λdB: Blame tracking at higher fidelity

JAKUB ZALEWSKI, University of Edinburgh
JAMES MCKINNA, University of Edinburgh
J GARRETT MORRIS, University of Kansas
PHILIP WADLER, University of Edinburgh

This paper introduces λdB, a blame calculus with dependent types. It supports dependent functions, predicate refinement at all types, the dynamic type, and full blame tracking. It is inspired by and extends previous work on hybrid types and Sage, by Flanagan and others; manifest contracts, by Greenberg, Pierce, and Weirich; and blame calculus by Wadler and Findler. While previous work only allows refinement over base types, λdB supports refinement over any type. We introduce novel techniques in order to prove blame safety for this language, including a careful analysis that reduces open judgments on terms to closed ones on values, and the idea of 'subtyping with a witness', which fix flaws in the previous work of Wadler and Findler. These technical contributions mean that we can achieve a completely operational account of the metatheory of our language, and thereby avoid the need to intertwine operational and semantic models which bedevils the work on hybrid types and manifest contracts.

ACM Reference Format:
Jakub Zalewski, James McKinna, J Garrett Morris, and Philip Wadler. 2020. λdB: Blame tracking at higher fidelity. In *Informal Proceedings of the first ACM SIGPLAN Workshop on Gradual Typing (WGT20).* ACM, New York, NY, USA, 22 pages.

1 INTRODUCTION

Today half the research community is attempting to make typing more precise, via dependent types, while the other half is attempting to make typing less precise, via gradual types. Our concern here is with gradual dependent types, which aim to achieve both.

Our concerns are not merely academic. Developers are paying increasing attention to dependently-typed systems such as Coq, Agda, Idris, and $F^\star$, while vendors are rolling out gradually-typed languages such as Microsoft's TypeScript, Facebook's Hack, and Google's Dart, and established languages such as Racket, *C#*, and Python are adding features for gradual typing.

A long line of work addresses gradual dependent types.

Findler and Felleisen [2002] introduced contracts for higher-order programming languages. A flat contract ensures that a value satisfies a predicate, and a function contract ensures that its argument and result each satisfy a contract. They also introduced blame tracking, where blame can fall either on the term contained in the contract (positive blame) or the context containing the contract (negative blame). Extension of blame tracking to higher-order functions, where blame behaves covariantly on the range and contravariantly on the domain, was one of their key insights. They also consider dependent function contracts, where the contract for the result depends upon the value of the argument.

Tobin-Hochstadt and Felleisen [2006] and Matthews and Findler [2007] both apply contracts to integrate typed and untyped code, and both show blame safety: if a contract fails, blame must lie with the untyped code. Each requires a sophisticated proof based on operational equivalence.

Supported by EPSRC grants EP/L01503X/1 (CDT in Pervasive Parallelism) and EP/K034413/1 (ABCD: A Basis for Concurrency and Distribution).

WGT20, January 25, 2020, New Orleans
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.

Siek and Taha [2006] applied contracts to integrate static and dynamic typing, coining the phrase "gradual typing" to describe such systems.

Ou et al. [2004] applied contracts to integrate dependent typing with ordinary typing, performing dynamic checks to ensure an ordinary typed value conforms to the more precise dependent type. They required that the validity of every predicate in a refinement type can be decided at compile time.

Flanagan [2006] applied contracts to allow more flexible dependent types, called hybrid types. Rather than requiring that every predicate in a dependent type can be decided at compile time, they mix static verification and dynamic checking to support arbitrarily expressive predicates. Knowles et al. [2006] describes Sage, a language that supports both hybrid and gradual types. Flanagan [2006] supports subsumption, forcing the definitions of typing and subtyping judgements to be mutually dependent. Knowles and Flanagan [2010] observes that Flanagan [2006] is ill-defined, due to a typing judgement appearing to the left of an implication in a subtyping judgement, violating monotonicity, and resolve the problem by relying on a denotational semantics for types to ensure their definitions are well founded.

Greenberg et al. [2010, 2012] introduce the names *latent* and *manifest* to distinguish contracts from hybrid types, and observe there are two possible formulations of latent systems that they dub *lax* and *picky*. Like Knowles and Flanagan [2010] they support subsumption, and rely on a denotational semantics of types to ensure their definitions are well founded. They give translations between the lax and picky latent systems and the manifest system, showing that some translations can be exact while others will overapproximate.

Siek and Taha [2006], Ou et al. [2004], and Flanagan [2006] all feature a similar translation from a surface language to a core calculus with casts, where casts act as the analogue of contracts. None of them features blame tracking; in Siek and Taha [2006] and Ou et al. [2004] casts are unlabelled, while in Flanagan [2006] casts are labelled but there is no notion of positive and negative blame, which means they cannot pin a failure to one side of a cast. As a result, all three only characterise correctness globally: if all the casts are from subtypes to supertypes, then the program never fails. But this characterisation is too strict for the most common use case of all three systems, since if any dynamic check may fail then no formal guarantees apply.

Wadler and Findler [2009] defined a calculus similar to the core language of the three preceeding papers. They added blame tracking, and proved a more forgiving form of blame safety: if a cast fails, blame must lie with the less-precisely-typed side of the cast. As opposed to the global characterisation of the three preceeding papers, blame safety can be established on a cast-by-cast basis. Like Ou et al. [2004] and Flanagan [2006] they only permit refinement over base types, though they are even more restrictive in that they do not support dependent function types. Whereas Tobin-Hochstadt and Felleisen [2006] and Matthews and Findler [2007] provide sophisticated proofs of blame safety based on operational equivalence, Wadler and Findler [2009] provides a simple proof of blame safety based on preservation and progress.

Our goal here is to combine the dependent function types studied by Flanagan [2006] and Greenberg et al. [2010, 2012] with the blame tracking of Wadler and Findler [2009], allowing us to establish blame safety on a cast-by-cast basis while supporting dependent function types. Unlike these previous works, which only permit refinements over base types, we permit refinement over any type. Whereas Knowles and Flanagan [2010] and Greenberg et al. [2012] support subsumption and rely on a denotational semantics of types to ensure their definitions are well founded, we avoid subsumption, permitting us to disentangle the definitions of typing and subtyping judgements and to avoid reliance on a semantics of types.

Flanagan [2006] and Greenberg et al. [2010, 2012] use name-dependent typing, where types may depend upon arbitrary terms and call-by-name evaluation is used. Here we use value-dependant

typing, where types only depend upon values, and call-by-name evaluation is used. We follow the development standard in other works, such as [Swamy et al. 2013]. Value-dependent typing fixes the order of evaluation, facilitating reasoning about blame as an effect, as well as reasoning about other effects.

Like Wadler and Findler [2009] and Greenberg et al. [2012], we have no way of checking at compile-time that one refinement implies another, limiting such (potentially undecidable) judgements to the run-time type system. Also like Wadler and Findler [2009], all casts are explicit in the source language, and at run-time all values of refinement type are explicitly labeled. This gives us a simple system, suitable as a core calculus, which has a clean metatheory with properties such as unicity of types. In practice, one would probably also want a translation from a surface language into this core calculus, similar to those discussed by Ou et al. [2004], Siek and Taha [2006], and Flanagan [2006]. We believe such a translation would be the correct place to add support for subsumption and compile-time validation that a value satisfies a refinement, but we leave this for future work.

We have mentioned above the main influences on our development. We began this work several years ago; since then, there have been additional relevant works. Lehmann and Tanter [2017] introduces refinement types with support for unknown refinements, and again is restricted to refinement of base types. Tanter and Tabareau [2015] and Dagand et al. [2016, 2018] present an interoperability framework added as a library to Coq, and Eremondi et al. [2019] present a gradual type theory with both unknown values and unknown types. These latter systems do not support refinement types *per se*, but do support the related notion of sigma types and are not restricted to base types. None of these systems supports blame.

It turns out there are two flaws in the development of Wadler and Findler [2009], which we describe and correct here. First, their proof of blame safety is incorrect, as there are counter-examples to its claim that reduction preserves blame safety. Second, they fail to correctly define blame safety for open terms. We detail examples of these flaws in Section 2. The development given here avoids these flaws, by introducing a careful analysis that reduces open judgements on terms to closed judgements on values, and by a novel characterisation of blame safety. Whereas the earlier work uses a subtyping relation over two types, here we use a three-place relation between a closed value and two closed types, and a four-place relation between an open term and two open types in a given type environment; the relations are defined by mutual recursion. We dub these relations 'subtyping with a witness'.

This paper represents preliminary work. Many of the proofs have not been carried out in detail, and accordingly we often label results as conjectures rather than propositions.

