L\, X|\!) = (\!|L|\!)\, X$$
$$(\!|\nu X{:=}C.A|\!) = (\!|N|\!)[X{:=}A]$$
$$(\!|M : A \overset{Q}{\Longrightarrow} B|\!) = (\!|M|\!)$$

Figure 7. Polymorphic lambda calculus $\lambda\mathsf{F}$, and polymorphic lambda calculus with generativity $\lambda\mathsf{G}$

4. Polymorphic λ-calculus, $\lambda\mathsf{F}$, and polymorphic λ-calculus with generativity, $\lambda\mathsf{G}$

Figure 7 presents the polymorphic lambda calculus, $\lambda\mathsf{F}$, and the polymorphic lambda calculus with runtime type generation, $\lambda\mathsf{G}$, and translations between them.

Polymorphic lambda calculus, $\lambda\mathsf{F}$ consists of constants, operators, variables, function abstraction, function application, type abstraction, and type application. Type abstraction is restricted so the body is a value. We write $M \longrightarrow_{\mathsf{F}} M'$ for reduction. The reduction rules for constants and functions are as in $\lambda\mathsf{B}$ and $\lambda\mathsf{K}$, and the reduction rule for for type abstractions is standard:

$$(\Lambda X.\, V)\, A \longrightarrow_{\mathsf{F}} V[X{:=}A]$$

Polymorphic lambda calculus with runtime type generation, $\lambda\mathsf{G}$, is the subset of blame calculus $\lambda\mathsf{B}$ without casts or the dynamic type. We write $\Sigma \rhd M \longrightarrow_{\mathsf{G}} \Sigma' \rhd M'$ for reduction. Its reduction rules are those of $\lambda\mathsf{B}$, minus those that deal with casts or the dynamic type.

We write $(\!|M|\!)^{\mathsf{FG}}$ for the translation from a term of $\lambda\mathsf{F}$ to a term of $\lambda\mathsf{G}$. The translation is homomorphic save for type application, which introduces a generator and a conversion.

We write $(\!|M|\!)^{\mathsf{GF}}$ for the translation from a term of $\lambda\mathsf{G}$ to a term of $\lambda\mathsf{F}$. The translation is homomorphic save for generators, which substitute the associated type for the name, and conversions, which are elided. We extend the translation to configurations, by taking $(\!|\Sigma \rhd M|\!)^{\mathsf{GF}} = (\!|M|\!)^{\mathsf{GF}}\Sigma$, applying store Σ as a substitution.

The two translation are inverses.

Lemma 12 (Inverses). $(\!|(\!|M|\!)^{\mathsf{FG}}|\!)^{\mathsf{GF}} = M$.

The translation from $\lambda\mathsf{G}$ to $\lambda\mathsf{F}$ is a bisimulation.

Proposition 13 (Bisimulation, $\lambda\mathsf{G}$ to $\lambda\mathsf{F}$).
Assume $\Sigma \rhd N : A$ *and* $(\!|\Sigma \rhd N|\!)^{\mathsf{GF}} = M$.

- *If* $N =_{\mathsf{G}} W$ *then* $M =_{\mathsf{F}} V$, *for some* V.
- *If* $M =_{\mathsf{F}} V$ *then* $N =_{\mathsf{G}} W$, *for some* W.
- *If* $\Sigma \rhd N \longrightarrow_{\mathsf{G}} \Sigma' \rhd N'$ *then* $M \longrightarrow_{\mathsf{F}}^{?} M'$ *and* $(\!|\Sigma' \rhd N'|\!)^{\mathsf{GF}} =_{\mathsf{F}} M'$, *for some* M'.
- *If* $M \longrightarrow_{\mathsf{F}} M'$ *then* $\Sigma \rhd N \longrightarrow_{\mathsf{G}}^{*} \Sigma' \rhd N'$ *and* $(\!|\Sigma' \rhd N'|\!)^{\mathsf{GF}} =_{\mathsf{F}} M'$, *for some* Σ', N'.

The proof of this proposition is in the supplemental material.