Summary of contributions.

- Whereas previous gradual type systems support either dependent types or blame safety, we present a system that supports both.
- Whereas previous systems only support refinement over base types, we support refinement over any type.
- Whereas previous systems support name-dependent typing, we support value-dependent typing, which is better suited to programs with computational effects such as blame.
- Whereas previous systems support subsumption and require denotational semantics to break circularity in the definition, we eschew avoid subsumption and our semantics is purely operational.
- We reveal flaws in Wadler and Findler which are corrected in our system by introducing an analysis that reduces open judgements on terms to closed ones on values and novel three-place and four-place subtyping relations.

The outline of the remainder of this paper is as follows. Section 2 highlights the problem in the original formulation of Blame Theorem by [Wadler and Findler 2009]. Section 3 introduces λdB syntax, types, and reductions. Section 4 introduces our formulation of type safety for λdB. Section 5 introduces our formulation of subtyping and blame safety. Section 6 describes related work.

2 OVERVIEW

Let's consider the type of positive integers and the type of natural numbers. Adopting the syntax of Swamy et al. [2013], we define the type of positive numbers and the type of natural numbers as following:

$$Pos \stackrel{\text{def}}{=} (x : \mathtt{int})\{x > 0\}$$
$$Nat \stackrel{\text{def}}{=} (x : \mathtt{int})\{x \geq 0\}$$

We cast the integer 2 to the type *Pos* as follows:

$$2 : \mathtt{int} \stackrel{p}{\Rightarrow} Pos$$

At runtime, the cast evaluates the predicate $x > 0$ with x instantiated to 2, and it raises blame p if the predicate evaluates to false. In this case, the predicate evaluates to true, so the cast returns the following value:

$$2 : \mathtt{int} \Rrightarrow Pos$$

Here we write $\Rrightarrow$ in place of $\stackrel{p}{\Rightarrow}$; the blame label is omitted because the predicate is now verified and cannot fail. The value type-checks only if the predicate $x > 0$ evaluates to true when x is 2. (In Wadler and Findler [2009], the two terms above are written as $\langle Pos \Leftarrow^p \mathtt{int} \rangle\, 2$ and 2_{Pos}, respectively.)

Since predicates may be arbitrarily complicated, type checking for the $\Rrightarrow$ construct is undecidable. To deal with this issue, we partition our language into a compile-time subset with decidable type checking (which includes $\stackrel{p}{\Rightarrow}$) and a runtime superset with undecidable type checking (which adds $\Rrightarrow$). Undecidable type checking of the runtime language is not a serious issue, since the compile-time language is decidable, translation to the runtime language preserves typing, and reduction in the runtime language also preserves typing.

We may now take our value of type *Pos* and cast it to type *Nat*:

$$(2 : \mathtt{int} \Rrightarrow Pos) : Pos \stackrel{p}{\Rightarrow} Nat$$

We abbreviate the above as follows:

$$2 : \mathtt{int} \Rrightarrow Pos \stackrel{p}{\Rightarrow} Nat$$

In general, we often collapse sequences of casts (either $\Rrightarrow$ or $\Rightarrow$ or both) in this way. Again the cast succeeds, resulting in

$$2 : \mathtt{int} \Rrightarrow Nat$$

Indeed, any cast from type *Pos* to type *Nat* must succeed, since $x > 0$ implies $x \geq 0$ for any integer x. Again, this is undecidable, but this property is used for reasoning about programs rather than compiling.

In Wadler and Findler [2009], the fact that the cast always succeeds is indicated by writing:

$$Pos <: Nat$$

Indeed, they introduce four related notions:

$$A <: B \qquad A <:^{+} B \qquad A <:^{-} B \qquad A <:_n B$$

Consider a cast from A to B with label p.

- The first, ordinary subtyping, holds if the cast never raises blame.
- The second, positive subtyping, holds if the cast never raises blame p.
- The third, negative subtyping, holds if the cast never raises blame $-p$.
- The fourth, naive subtyping, holds if type A is more precise than type B.

A term is said to be *safe for p* if every subterm with a cast from A to B satisfies $A <:^{+} B$ if it has label p, and $A <:^{-} B$ if it has label $-p$. (In Wadler and Findler [2009], negative blame $-p$ is written as $\overline{p}$.)

To ensure the above properties, Wadler and Findler [2009] claim blame preservation: if a term is safe for p and it reduces to another term, then the new term is also safe for p. One of the reduction rules in that paper is (in the notation of this paper) as follows:

$$(V : A \Rrightarrow (x : A)\{P[x]\} \overset{p}{\Rightarrow} B) \longrightarrow (V : A \overset{p}{\Rightarrow} B)$$

In our case, this means we have the reduction:

$$(2 : \texttt{int} \Rrightarrow \mathit{Pos} \overset{p}{\Rightarrow} \mathit{Nat}) \longrightarrow (2 : \texttt{int} \overset{p}{\Rightarrow} \mathit{Nat})$$

But while $\mathit{Pos} <: \mathit{Nat}$ clearly holds, $\texttt{int} <: \mathit{Nat}$ clearly does not hold; casting any negative integer to the natural type will fail. So even though the reduction yields a term that won't raise blame, it does not yield a term that is safe for blame. The claim that reductions preserve blame safety is flawed.

Here we fix the claim by moving from a two-place relation

$$A <: B$$

where A and B are types, to a three-place relation

$$V : A <: B$$

where V is a value and A and B are types. Although $\texttt{int} <: \mathit{Nat}$ does not hold, it will turn out that $2 : \texttt{int} <: \mathit{Nat}$ does hold, allowing us to present a correct proof of blame safety.

Another issue with Wadler and Findler [2009] is that if one looks closely at the definition of the relations $<:$, $<:^{+}$, and $<:^{-}$ it becomes clear that they only make sense for closed types, while the notion of blame safety is defined on open types. In particular, that paper defines $A <: (x : B)\{P[x]\}$ to hold only if

$$\text{for every } V : A \text{ if } (V : A \overset{p}{\Rightarrow} B) \longrightarrow^{*} W \text{ then } P[W] \longrightarrow^{*} \texttt{true}$$

(see Figure 5 of that paper). But this definition does not consider any free variables other than x that might appear in P (or A or B). However, a lambda abstraction is deemed safe for p only when its body is safe for p (see Figure 7 of that paper), and the bound variable of the lambda abstraction appears free in its body.

Here we fix the issue by also defining a four-place relation

$$\Delta \vdash M : A <: B$$

that holds if and only if

$$\text{for every closing substitution } \eta \text{ of } \Delta, \text{ if } \eta(M) \longrightarrow^{*} V \text{ then } V : A <: B.$$

It will turn out that the three- and four-place relations are mutually recursive, as the four-place relation will prove instrumental in properly defining the three-place relation for function types.

Wadler and Findler [2009] only considered refinements over base types. This is because if refinements were permitted over function types then the subtyping relation defined in that paper

$$
\begin{array}{lrl}
\text{Base Types} & \iota & ::= \texttt{bool} \mid \texttt{int} \\
\text{Types} & A, B, C, D & ::= \iota \mid \bigstar \mid (x : A)\{P\} \mid (x : A) \to B \\
\text{Values} & V, W & ::= c \mid x \mid \lambda x{:}A.\ N \\
\text{Terms} & L, M, N, P, Q & ::= V \mid op(\vec{M}) \mid L\ V \mid \texttt{let}\ x = M\ \texttt{in}\ N \\
 & & \quad \mid \texttt{if}\ P\ \texttt{then}\ M\ \texttt{else}\ N \mid M : A \overset{p}{\Rightarrow} B \\
\text{Labels} & p, q & ::= +\ell \mid -\ell \\
\text{Environments} & \Gamma & ::= \cdot \mid \Gamma,\ x : A
\end{array}
$$

Fig. 1. Compile-time language syntax.

would not be transitive. The counter-example was discovered by Greenberg and Pierce (personal communication):

$$
\begin{array}{rcl}
Pos \to Pos & <:^{+} & Nat \to Nat \\
Nat \to Nat & <:^{+} & (f : \texttt{int} \to \texttt{int})\{f\ 0 \geq 0\} \\
Pos \to Pos & \not<:^{+} & (f : \texttt{int} \to \texttt{int})\{f\ 0 \geq 0\}
\end{array}
$$

The final subtyping relation does not hold, as the predicate will not hold, since $f\ 0$ always reduces to blame when f has type $Pos \to Pos$. It will turn out that our new definitions fix the problem, partly because we replace $P[V] \longrightarrow^{*} \texttt{true}$ by $P[V] \not\longrightarrow^{*} \texttt{false}$, and partly because our definition of closing substitution excludes from consideration terms in the environment that raise blame rather than reducing to a value.

We can now proceed with our formal development.

3 DEPENDENT BLAME CALCULUS (λdB)

We present dependently-typed blame calculus, λdB, which integrates dependently-typed code and simply-typed code using *casts*, and incorporates *refinements over arbitrary types* and *dependent function types*. The language is purely functional, with no mutable state and non-termination and raising blame as the only computational effects.

We present λdB factored into a compile-time language (Figure 1) with with decidable type-checking and a run-time language (Figure 3) with explicitly tagged values of refinement type where checking that a value satisfies a predicate may be undecidable. The run-time language is a strict superset of the compile-time language.

3.1 Compile-Time Language

Our language is influenced by λH of Knowles and Flanagan [2010], blame calculus of Wadler and Findler [2009], and value-dependency and syntax of refinements of Swamy et al. [2013].

Our language differs from λH with our treatment of casts. First and foremost, we keep the blame labels on casts in order to track blame and prove the blame theorem. Second, instead of relying on subsumption and automatic insertions of casts we forego subsumption and require all casts to be explicit. Such decision allows us to achieve decidable type-checking of the compile-time language and provide a simpler formalism, as we explain in Section 3.8.

We let ι range over base types, which are either integers or booleans.

We let A, B, C, D range over types: a type is either a base type ι, the dynamic type $\bigstar$, a refinement $(x : A)\{P\}$ of type A with a predicate P acting on x, or a dependent function $(x : A) \to B$, where x of type A is bound in B.

We let V, W range over values, L, M, N range over terms, and P, Q range over terms that occur as predicates.

A value is a constants c, variable x, or dependent lambda abstractions $\lambda x{:}A.\, N$. Since the language is call-by-value, a variable is always bound to a value and evaluating a variable can have no computational effect (such as raising blame).

A term is either a value, an application of a built-in operator $op(\vec{M})$, an application of a dependent function to a value LV, a non-dependent let binding $\texttt{let}x = M\,\texttt{in}N$, a conditional $\texttt{if}N\,\texttt{then}M\,\texttt{else}L$, or a cast $M : A \overset{p}{\Rightarrow} B$. Following the approach to value-dependency of Swamy et al. [2013], we restrict applications of functions to values.

We let p, q range over blame labels, and ℓ range over locations. A blame label is either positive $+\ell$ or negative $-\ell$. We define an involutive negation on blame labels $-p$ as follows

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

The mapping $|\cdot|$ from blame labels to locations is given by $|(-\ell)| = \ell = |(+\ell)|$.

We let Γ range over environments, which associate variables with types.

3.2 Type System for Compile-Time Language

Each constant c and built-in operator op has a type defined by $\mathrm{type}(c)$ and $\mathrm{type}(op)$. Each built-in operator op is specified by a total meaning function $[\![op]\!]$.

For variables, we define a context lookup relation, $\Gamma \ni x : A$, which is standard.

Lambda abstractions are dependently-typed and the argument is bound in the result type.

We restrict our applications to values as value-dependency is a well-understood technique to reason about side-effects such as raising blame [Swamy et al. 2013]; since the function argument x is bound in the result type B we substitute the argument V for x in the result type $B[x := V]$ (sometimes writing $B[V]$ where this is unambiguous).

We restrict our let-expressions to be non-dependent via the additional clause $\Gamma \vdash_{\mathrm{ct}} B : \mathrm{tp}$ which ensures that B does not depend on x. By restricting our function applications to be value-dependent and by restricting our let-expressions to be non-dependent we fix the order of evaluation on the term level and on the type level to standard call-by-value evaluation strategy which simplifies reasoning in our language.

We use standard conditional expressions for the compile-time language. We will extend let and conditional expressions in the run-time language of the next section.

Casts must be between compatible types A and B, which we write as $A \sim B$. We extend the standard notion of compatibility to include dependent functions. It is straightforward to show that compatibility is closed under value substitution. We further restrict our casts to be defined between well-formed types, via the additional clause $\Gamma \vdash_{\mathrm{ct}} B : \mathrm{tp}$.

3.3 Run-Time Language

We let Δ range over run-time environments. These include not only the variable bindings $x : A$ as in the compile-time environments, but also let bindings $x : A = M$, and predicate bindings P. These last two track terms bound by let-expressions and predicates tested by conditional expressions. Our environments are similar to those of Knowles et al. [2006], but we permit let bindings to arbitrary terms, whereas Knowles et al. restrict their bindings to values. Predicate bindings mirror those of Ou et al. [2004]. We let Ξ range over closed environments, which only contain let bindings and predicate bindings, and ρ, σ, η range over substitutions, which map variables to values. Our environments form the following hierarchy

$$\Gamma \subset \Delta \supset \Xi \supset \sigma.$$

$\boxed{\Gamma\ \mathtt{ctx}}$

$$\cdot\ \mathtt{ctx} \qquad \frac{\Gamma\ \mathtt{ctx} \qquad \Gamma \vdash_{\mathrm{ct}} A : \mathtt{tp}}{\Gamma, x : A\ \mathtt{ctx}}$$

$\boxed{\Gamma \ni x : A}$

$$\Gamma, x : A \ni x : A \qquad \frac{\Gamma \ni x : A \qquad x \neq y}{\Gamma, y : B \ni x : A}$$

$\boxed{\Gamma \vdash_{\mathrm{ct}} A : \mathtt{tp}}$

$$\Gamma \vdash_{\mathrm{ct}} \iota : \mathtt{tp} \qquad \Gamma \vdash_{\mathrm{ct}} \bigstar : \mathtt{tp} \qquad \frac{\Gamma \vdash_{\mathrm{ct}} A : \mathtt{tp} \qquad \Gamma, x : A \vdash_{\mathrm{ct}} P : \mathtt{bool}}{\Gamma \vdash_{\mathrm{ct}} (x : A)\{P\} : \mathtt{tp}}$$

$$\frac{\Gamma \vdash_{\mathrm{ct}} A : \mathtt{tp} \qquad \Gamma, x : A \vdash_{\mathrm{ct}} B : \mathtt{tp}}{\Gamma \vdash_{\mathrm{ct}} (x : A) \to B : \mathtt{tp}}$$

$\boxed{A \sim B}$

$$\iota \sim \iota \qquad \bigstar \sim B \qquad A \sim \bigstar \qquad \frac{A \sim B}{A \sim (x : B)\{P\}} \qquad \frac{A \sim B}{(x : A)\{P\} \sim B}$$

$$\frac{C \sim A \qquad B \sim D}{(x : A) \to B \sim (y : C) \to D}$$

$\boxed{\Gamma \vdash_{\mathrm{ct}} M : A}$

$$\frac{\mathrm{type}(c) = A}{\Gamma \vdash_{\mathrm{ct}} c : A} \qquad \frac{\Gamma \ni x : A}{\Gamma \vdash_{\mathrm{ct}} x : A} \qquad \frac{\mathrm{type}(op) = \vec{A} \to B \qquad \Gamma \vdash_{\mathrm{ct}} \vec{M} : \vec{A}}{\Gamma \vdash_{\mathrm{ct}} op(\vec{M}) : B}$$

$$\frac{\Gamma \vdash_{\mathrm{ct}} A : \mathtt{tp} \qquad \Gamma, x : A \vdash_{\mathrm{ct}} N : B}{\Gamma \vdash_{\mathrm{ct}} (\lambda x{:}A.\ N) : (x : A) \to B} \qquad \frac{\Gamma \vdash_{\mathrm{ct}} L : (x : A) \to B \qquad \Gamma \vdash_{\mathrm{ct}} V : A}{\Gamma \vdash_{\mathrm{ct}} L\ V : B[x := V]}$$

$$\frac{\Gamma \vdash_{\mathrm{ct}} M : A \qquad \Gamma, x : A \vdash_{\mathrm{ct}} N : B \qquad \Gamma \vdash_{\mathrm{ct}} B : \mathtt{tp}}{\Gamma \vdash_{\mathrm{ct}} (\mathtt{let}\ x = M\ \mathtt{in}\ N) : B}$$

$$\frac{\Gamma \vdash_{\mathrm{rt}} P : \mathtt{bool} \qquad \Gamma \vdash_{\mathrm{rt}} M : A \qquad \Gamma \vdash_{\mathrm{rt}} N : A}{\Gamma \vdash_{\mathrm{rt}} (\mathtt{if}\ P\ \mathtt{then}\ M\ \mathtt{else}\ N) : A} \qquad \frac{\Gamma \vdash_{\mathrm{ct}} M : A \qquad A \sim B \qquad \Gamma \vdash_{\mathrm{ct}} B : \mathtt{tp}}{\Gamma \vdash_{\mathrm{ct}} (M : A \overset{p}{\Rightarrow} B) : B}$$

Fig. 2. Compile-time language type system.