We say M and N are *contextually equivalent*, and write $M \overset{\mathsf{ctx}}{=}_{\mathsf{F}} N$, if for every context $\mathcal{C}$ we have $\mathcal{C}[M] \longrightarrow_{\mathsf{F}}^{*} c$ if and only if $\mathcal{C}[N] \longrightarrow_{\mathsf{F}}^{*} c$. Similarly, mutatis mutandis, for $M \overset{\mathsf{ctx}}{=}_{\mathsf{G}} N$.

The translation from $\lambda\mathsf{G}$ to $\lambda\mathsf{F}$ is fully abstract.

Proposition 14 (Full abstraction, $\lambda\mathsf{G}$ to $\lambda\mathsf{F}$). *Assume* $M : A$ *and* $N : A$ *in* $\lambda\mathsf{G}$. *Then* $M \overset{\mathsf{ctx}}{=}_{\mathsf{G}} N$ *iff* $(\!|M|\!)^{\mathsf{GF}} \overset{\mathsf{ctx}}{=}_{\mathsf{F}} (\!|N|\!)^{\mathsf{GF}}$.

The proof of this proposition is in the supplemental material.

It follows immediately from Lemma 12 that the inverse translation is also fully abstract.

Proposition 15 (Full abstraction, $\lambda\mathsf{F}$ to $\lambda\mathsf{G}$). *Assume* $M : A$ *and* $N : A$ *in* $\lambda\mathsf{F}$. *Then* $M \overset{\mathsf{ctx}}{=}_{\mathsf{F}} N$ *iff* $(\!|M|\!)^{\mathsf{FG}} \overset{\mathsf{ctx}}{=}_{\mathsf{G}} (\!|N|\!)^{\mathsf{FG}}$.

5. Translation of $\lambda\mathsf{F}$ to $\lambda\mathsf{K}$

Figure 8 presents the translation from $\lambda\mathsf{F}$ to $\lambda\mathsf{K}$ given by Pierce and Sumii (2000) (their Figures 4 and 5). Since we restrict the body of type abstractions to be values, our version of their translation is slightly simpler, in that we do not need to introduce function abstractions with a dummy argument when translating type abstractions. We have adjusted their notation slightly to better correspond to our own: we write $\mathcal{T}^{\ell}(M)$ where they write $\mathcal{T}(M)$ for the top-level translation of programs from $\lambda\mathsf{F}$ to $\lambda\mathsf{K}$. We write $v \mathbin{@} \mathcal{C}_k^{\pm X,\ell}(A)$ where they write $\mathcal{C}_X^{\pm}(v, k, A)$ to encode each type variable X into `bits` and to encrypt/decrypt values of type X. We write $v \mathbin{@} \mathcal{G}^{\pm\ell}(A)$ where they write $\mathcal{G}^{\pm}(v, A)$ to guard parametricity via encryption/decryption. In each case we have added a primitive label ℓ for reporting blame as an additional parameter.

Define the following mapping from types of $\lambda\mathsf{F}$ to types of $\lambda\mathsf{B}$.

$$|\iota| = \iota \qquad |\forall X.B| = |B|$$
$$|A \to B| = |A| \to |B| \qquad |X| = \star$$

Let M be a term of $\lambda\mathsf{F}$, $(\!|M|\!)^{\mathsf{FG}}$ be the translation of terms of $\lambda\mathsf{F}$ to terms of $\lambda\mathsf{G}$ from Section 4, $(\!|M|\!)_R^{\mathsf{BK}}$ and $(\!|A \Longrightarrow^{p} B|\!)_R^{\mathsf{BK}}$ be the translations of terms and casts of $\lambda\mathsf{B}$ to $\lambda\mathsf{K}$ from Section 3.1, and $\overset{\mathsf{ctx}}{=}_{\mathsf{K}}$ be contextual equality on terms of $\lambda\mathsf{K}$, defined as usual. Then we conjecture that Pierce and Sumii's top-level translation from $\lambda\mathsf{F}$ to $\lambda\mathsf{K}$ is contextually equivalent to our translations as follows.

Conjecture 16 (Top-level translations $\mathcal{T}$).

$$\mathcal{T}^{\ell}(M) \overset{\mathsf{ctx}}{=}_{\mathsf{K}} (\!|(\!|M|\!)^{\mathsf{FG}}|\!)_R^{\mathsf{BK}} \mathbin{@} (\!|A \overset{+\ell}{\Longrightarrow} |A|\,|\!)_R^{\mathsf{BK}} \qquad \textit{if } R = \bullet$$

Indeed, each part of their translation relates directly to parts of our translation. Define forward composition:

$$f \mathbin{;} g \overset{\text{def}}{=} \lambda v.\, (v \mathbin{@} f) \mathbin{@} g$$

Then we have the following.