We include substitutions in the environment hierarchy, as we later show that we can evaluate closed environments to closing substitutions.

For the run-time language we extend the syntax with tagged dynamically-typed values $V : G \Rrightarrow \bigstar$, tagged dependently-typed values $V : A \Rrightarrow (x : A)\{P\}$, and blame terms $\mathtt{blame}\ p$. These three term forms may be introduced by reductions.

Ground Types	$G ::= \iota \mid \star \to \star$
Values	$V, W ::= \cdots \mid V : G \Rrightarrow \star \mid V : A \Rrightarrow (x : A)\{P\}$
Terms	$L, M, N, P, Q ::= \cdots \mid \texttt{blame}\ p$
Environments	$\Delta ::= \cdots \mid \Delta, x : A = M \mid \Delta, P$
Closed Environments	$\Xi ::= \cdot \mid \Xi, x : A = M \mid \Xi, P$
Substitutions	$\sigma, \rho ::= \cdot \mid \sigma, x = V$
Evaluation Frames	$\mathcal{E} ::= op(\vec{V}, \square, \vec{M}) \mid \square\ V \mid \texttt{let}\ x = \square\ \texttt{in}\ M$
	$\mid \texttt{if}\ \square\ \texttt{then}\ M\ \texttt{else}\ L \mid \square : A \overset{p}{\Rrightarrow} B$

Fig. 3. Run-time language syntax

The value $V : G \Rrightarrow \star$ represents the injection of a value V with type G into the dynamic type $\star$. The value $V : A \Rrightarrow (x : A)\{P\}$ represent a dependently-typed value with type $(x : A)\{P\}$, where the underlying value V has type A. We prohibit these two forms of values from appearing in compile-time programs. Prohibiting the latter ensures that the compile-time language benefits from decidable type-checking.

The blame expression $\texttt{blame}\ p$ results from a failed run-time cast with blame label p. We choose not to annotate blame expression with their types because it simplifies the reduction rules. It would be straightforward to make the other choice, and doing so would yield a stronger unicity result.

3.4 Run-Time Typing

Typing judgements for the run-time system are shown in Figure 4. The three occurrence of ellipses are to be replaced by corresponding rules from the compile-time system of Figure 2, save that occurrences of Γ are replaced by Δ.

The difference between let-expressions and dependent functions is also reflected in the structure of typing environment in our system: a λ abstraction extends the environment with a variable binding, while a let-expression extends the environment with a let binding. Conditionals are also tracked in the environment with predicate bindings. Environments with let bindings and predicate bindings will provide extra information useful when it comes to blame safety.

We extend the typing judgement with rules for tagged dynamically-typed values, tagged dependently-typed values, and blame expressions. We modify the typing rule for *let* expressions to track the binding via an environment extension with a term.

The typing rule for tagged dependently-typed values is similar to the one used by Wadler and Findler [2009], but we use a different notion of entailment, as described below. Both rules used closed environments Ξ, which is sound because these constructs only arise from reduction, and reduction always takes place on closed terms. One might expect one could simply use the empty environment for these type rules, but we shall see one of the reduction rules introduces a let-expression and a conditional on the right-hand side, hence the need for a closed environment rather than an empty environment.

For conditional expressions we adopt the 'Princeton' formulation of the typing rule [Ou et al. 2004], which explicitly tracks the predicate in Δ such that for the *then* branch we record that the predicate should hold, whereas for the *else* branch we record that the predicate should not hold. We use $\neg P$ to denote negation of a predicate, defined as

$$\neg P \quad \overset{\text{def}}{=} \quad \texttt{if}\ P\ \texttt{then}\ \texttt{false}\ \texttt{else}\ \texttt{true}.$$

$\boxed{\Delta\ \mathtt{ctx}}$

$$\cdots \qquad \frac{\Delta\ \mathtt{ctx} \qquad \Delta \vdash_{\mathrm{ct}} M : A}{\Delta, x : A = M\ \mathtt{ctx}} \qquad \frac{\Delta\ \mathtt{ctx} \qquad \Delta \vdash_{\mathrm{rt}} P : \mathtt{bool}}{\Delta, P\ \mathtt{ctx}}$$

$\boxed{\Delta \ni x : A}$

$$\cdots \qquad \Delta, x : A = M \ni x : A \qquad \frac{\Gamma \ni x : A \qquad x \neq y}{\Gamma, y : A = M \ni x : A} \qquad \frac{\Delta \ni x : A}{\Delta, P \ni x : A}$$

$\boxed{\Delta \vdash_{\mathrm{rt}} M : A}$

$$\cdots \qquad \frac{\Delta \vdash_{\mathrm{rt}} M : A \qquad \Delta, x : A = M \vdash_{\mathrm{ct}} N : B \qquad \Delta \vdash_{\mathrm{rt}} B : \mathtt{tp}}{\Delta \vdash_{\mathrm{rt}} (\mathtt{let}\ x = M\ \mathtt{in}\ N) : B}$$

$$\frac{\Delta \vdash_{\mathrm{rt}} P : \mathtt{bool} \qquad \Delta, P \vdash_{\mathrm{rt}} M : A \qquad \Delta, \neg P \vdash_{\mathrm{rt}} N : A}{\Delta \vdash_{\mathrm{rt}} (\mathtt{if}\ P\ \mathtt{then}\ M\ \mathtt{else}\ N) : A} \qquad \frac{\Xi \vdash_{\mathrm{rt}} V : G}{\Xi \vdash_{\mathrm{rt}} (V : G \Rrightarrow \bigstar) : \bigstar}$$

$$\frac{\Xi \vdash_{\mathrm{rt}} V : A \qquad \Xi \vdash_{\mathrm{rt}} (x : A)\{P\} : \mathtt{tp} \qquad \Xi \models P[x := V]}{\Xi \vdash_{\mathrm{rt}} (V : A \Rrightarrow (x : A)\{P\}) : (x : A)\{P\}} \qquad \frac{\Delta \vdash_{\mathrm{rt}} A : \mathtt{tp}}{\Delta \vdash_{\mathrm{rt}} \mathtt{blame}\ p : A}$$

Closed Environment Entailment. $\boxed{\Xi \models P}$

$$\frac{\text{for all } \sigma, (\Xi \longrightarrow^* \sigma \text{ implies } \sigma^*(P) \not\longrightarrow^* \mathtt{false})}{\Xi \models P}$$

Closed Environment Reduction. $\boxed{\Xi \longrightarrow \Xi', \sigma}$

$$\frac{M \longrightarrow N}{(x : A = M, \Xi) \longrightarrow (x : A = N, \Xi), \cdot} \qquad (x : A = V, \Xi) \longrightarrow \Xi, (\cdot, x = V)$$

$$\frac{P \longrightarrow Q}{(P, \Xi) \longrightarrow (Q, \Xi), \cdot} \qquad (\mathtt{true}, \Xi) \longrightarrow \Xi, \cdot$$

Closed Environment Evaluation. $\boxed{\Xi \longrightarrow^* \sigma}$

$$\cdot \longrightarrow^* \cdot \qquad \frac{\Xi \longrightarrow \Xi', \cdot \qquad \Xi' \longrightarrow^* \sigma}{\Xi \longrightarrow^* \sigma} \qquad \frac{\Xi \longrightarrow \Xi', (\cdot, x = V) \qquad \Xi'[x := V] \longrightarrow^* \sigma}{\Xi \longrightarrow^* x = V, \sigma}$$

Fig. 4. Run-time language type system

While other work uses explicit constructs to represent casts in progress [Knowles and Flanagan 2010; Wadler and Findler 2009], we instead use the 'Princeton' conditional directly.

3.5 Closed Environment Evaluaion

Our notion of entailment $\Xi \models P$ relies on using a closed environment Ξ, one which permits only extensions by predicates Ξ, P and let bindings $\Xi, x : A = M$. Our notion of entailment further depends on the successful evaluation $\Xi \longrightarrow^* \sigma$ of a closed environment Ξ to a closing substitution σ.

We evaluate environments to closing substitutions, by iterating reduction, and accumulating the resulting value bindings in the eventual closing substitution. For a let binding of the form $x : A = M$ we take the result V of evaluating M, and for a predicate binding P we check that the predicate evaluates to `true` in the current environment.

3.6 Closed Environment Entailment

We say that closed environment Ξ *entails* predicate P, written $\Xi \models P$ if for every evaluation $\Xi \longrightarrow^* \sigma$ to a closing substitution it is the case that $\sigma^*(P) \not\longrightarrow^* \texttt{false}$.

There is a noteworthy subtlety here. In Flanagan [2006]; Knowles and Flanagan [2010] and Greenberg et al. [2012] one finds the condition $\sigma(P) \longrightarrow^* \texttt{true}$, while in Keil and Thiemann [2015] and our work one finds $\sigma(P) \not\longrightarrow^* \texttt{false}$. The former outlaws the case where $\sigma(P)$ reduces to blame or does not terminate, while the latter permits it. Switching from the former to the latter allowed us to remove the restriction appearing in earlier work, such as Wadler and Findler [2009], that limits refinement to base types.

3.7 Defining substitutions σ

The substitutions we use here, σ, are simply finite maps from from names to *values*.

We write $\sigma^*(\Xi)$, $\sigma^*(M)$, and $\sigma^*(A)$ to denote applying substitution σ to closed environments, terms, and types respectively, with the usual Curry-style definition (with suitable choices of sufficiently fresh names), as in Figure 5. The action of a substitution may be extended to evaluation frames, $\sigma^*(\mathcal{E})$, and hence redexes, in the obvious structural way, satisfying $\sigma^*(\mathcal{E}[M]) = (\sigma^*(\mathcal{E}))[\sigma^*(M)]$.

We write $[x := V]$ to denote a single substitution of V for x and we write $[x]$ to denote a single substitution of x for x, which signified a potential hole in the term

$$M[x := V] \stackrel{\text{def}}{=} (\cdot, x = V)^*(M)$$
$$M[x] \stackrel{\text{def}}{=} (\cdot, x = x)^*(M)$$

3.8 Comparison to earlier approaches

Our definition of entailment is similar to the *Implication* rule of Knowles and Flanagan [2010], but thanks to the absence of subsumption we are working in a simpler setting. While Knowles and Flanagan [2010] needs to show implication between two predicates for all substitutions, we simply need to show that for all substitutions our predicate does not evaluate to false.

The initial work on hybrid type checking [Flanagan 2006] used typing judgements to define closing substitution, however that lead to an unpermitted circularity between their typing rule, subtyping rule, and implication rule, because the implication rule refers to closing substitutions in the negative position of an implication. Subsequent works on hybrid type checking [Greenberg et al. 2012; Knowles and Flanagan 2010] break that circularity by providing a denotational semantics of refinement types and defining the closing substitution in terms of the denotational semantics.

We believe our system solves that problem by foregoing subsumption and factoring λdB into a compile-time and a run-time language. As a consequence, first, we achieve decidable type-checking in our compile-time language. Second, we break the circularity present in the previous systems. We avoid the occurrence of the typing judgement in a negative position of an implication by permitting tagged dependently-typed values only in closed contexts. In such a case there is only a single substitution possible, therefore there is no need to quantify over all possible well-typed substitutions available in the context.

$$\boxed{\sigma^*(\Xi) = \Xi'}$$

$$\begin{array}{rcll}
\sigma^*(x : A = M, \Xi) & = & y : \sigma^*(A) = \sigma^*(M), (\sigma, x = y)^*(\Xi) & \text{fresh } y \\
\sigma^*(P, \Xi) & = & \sigma^*(P), \sigma^*(\Xi) &
\end{array}$$

$$\boxed{\sigma^*(A) = B}$$

$$\begin{array}{rcll}
\sigma^*(\iota) & = & \iota & \\
\sigma^*(\bigstar) & = & \bigstar & \\
\sigma^*((x : A)\{P\}) & = & (y : \sigma^*(A))\{(\sigma, x = y)^*(P)\} & \text{fresh } y \\
\sigma^*((x : A) \to B) & = & (y : \sigma^*(A)) \to (\sigma, x = y)^*(B) & \text{fresh } y
\end{array}$$

$$\boxed{\sigma^*(M) = N}$$

$$\begin{array}{rcll}
\sigma^*(c) & = & c & \\
(\cdot)^*(x) & = & x & \\
(\sigma, x = V)^*(x) & = & V & \\
(\sigma, x = V)^*(y) & = & \sigma^*(y) & \text{if } x \neq y \\
\sigma^*(\lambda x{:}A.\, N) & = & \lambda y{:}\sigma^*(A).\, (\sigma, x = y)^*(N) & \text{fresh } y \\
\sigma^*(V : G \Rrightarrow \bigstar) & = & \sigma^*(V) : G \Rrightarrow \bigstar & \\
\sigma^*(V : A \Rrightarrow (x : A)\{P\}) & = & \sigma^*(V) : \sigma^*(A) \Rrightarrow (y : \sigma^*(A))\{(\sigma, x = y)^*(P)\} & \text{fresh } y \\
\sigma^*(op(\vec{M})) & = & op(\sigma^*(\vec{M})) & \\
\sigma^*(L\ V) & = & \sigma^*(L)\ \sigma^*(V) & \\
\sigma^*(\texttt{let } x = M \texttt{ in } N) & = & \texttt{let } y = \sigma^*(M) \texttt{ in } (\sigma, x = y)^*(N) & \text{fresh } y \\
\sigma^*(\texttt{if } P \texttt{ then } M \texttt{ else } N) & = & \texttt{if } \sigma^*(P) \texttt{ then } \sigma^*(M) \texttt{ else } \sigma^*(N) & \\
\sigma^*(M : A \overset{p}{\Rightarrow} B) & = & \sigma^*(M) : \sigma^*(A) \overset{p}{\Rightarrow} \sigma^*(B) & \\
\sigma^*(\texttt{blame } p) & = & \texttt{blame } p &
\end{array}$$

Fig. 5. Action of substitutions on environments, types, and terms.

3.9 Dynamic Semantics

Figure 6 gives the reduction rules for $\lambda\mathsf{dB}$. We define evaluation for $\lambda\mathsf{dB}$ in terms of call-by-value reduction on terms (Figure 3), similar to Wadler and Findler [2009].

Our reduction rules for the evaluation of the operators, application of lambda abstractions, let bindings, and conditionals are standard.

Our rules for casts are reminiscent of Wadler and Findler [2009], with some technical differences. The earlier work had two separate reductions for casting base types and function types to $\bigstar$, where here we have a single rule which adds an intermediate cast to a compatible ground type. And the earlier work treats casts from functions to functions as values, where here such a cast results in a lambda abstraction, which is itself a value. Our system is shown to satisfy a property similar to the rule of the previous system:

$$(V : (x : A) \to B \overset{p}{\Rightarrow} (y : C) \to D)\ W \longrightarrow \texttt{let } x = (W : C \overset{-p}{\Rightarrow} A) \texttt{ in } V x : B[x] \overset{p}{\Rightarrow} D[W]$$

$$\boxed{M \longrightarrow N}$$

$$op(\vec{V}) \longrightarrow [\![op]\!](\vec{V}) \tag{1}$$

$$(\lambda x{:}A.\, N)\, V \longrightarrow N[x := V] \tag{2}$$

$$\texttt{let}\ x = V\ \texttt{in}\ N \longrightarrow N[x := V] \tag{3}$$

$$\texttt{if true then}\ M\ \texttt{else}\ N \longrightarrow M \tag{4}$$

$$\texttt{if false then}\ M\ \texttt{else}\ N \longrightarrow N \tag{5}$$

$$V : \iota \overset{p}{\Rightarrow} \iota \longrightarrow V \tag{6}$$

$$V : \star \overset{p}{\Rightarrow} \star \longrightarrow V \tag{7}$$

$$V : A \overset{p}{\Rightarrow} \star \longrightarrow V : A \overset{p}{\Rightarrow} G \Rrightarrow \star \qquad \text{if } A \neq \star, A \sim G \tag{8}$$

$$V : G \Rrightarrow \star \overset{p}{\Rightarrow} A \longrightarrow V : G \overset{p}{\Rightarrow} A \qquad \text{if } G \sim A \tag{9}$$

$$V : G \Rrightarrow \star \overset{p}{\Rightarrow} B \longrightarrow \texttt{blame}\, p \qquad \text{if } G \not\sim A \tag{10}$$

$$V : A \Rrightarrow (x : A)\{P\} \overset{p}{\Rightarrow} B \longrightarrow V : A \overset{p}{\Rightarrow} B \tag{11}$$

$$V : A \overset{p}{\Rightarrow} (y : B)\{Q\} \longrightarrow \texttt{let}\ y = (V : A \overset{p}{\Rightarrow} B)\ \texttt{in}$$
$$\texttt{if}\ Q\ \texttt{then}\ y : B \Rrightarrow (y : B)\{Q\}\ \texttt{else}\ \texttt{blame}\, p \tag{12}$$

$$(\lambda x{:}A.\, N) : (x : A) \to B \overset{p}{\Rightarrow} (y : C) \to D \longrightarrow \lambda y{:}C.\, \texttt{let}\ x = (y : C \overset{-p}{\Rightarrow} A)\ \texttt{in}\ N : B[x] \overset{p}{\Rightarrow} D[y] \tag{13}$$

$$\frac{M \longrightarrow N}{\mathcal{E}[M] \longrightarrow \mathcal{E}[N]} \qquad \frac{}{\mathcal{E}[\texttt{blame}\, p] \longrightarrow \texttt{blame}\, p}$$

Fig. 6. Dynamic Semantics

We have standard rules to take the compatible closure under evaluation frames, and to propogate blame through an enclosing evaluation frame.