- $\mathcal{E}(A) = (\!|\,|A|\,|\!)^{\mathsf{BK}}$
- $\mathcal{E}(M)$ relates to $(\!|(\!|M|\!)^{\mathsf{FG}}|\!)^{\mathsf{BK}}$

Top level $\boxed{\mathcal{T}^\ell(M)}$

$$\mathcal{T}^\ell(M) = \mathcal{E}(M) \mathbin{@} \mathcal{G}^{+\ell}(A) \quad \text{where } M : A$$

Types $\boxed{\mathcal{E}(A)}$

$$\begin{aligned}
\mathcal{E}(\iota) &= \iota\\
\mathcal{E}(A \to B) &= \mathcal{E}(A) \to \mathcal{E}(B)\\
\mathcal{E}(\forall X.B) &= \mathcal{E}(B)\\
\mathcal{E}(X) &= \texttt{bits}
\end{aligned}$$

Terms $\boxed{\mathcal{E}(M)}$

$$\begin{aligned}
\mathcal{E}(c) &= c\\
\mathcal{E}(op(\vec{M})) &= op(\mathcal{E}(\vec{M}))\\
\mathcal{E}(x) &= x\\
\mathcal{E}(\lambda x{:}A.\,N) &= \lambda x{:}\mathcal{E}(A).\,\mathcal{E}(N)\\
\mathcal{E}(L\,M) &= \mathcal{E}(L)\,\mathcal{E}(M)\\
\mathcal{E}(\Lambda X.\,V) &= \mathcal{E}(V)\\
\mathcal{E}(L\,A) &= \texttt{let}\; k = \texttt{new}\langle\mathcal{E}(A)\rangle\;\texttt{in}\\
&\qquad \mathcal{E}(L) \mathbin{@} \mathcal{C}_k^{-X,\bullet}(B) \quad \text{where } L : \forall X.B
\end{aligned}$$

Conversions $\boxed{\mathcal{C}_k^{Q,\ell}(A)}$

$$\begin{aligned}
\mathcal{C}_k^{Q,\ell}(\iota) &= \lambda v.\,v\\
\mathcal{C}_k^{Q,\ell}(A \to B) &= \lambda v.\,\lambda w.\,v(w \mathbin{@} \mathcal{C}_k^{-Q,\ell}(A)) \mathbin{@} \mathcal{C}_k^{Q,\ell}(B)\\
\mathcal{C}_k^{Q,\ell}(\forall X.B) &= \lambda v.\,v \mathbin{@} \mathcal{C}_k^{Q,\ell}(B) \qquad \text{if } X \notin Q\\
\mathcal{C}_k^{+X,\ell}(X) &= \lambda v.\,\lfloor v \rfloor_k\\
\mathcal{C}_k^{-X,\ell}(X) &= \lambda v.\,\lceil v \rceil_k^{-\ell}\\
\mathcal{C}_k^{Q,\ell}(X) &= \lambda v.\,v \qquad \text{if } X \notin Q
\end{aligned}$$

Guards $\boxed{\mathcal{G}^p(A)}$

$$\begin{aligned}
\mathcal{G}^p(\iota) &= \lambda v.\,v\\
\mathcal{G}^p(A \to B) &= \lambda v.\,\lambda w.\,v(w \mathbin{@} \mathcal{G}^{-p}(A)) \mathbin{@} \mathcal{G}^p(B)\\
\mathcal{G}^{+\ell}(\forall X.B) &= \lambda v.\,v \mathbin{@} \mathcal{G}^{+\ell}(B)\\
\mathcal{G}^{-\ell}(\forall X.B) &= \lambda v.\,\texttt{let}\; k = \texttt{new}\langle\texttt{bits}\rangle\;\texttt{in}\\
&\qquad (v \mathbin{@} \mathcal{C}_k^{-X,\ell}(B)) \mathbin{@} \mathcal{G}^{-\ell}(B)\\
\mathcal{G}^p(X) &= \lambda v.\,v
\end{aligned}$$