Conjecture 3.1 (Diamond Property). *If $M \longrightarrow N$ and $M \longrightarrow N'$, with $N \neq N'$ there is some term L such that $N \longrightarrow L$ and $N' \longrightarrow L$.*

Proof. The most interesting case is given by $M \equiv V : (x : A)\{P\} \overset{p}{\Rightarrow} (y : B)\{Q\}$. This yields the following two paths:

- by matching on the refinement type on the left first

$$\begin{aligned} & V : A \Rrightarrow (x : A)\{P\} \overset{p}{\Rightarrow} (y : B)\{Q\} \\ \longrightarrow\ & V : A \overset{p}{\Rightarrow} (y : B)\{Q\} \\ \longrightarrow\ & \texttt{let}\ y = V : A \overset{p}{\Rightarrow} B\ \texttt{in}\ \texttt{if}\ Q\ \texttt{then}\ y : B \Rrightarrow (y : B)\{Q\}\ \texttt{else}\ \texttt{blame}\, p \end{aligned}$$

- by matching on the refinement type on the right first

$$V : A \Rrightarrow (x : A)\{P\} \overset{p}{\Rrightarrow} (y : B)\{Q\}$$
$$\longrightarrow \texttt{let}\ y = V : A \Rrightarrow (x : A)\{P\} \overset{p}{\Rrightarrow} B\ \texttt{in}\ \texttt{if}\ Q\ \texttt{then}\ y : B \Rrightarrow (y : B)\{Q\}\ \texttt{else}\ \texttt{blame}\ p$$
$$\longrightarrow \texttt{let}\ y = V : A \overset{p}{\Rrightarrow} B\ \texttt{in}\ \texttt{if}\ Q\ \texttt{then}\ y : B \Rrightarrow (y : B)\{Q\}\ \texttt{else}\ \texttt{blame}\ p$$

which are confluent. □

4 TYPE SAFETY

Every typeable term in our language has a well-formed type.

Proposition 4.1 (Well-formed types). *If* $\Delta \vdash_{\mathrm{rt}} M : A$ *then* $\Delta \vdash_{\mathrm{rt}} A : \mathtt{tp}$.

Proof. By induction on $\Delta \vdash_{\mathrm{rt}} M : A$. □

Moreover, the system enjoys unicity of types.

Proposition 4.2 (Unicity). *Let* M *be a term where no subterm has the form* `blame` *for any* p. *If* $\Delta \vdash_{\mathrm{rt}} M : A$ *and* $\Delta \vdash_{\mathrm{rt}} M : B$ *then* $A = B$.

Proof. For each term except blame there is a unique typing derivation. □

It would be easy to extend the result to include blame, if blame terms explicitly carried their type. We require slight adjustments to the usual canonical forms lemma.

Lemma 4.3 (Canonical forms). *Let* V *be a value that is well typed in the empty context.*

- *If* $\cdot \vdash_{\mathrm{rt}} V : \iota$, *then* $V = c$ *with* $\mathit{type}(c) = \iota$.
- *If* $\cdot \vdash_{\mathrm{rt}} V : \bigstar$, *then* $V = W : G \Rrightarrow \bigstar$ *with* $\cdot \vdash_{\mathrm{rt}} W : G$.
- *If* $\cdot \vdash_{\mathrm{rt}} V : (x : A)\{P\}$, *then* $V = W : A \Rrightarrow (x : A)\{P\}$ *with* $\cdot \vdash_{\mathrm{rt}} W : A$ *and* $\cdot \models P[W]$.
- *If* $\cdot \vdash_{\mathrm{rt}} V : (x : A) \to B$ *then* $V = \lambda x{:}A.\, N$ *with* $x : A \vdash_{\mathrm{rt}} N : B$.

Proof. By case analysis on the typing derivation of V in the empty context. □

Whereas traditional progress shows that a term well-typed in the empty context either is a value or takes a step, here there is a third possibility, which is that it results in blame.

Proposition 4.4 (Progress). *If* $\cdot \vdash_{\mathrm{rt}} M : A$ *then either:*

- M *is a value.*
- $M \longrightarrow N$ *for some term* N.
- M *is* `blame` p *for some blame label* p.

Proof. By induction on the typing derivation of M in the empty context. □

It is straightforward to show that reduction of closed terms preserves types.

Conjecture 4.5 (Preservation). *If* $\cdot \vdash_{\mathrm{rt}} M : A$ *and* $M \longrightarrow N$ *then* $\cdot \vdash_{\mathrm{rt}} N : A$

Proof. By case analysis over the reduction rules. The case for a reduction with left-hand side

$$V : A \overset{p}{\Rrightarrow} (y : B)\{Q\}$$

depends crucially on the type rule for $V : A \Rrightarrow (y : A)\{Q\}$ using a closed context Ξ rather than the empty context. □

Context Morphism. $\boxed{\rho : \Xi \longrightarrow \Delta}$

$$\frac{}{\cdot : \cdot \longrightarrow \cdot} \qquad \frac{\rho : \Xi \longrightarrow \Delta \qquad \Xi \vdash_{rt} \rho^*(V) : \rho^*(A)}{(\rho, x = \rho^*(V)) : \Xi \longrightarrow (\Delta, x : A)}$$

$$\frac{\rho : \Xi \longrightarrow \Delta \qquad \Xi \vdash_{rt} \rho^*(M) : \rho^*(A)}{(\rho, x = y) : (\Xi, y : \rho^*(A) = \rho^*(M)) \longrightarrow (\Delta, x : A = M)} \qquad \frac{\rho : \Xi \longrightarrow \Delta \qquad \Xi \vdash_{rt} \rho^*(P) : \mathit{bool}}{\rho : (\Xi, \rho^*(P)) \longrightarrow (\Delta, P)}$$

Fig. 7. Context Morphisms

4.1 Context morphisms

We now introduce additional technical machinery that lets us describe type preservation for arbitrary as well as empty contexts, and that will prove essential in defining blame safety in the next section.

Specifically, in addition to our analysis of entailment in closed environments, we further need a device to enable us to pass from an arbitrary environment to such closed environments. Such a device performs a similar role to the ordinary notion of 'closing substitution', or to the let-bound values of Knowles et al. [2006], but here, we also need to pay attention to the structure of let and predicate bindings in the environment.

The device we introduce here is an adaptation of the classical notion of *context morphism* from categorical logic [Pitts 2000] and the metatheory of dependent types [Dybjer 1995; Goguen and McKinna 1997]. In those works, context morphisms may be thought of as a kind of 'well-typed substitution', $\theta : \Gamma \longrightarrow \Gamma'$, where θ is a mapping of variables to terms such that if $x : A$ appears in Γ' then $\theta(x) = M$ with $\Gamma \vdash_{\mathrm{rt}} M : A$. In categorical terms, context morphisms provide a means of reindexing a judgement typed in the target context to one typed in the source context. Reindexing is contravariant, which is why the direction of the morphism arrow is opposite to the action of the associated substitution. Context morphisms satisfy obvious notions of identity and composition.

Here, we introduce a specialised notion of context morphism, $\rho : \Xi \longrightarrow \Delta$, defined inductively in Figure 7. Now the substitution is restricted to yield values rather than arbitrary terms; the source is restricted to a closed environment; both the source and target environment may contain let bindings and predicate bindings. Our definition ensures that the structure of successive let and predicate bindings in the target Δ is carried back into the source Ξ, and the value bindings in ρ instantiate the variable bindings in Δ: if $x : A$ appears in Δ then $\rho(x) = V$ with $\Xi \vdash_{\mathrm{rt}} V : A$.