Figure 8. Translation from λF to λK, after Pierce and Sumii (2000)

- $\mathcal{C}_j^{-X,\bullet}(B)$ relates to $(\!|B \overset{+X}{\Longrightarrow} B[X{:=}A]|\!)_R^{\mathsf{BK}}$, where $X \mapsto (j,k) \in R$
- $\mathcal{C}_j^{+X,\bullet}(B)$ relates to $(\!|B[X{:=}A] \overset{-X}{\Longrightarrow} B|\!)_R^{\mathsf{BK}}$, where $X \mapsto (j,k) \in R$
- $\mathcal{C}_{k_1}^{-X_1,\ell}(A) ; \cdots ; \mathcal{C}_{k_n}^{-X_n,\ell}(A) ; \mathcal{G}^{-\ell}(A)$ relates to $(\!|A \overset{+\ell}{\Longrightarrow} |A||\!)_R^{\mathsf{BK}}$ where $R = \{X_1 \mapsto (j_1,k_1), \ldots X_n \mapsto (j_n,k_n)\}$.
- $\mathcal{G}^{+\ell}(A) ; \mathcal{C}_{k_n}^{+X_n,\ell}(A) ; \cdots ; \mathcal{C}_{k_1}^{+X_1,\ell}(A)$ relates to $(\!||A| \overset{-\ell}{\Longrightarrow} A|\!)_R^{\mathsf{BK}}$ where $R = \{X_1 \mapsto (j_1,k_1), \ldots X_n \mapsto (j_n,k_n)\}$.

Note the reversal of sign! Their $\mathcal{C}$ labelled with $-X$ corresponds to conversions or casts labelled with $+X$, and conversely.

The comparison reveals a surprise. Occurrences of $\mathcal{C}_k^{\pm X,\bullet}(A)$ in $\mathcal{E}(M)$ corresponds to the translation of conversions, while occurrences of $\mathcal{C}_k^{\pm X,\ell}(A)$ in $\mathcal{G}^{\pm\ell}(M)$ corresponds to the translation of casts! Encryptions and decryptions in the former use keys of type $\texttt{key}\langle A\rangle$, and are guaranteed not to fail, while encryptions and decryptions in the latter use keys of type $\texttt{key}\langle\texttt{bits}\rangle$, and may fail.

6. Related Work

Polymorphic lambda calculus Girard and Reynolds independently discovered the polymorphic lambda calculus (Girard 1972; Reynolds 1974, 1983). Their calculus corresponds to second-order predicate calculus P2 via the Curry-Howard correspondence. Girard's representation theorem provides a projection from P2 into λF and Reynold's abstraction theorem provides an injection from λF into P2. Wadler (2007) proves that this pair of functions form an embedding-projection pair. Parametricity has many applications, e.g., providing "theorems for free" (Wadler 1989a), and it has broader connections to category theory (Hermida et al. 2014). Because of parametricity, λF can be implemented by erasure to the the untyped lambda calculus, as recently proved correct by Hou et al. (2015).

Cryptographic sealing The use of cryptographic 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 correspond to encryption and decryption, respectively. 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. Pierce and Sumii (2000) study a simply-typed λ-calculus with cryptographic sealing, upon which the λK language of this paper in based. They define a logical relation and prove parametricity for the language, and they present an embedding of polymorphic lambda calculus into their cryptographic lambda calculus which they conjecture is fully abstract. Our Section 5 describes the relation between their embedding and our translations from λF to λG and from λB to λK. Sumii and Pierce (2003) updates their calculus and logical relation result, and applies them to reason about cryptographic protocols. Sumii and Pierce (2004) study an untyped variant of the cryptographic lambda calculus and develop a bisimulation proof method for reasoning about contextual equivalence in the language.

Runtime type generation and conversions Variations on the generation and conversion constructs of λB have appeared many times in the literature. Grossman et al. (2000) use brackets to hide a host's knowledge that $X{=}A$ from a client; so their $[M]_h^B$ corresponds to our negative conversion $M : B[X{:=}A] \Longrightarrow^{-X} B$ and their $[M]_c^B$ corresponds to our positive conversion $M : B \Longrightarrow^{+X} B[X{:=}A]$. Rossberg (2003) uses generation and conversion to enforced type abstraction in a language with modules and dynamic linking. He generates type names with the construct $\mathrm{N}X \approx A.M$, which corresponds to our $\nu X{:=}A.M$, and uses conversions $\{M : B\}_{X\approx A}^{+}$ and $\{M : B\}_{X\approx A}^{-}$, which correspond to our negative and positive conversions, respectively. (Note that they are flipped!) Vytiniotis et al. (2005) apply runtime type generation to provide open and closed forms of ad-hoc polymorphism. Their construct new $X : \kappa = A$ in M corresponds to our type generation and their positive and negative conversions, $\{\!\{M : B\}\!\}_{X=A}^{+}$ and $\{\!\{M : B\}\!\}_{X=A}^{-}$, corresponds to our negative and positive conversions. (Again flipped.)

Sewell (2001) enforces type abstraction in the context of distributed computing.

Matthews and Ahmed (2008) use runtime type generation and boundaries to enforce parametricity in a multi-language setting, modeling interoperation between ML and Scheme. They also show how to implement boundaries in terms of runtime sealing. Their translation $E_A^{p;q}(\rho, M)$ corresponds to our $M \mathbin{@} \mathcal{G}'^p(A)_R$, where their parties p, q play a role similar to our blame label p, and their ρ plays a role similar to our R. Their translation differs from that of Pierce and Sumii (2000) and ours in that it generates seals for universals in both positive and negative situations.

Neis et al. (2009, 2011) study a language G with runtime type generation and a variant of Girard's J operator, which causes G to, in general, not be parametric. However, they show how to regain parametricity by wrapping universally-typed terms using runtime type generation. Our language λG is a subset of their G (minus Girard's J), as well as a subset of our λB (minus casts). The relative power of Girard's J and our casts is an open question and the subject of ongoing investigation.

Contracts and gradual typing Abadi et al. (1991) integrate dynamic typing into a statically-type language with a type named Dynamic. Findler and Felleisen (2002) introduce higher-order contracts and develop blame tracking to indicate which party is at fault when a contract is violated. Ou et al. (2004) combines simple types and dependent typing into a single language, with implicit coercions between the different regions. Flanagan (2006) introduces hybrid typing, which combines simple types with refinement types and inserts runtime checks whenever the automated theorem prover cannot prove that an implication corresponding to a subtype coercions is either true or false. Siek and Taha (2006, 2007) introduce gradual typing to enable fine-grained migration of code between static and dynamic typing disciplines, building on prior work of Thatte (1990) and Anderson and Drossopoulou (2003). Gronski et al. (2006) combines gradual typing with hybrid typing in the Sage language. Tobin-Hochstadt and Felleisen (2006) introduce coarse-grained (module-level) gradual typing and the use of blame tracking to characterizes the safe versus possibly unsafe casts. Wadler and Findler (2009) introduce the blame theorem for fine-grained gradual typing and show how to formulate the proof using a progress and preservation argument. Greenberg et al. (2010) studies the relationship between contracts and refinement types with casts (manifest contracts), extending the work of Gronski and Flanagan (2007) to handle dependent function contracts and dependent refinement types. Wadler (2015) surveys work on the blame calculus.

Polymorphism in contracts and gradual typing Guha et al. (2007) use dynamic sealing in the design of polymorphic contracts for PLT Scheme. Belo et al. (2011) and Greenberg (2013) study parametricity in the context of manifest contracts, which is quite different from the setting of this paper because their type system statically enforces parametricity. Ahmed et al. (2009, 2011) use runtime type generation to enforce parametricity in the polymorphic blame calculus, the predecessor of λB of this paper. Rastogi et al. (2015) erase polymorphic types and forbid their use in dynamic contexts, which sidesteps the issues addressed by λB.

7. Conclusion

We have introduced four calculi and described the relations between them: the polymorphic blame calculus, λB; the cryptographic lambda calculus, λK; the polymorphic lambda calculus, λF, and the polymorphic lambda calculus with seals, λG. The translations from λB to λK and back are simulations; while the translations from λF to λG and back are bisimulations and fully abstract. We relate our translations to the embedding of polymorphic lambda calculus into cryptographic lambda calculus of Pierce and Sumii (2000), revealing how conversions and casts offer insight into the structure of the translation.