We have the following easily proved property, stability of entailment under context morphisms:

Lemma 4.6 (Stability). *If* $\Xi \models P$ *and* $\rho : \Xi' \longrightarrow \Xi$ *then* $\Xi' \models \rho^*(P)$.

There is an obvious notion of identity morphism, $\rho_{id} : \Xi \longrightarrow \Xi$ when source and target coincide, where $\rho_{id} = \cdot$ with ${\rho_{id}}^*(M) = M$. More generally, any morphism between closed environments is a *renaming*, namely a substitution consisting entirely of value bindings of the form $x = y$ where x is a variable in a let-binding of the target and y the corresponding variable in a let-binding of the source. The identity morphism is a renaming, and renamings are closed under composition.

We insist on the source environment being closed in order that we may consider its behaviour under evaluation. As shown in Figure 8, we say that η is a closing substitution for Δ, written $\eta : \Delta$, if there exists a closed context Ξ such that $\rho : \Xi \longrightarrow \Delta$ and $\Xi \longrightarrow^* \sigma$ with $\eta = \sigma \circ \rho$.

Closing substitutions preserve types.

Conjecture 4.7 (Closing Substitution). *Assume a closing substitution* $\eta : \Delta$.

Closing Substitution. $\boxed{\eta : \Delta}$

$$\frac{\rho : \Xi \longrightarrow \Delta \qquad \Xi \longrightarrow^* \sigma \qquad \eta = \sigma \circ \rho}{\eta : \Delta}$$

Fig. 8. Closing Substitution

- *If* $\Delta \vdash_{\mathrm{rt}} M : A$ *then* $\cdot \vdash_{\mathrm{rt}} \eta^*(M) : \eta^*(A)$.
- *If* $\Delta \vdash_{\mathrm{rt}} A : \mathtt{tp}$ *and* $\eta : \Delta$ *then* $\cdot \vdash_{\mathrm{rt}} \eta^*(A) : \mathtt{tp}$

We expect the proofs of the above to be routine, if intricate, inductive arguments.

Having developed the above machinery, we may now generalise type preservation from closed terms to open terms.

Corollary 4.8 (Type safety for open terms). *Assume a closing substitution* $\eta : \Delta$. *If* $\Delta \vdash_{\mathrm{rt}} M : A$ *and* $\eta^*(M) \longrightarrow^* N$, *then* $\cdot \vdash_{\mathrm{rt}} N : \eta^*(A)$.

We will use the same machinery for defining blame safety on open terms in the next section.

5 BLAME SAFETY AND SUBTYPING

Our approach to blame safety follows that of Wadler and Findler [2009], in that we define a safety judgment and show preservation of safety under reduction. However, where that paper used subtyping relations on pairs of types, here we use novel three-place relations on closed values and four-place relations on open terms, which are defined by mutual recursion. The customary treatment of subtyping as a two-place relation between types then arises as a derived notion.

5.1 Subtyping with a Witness

Our new concept, *subtyping with a witness*, is defined in Figure 9. We define a three-place relation between a closed term V and two closed types A and B,

$$V : A <: B$$

which presumes $\cdot \vdash_{\mathrm{rt}} V : A$, $\cdot \vdash_{\mathrm{rt}} B : \mathtt{tp}$, and $A \sim B$. We also define a four-place relation between an environment Δ, an open term M, and two open types A and B,

$$\Delta \vdash M : A <: B$$

which presumes $\Delta \vdash_{\mathrm{rt}} M : A$, $\Delta \vdash_{\mathrm{rt}} B : \mathtt{tp}$, and $A \sim B$.

In fact, we define four variants of subtyping:

$$A <: B \qquad A <:^+ B \qquad A <:^- B \qquad A <:_n B$$

We also use $A <:^{\pm}_n B$ to range over any of these four relations.

Our motivation in including a value in the subtyping judgement, is that a value *witnesses* the possibility that we may successfully cast from the first type to the second type.

For our rule for subtyping from a refinement,

$$\frac{V = (W : A \Rrightarrow (x : A)\{P\}) \qquad W : A <: B}{V : (x : A)\{P\} <: B}$$

the Canonical Forms Lemma guarantees that a value of refinement type must have the form $W : A \Rrightarrow (x : A)\{P\}$, we then remove the tag and check that $W : A <: B$.

Ordinary Subtyping with a witness. $\boxed{V : A <: B}$

$$V : \iota <: \iota \qquad V : \bigstar <: \bigstar \qquad \frac{V : A <: G}{V : A <: \bigstar}$$

$$\frac{V = (W : A \Rrightarrow (x : A)\{P\}) \qquad W : A <: B}{V : (x : A)\{P\} <: B} \qquad \frac{V : A <: B \qquad y : B = (V : A \overset{\bullet}{\Rightarrow} B) \models Q}{V : A <: (y : B)\{Q\}}$$

$$\frac{y : C \vdash y : C <: A \qquad y : C, x : A = (y : C \overset{\bullet}{\Rightarrow} A) \vdash (V\ x) : B <: D}{V : (x : A) \to B <: (y : C) \to D}$$

Positive Subtyping with a witness. $\boxed{V : A <:^+ B}$

$$V : \iota <:^+ \iota \qquad V : A <:^+ \bigstar$$

$$\frac{V = (W : A \Rrightarrow (x : A)\{P\}) \qquad W : A <:^+ B}{V : (x : A)\{P\} <:^+ B} \qquad \frac{V : A <:^+ B \qquad y : B = (V : A \overset{\bullet}{\Rightarrow} B) \models Q}{V : A <:^+ (y : B)\{Q\}}$$

$$\frac{y : C \vdash y : C <:^- A \qquad y : C, x : A = (y : C \overset{\bullet}{\Rightarrow} A) \vdash (V\ x) : B <:^+ D}{V : (x : A) \to B <:^+ (y : C) \to D}$$

Negative Subtyping with a witness. $\boxed{V : A <:^- B}$

$$V : \iota <:^- \iota \qquad V : \bigstar <:^- \bigstar \qquad \frac{V : A <:^- G}{V : A <:^- B}$$

$$\frac{V = (W : A \Rrightarrow (x : A)\{P\}) \qquad W : A <:^- B}{V : (x : A)\{P\} <:^- B} \qquad \frac{V : A <:^- B}{V : A <:^- (y : B)\{Q\}}$$

$$\frac{y : C \vdash y : C <:^+ A \qquad y : C, x : A = (y : C \overset{\bullet}{\Rightarrow} A) \vdash (V\ x) : B <:^- D}{V : (x : A) \to B <:^- (y : C) \to D}$$

Naive Subtyping with a witness. $\boxed{V : A <:_n B}$

$$V : \iota <:_n \iota \qquad V : A <:_n \bigstar$$

$$\frac{V = (W : A \Rrightarrow (x : A)\{P\}) \qquad W : A <:_n B}{V : (x : A)\{P\} <:_n B} \qquad \frac{V : A <:_n B \qquad y : B = (V : A \overset{\bullet}{\Rightarrow} B) \models Q}{V : A <:_n (y : B)\{Q\}}$$

$$\frac{x : A \vdash x : A <:_n C \qquad y : C, x : A = (y : C \overset{\bullet}{\Rightarrow} A) \vdash (V\ x) : B <:_n D}{V : (x : A) \to B <:_n (y : C) \to D}$$

Open Subtyping with a witness. $\boxed{\Delta \vdash M : A <:^{\pm}_n B}$

$$\frac{\text{for all } \eta : \Delta, (\eta^*(M) \longrightarrow^* V \text{ implies } V : \eta^*(A) <:^{\pm}_n \eta^*(B))}{\Delta \vdash M : A <:^{\pm}_n B}$$

Fig. 9. Subtyping with a witness

For our rule for subtyping to a refinement,

$$\frac{V : A <: B \qquad y : B = (V : A \overset{\bullet}{\Rightarrow} B) \models Q}{V : A <: (y : B)\{Q\}}$$

we check that $V : A <: B$, the confirm that if we cast V from type A to type B the result satisfies the predicate Q. We write $\bullet$ when the choice of blame label does not matter.

For our rule for subtyping between dependent functions,

$$\frac{y : C \vdash y : C <: A \qquad y : C, x : A = (y : C \overset{\bullet}{\Rightarrow} A) \vdash (V\ x) : B <: D}{V : (x : A) \to B <: (y : C) \to D}$$

we check that the corresponding domains and ranges are also subtyping, where subtyping is contravariant in the domain and covariant in the range. Since the ranges are dependent, we use the judgement for open terms, in an environment where y is any value of type C, and x is the result of casting Y from type C to type A.