Pierce and Sumii (2000) conjectured that their embedding is fully abstract, but their conjecture remains open after a decade and a half. Ahmed et al. (2011) claimed that their system satisfied the Jack-of-all-Trades theorem, but their proof turned out to have an error which they could not repair; and they conjectured that their system satisfies relational parametricity, but never published a proof. Perhaps the insights offered here may provide a step toward progress on these problems.

References

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

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

A. Ahmed, J. Matthews, R. B. Findler, and P. Wadler. Blame for all. In *Workshop on Script-to-Program Evolution (STOP)*, pages 1–13, 2009.

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

C. Anderson and S. Drossopoulou. BabyJ - from object based to class based programming via types. In *WOOD '03*, volume 82. Elsevier, 2003.

J. Belo, M. Greenberg, A. Igarashi, and B. Pierce. Polymorphic contracts. In G. Barthe, editor, *Programming Languages and Systems*, volume 6602 of *Lecture Notes in Computer Science*, pages 18–37. Springer Berlin Heidelberg, 2011.

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

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

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

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

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

J. Gronski and C. Flanagan. Unifying hybrid types and contracts. In *Trends in Functional Programming (TFP)*, Apr. 2007.

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

D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM Trans. Prog. Lang. Syst.*, 22(6):1037–1080, Nov. 2000.

A. Guha, J. Matthews, R. B. Findler, and S. Krishnamurthi. Relationally-parametric polymorphic contracts. In *Dynamic Languages Symposium (DLS)*, pages 29–40, 2007.

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

C. Hermida, U. S. Reddy, and E. P. Robinson. Logical relations and parametricity – a reynolds programme for category theory and programming languages. *Electronic Notes in Theoretical Computer Science*, 303(0): 149 – 180, 2014.

K.-B. Hou, N. Benton, and R. Harper. Correctness of compiling polymorphism to dynamic typing. submitted to JFP, February 2015.

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

J. H. Morris, Jr. Protection in programming languages. *Commun. ACM*, 16 (1):15–21, Jan. 1973.

G. Neis, D. Dreyer, and A. Rossberg. Non-parametric parametricity. In *International Conference on Functional Programming (ICFP)*, pages 135–148, Sept. 2009.

G. Neis, D. Dreyer, and A. Rossberg. Non-parametric parametricity. *Journal of Functional Programming*, 21:497–562, 2011.

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

B. Pierce and E. Sumii. Relating cryptography and polymorphism. Manuscript, 2000. URL `www.cis.upenn.edu/ bcpierce/papers/infohide.ps`.

A. M. Pitts. *Existential types: Logical relations and operational equivalence*, pages 309–326. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. ISBN 978-3-540-68681-1. . URL `http://dx.doi.org/10.1007/BFb0055063`.

A. M. Pitts. Structural recursion with locally scoped names. *Journal of Functional Programming*, 21(03):235–286, 2011.

A. M. Pitts and I. D. B. Stark. 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, pages 122–141, London, UK, UK, 1993. Springer-Verlag.

G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

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

J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *LNCS*, pages 408–425. Springer-Verlag, 1974.

J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523. North-Holland, 1983.

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

P. Sewell. Modules, abstract types, and distributed versioning. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 236–247, New York, NY, USA, 2001. ACM.

J. Siek, P. Thiemann, and P. Wadler. Blame and coercion: together again for the first time. In *Programming Language Design and Implementation (PLDI)*, pages 425–435, 2015.

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

J. G. Siek and W. Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming (ECOOP)*, 2007.

I. Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, April 1995.

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

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

E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 161–172, New York, NY, USA, 2004. ACM.

S. Thatte. Quasi-static typing. In *Principles of Programming Languages (POPL)*, pages 367–381, 1990.

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

D. Vytiniotis, G. Washburn, and S. Weirich. An open and shut typecase. In *Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '05, pages 13–24, New York, NY, USA, 2005. ACM.

P. Wadler. Theorems for free! In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359. ACM, 1989a.

P. Wadler. Theorems for free. In *Functional Programming Languages and Computer Architecture (FPCA)*, Sept. 1989b.

P. Wadler. The girard–reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1–3):201 – 226, 2007.

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

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

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