For open terms, we quantify over all closing substitutions and then check the corresponding relation on closed terms,

$$\frac{\text{for all } \eta : \Delta, (\eta^*(M) \longrightarrow^* V \text{ implies } V : \eta^*(A) <: \eta^*(B))}{\Delta \vdash M : A <: B}$$

Properties of subtyping

It is easy to show that subtyping with a witness implies compatibility and is reflexive. It is a little trickier to formulate transitivity.

Corollary 5.1 (Properties of subtyping with a witness).

- *Compatibility:* $V : A <: B$ *implies* $A \sim B$.
- *Reflexivity:* $V : A <: A$.
- *Transitivity: If* $V : A <: B$ *and* $y = (V : A \overset{\bullet}{\Rightarrow} B) \vdash y : B <: C$ *then* $V : A <: C$.

Compatibility and reflexivity hold for all four relations, and transitivity holds for ordinary and naive subtyping.

We also adapt the Tangram property of Wadler and Findler [2009] to subtyping with a witness. The original Tangram property consists of two factoring lemmas:

- $A <: B$ iff $A <:^+ B$ and $A <:^- B$,
- $A <:_n B$ iff $A <:^+ B$ and $B <:^- A$.

In the second factoring lemma, the type on the left-hand side of the negative subtyping is swapped with the type on the right-hand side. The first of these adapts straightfowardly to witnesses, while the second uses a trick similar to that for transitivity.

Conjecture 5.2 (Tangram with a witness).

- $V : A <: B$ *iff* $V : A <:^+ B$ *and* $V : A <:^- B$.
- $V : A <:_n B$ *iff* $V : A <:^+ B$ *and* $y = (V : A \overset{\bullet}{\Rightarrow} B) \vdash y : B <:^- A$.

5.2 Subtyping without a witness

We can recover the two-place subtyping of Wadler and Findler [2009] by using the four-place relation to quantify over all possible witnesses.

$$A <: B \quad \overset{\text{def}}{=} \quad x : A \vdash x : A <: B.$$

This immediately yields the following corrolaries.

Corollary 5.3 (Properties of subtyping without a witness).

- *Compatibility:* $A <: B$ *implies* $A \sim B$.
- *Reflexivity:* $A <: A$.
- *Transitivity: If* $A <: B$ *and* $B <: C$ *then* $A <: C$.

Compatibility and reflexivity hold for all four relations, and transitivity holds for ordinary and naive subtyping.

Corollary 5.4 (Tangram without a witness).

- $A <: B$ *iff* $A <:^+ B$ *and* $A <:^- B$, *and*
- $A <:_n B$ *iff* $A <:^+ B$ *and* $B <:^- A$.

5.3 Blame Safety

We give an inductive definition of *safety with respect to a blame label* in Figure 10. The key part of the definition is as follows. Assume an environment Δ. A cast

$$M : A \overset{p}{\Rightarrow} B$$

is safe for p if $\Delta \vdash M : A <:^+ B$, and a cast

$$M : A \overset{-p}{\Rightarrow} B$$

is safe for p if $\Delta \vdash M : A <:^- B$, while a cast

$$M : A \overset{q}{\Rightarrow} B$$

with $|q| \neq |p|$ is always safe for p. We also need to check that any other types or terms appearing within the term are also safe for p.

An easy induction establishes the following Lemma:

Lemma 5.5 (Safe terms have safe types). *If* $\Delta \vdash_{\mathrm{rt}} M : A$ *and* $\Delta \vdash M$ `safe` p *then* $\Delta \vdash A$ `safe` p.

In a similar manner to Wadler and Findler [2009] we state our blame safety result in terms of blame safety preservation and blame safety progress. As above, our general strategy aims to prove the result in two stages. The first shows safety preservation from open judgments to the empty context:

Conjecture 5.6 (Blame safety for open terms). *Assume a closing substitution* $\eta : \Delta$ *where every value in* η *is safe, that is, if* $\eta(x) = V$ *then* $\cdot \vdash V$ `safe` p. *If* $\Delta \vdash_{\mathrm{rt}} M : A$ *and* $\Delta \vdash M$ `safe` p *then* $\cdot \vdash \eta^*(M)$ `safe` p.

For closed terms, we have blame safety preservation and progress.

Conjecture 5.7 (Blame safety preservation). *If* $\cdot \vdash_{\mathrm{rt}} M : A$ *and* $\cdot \vdash M$ `safe` p *and* $M \longrightarrow N$ *then* $\cdot \vdash N$ `safe` p.

Conjecture 5.8 (Blame safety progress). *If* $\cdot \vdash_{\mathrm{rt}} M : A$ *and* $\cdot \vdash M$ `safe` p *then* $M \neq$ `blame` p.

From these, we can then conclude the Blame Theorem of Wadler and Findler [2009].

Corollary 5.9 (Blame Theorem). *Let* $\mathfrak{C}$ *be a context of type* C *containing a hole of type* B, *and let* M *be a term of type* A, *where* p *and* $-p$ *do not appear in* $\mathfrak{C}$, M, A, B, *or* C.

- *If* $A <: B$ *then* $\mathfrak{C}[M : A \overset{p}{\Rightarrow} B] \not\longrightarrow^*$ `blame` p *and* $\mathfrak{C}[M : A \overset{p}{\Rightarrow} B] \not\longrightarrow^*$ `blame` $-p$.
- *If* $A <:_n B$ *then* $\mathfrak{C}[M : A \overset{p}{\Rightarrow} B] \not\longrightarrow^*$ `blame` p.
- *If* $B <:_n A$ *then* $\mathfrak{C}[M : A \overset{p}{\Rightarrow} B] \not\longrightarrow^*$ `blame` $-p$.

Safety for contexts. $\boxed{\Delta \ \mathsf{safe}\ p}$

$$\frac{}{\cdot \ \mathsf{safe}\ p} \qquad \frac{\Delta \ \mathsf{safe}\ p \qquad \Delta \vdash A \ \mathsf{safe}\ p}{\Delta, x : A \ \mathsf{safe}\ p} \qquad \frac{\Delta \vdash A \ \mathsf{safe}\ p \qquad \Delta \vdash M \ \mathsf{safe}\ p}{\Delta, x : A = M \ \mathsf{safe}\ p}$$

$$\frac{\Delta \ \mathsf{safe}\ p \qquad \Delta \vdash P \ \mathsf{safe}\ p}{\Delta, P \ \mathsf{safe}\ p}$$

Safety for types. $\boxed{\Delta \vdash A \ \mathsf{safe}\ p}$

$$\frac{\Delta \ \mathsf{safe}\ p}{\Delta \vdash \iota \ \mathsf{safe}\ p} \qquad \frac{\Delta \vdash A \ \mathsf{safe}\ p \qquad \Delta, x : A \vdash P \ \mathsf{safe}\ p}{\Delta \vdash (x : A)\{P\} \ \mathsf{safe}\ p} \qquad \frac{\Delta \vdash A \ \mathsf{safe}\ p \qquad \Delta, x : A \vdash B \ \mathsf{safe}\ p}{\Delta \vdash ((x : A) \to B) \ \mathsf{safe}\ p}$$

Safety for Terms. $\boxed{\Delta \vdash M \ \mathsf{safe}\ p}$

$$\frac{\Delta \ \mathsf{safe}\ p}{\Delta \vdash c \ \mathsf{safe}\ p} \qquad \frac{\Delta \vdash \vec{M} \ \mathsf{safe}\ p}{\Delta \vdash op(\vec{M}) \ \mathsf{safe}\ p} \qquad \frac{\Delta \ \mathsf{safe}\ p \qquad x \in \mathrm{dom}(\Delta)}{\Delta \vdash x \ \mathsf{safe}\ p} \qquad \frac{\Delta, x : A \vdash N \ \mathsf{safe}\ p}{\Delta \vdash (\lambda x{:}A.\, N) \ \mathsf{safe}\ p}$$

$$\frac{\Delta \vdash L \ \mathsf{safe}\ p \qquad \Delta \vdash V \ \mathsf{safe}\ p}{\Delta \vdash (L\ V) \ \mathsf{safe}\ p} \qquad \frac{\Delta \vdash M \ \mathsf{safe}\ p \qquad \Delta, x : A = M \vdash N \ \mathsf{safe}\ p}{\Delta \vdash (\mathtt{let}\ x = M\ \mathtt{in}\ N) \ \mathsf{safe}\ p}$$

$$\frac{\Delta \vdash P \ \mathsf{safe}\ p \qquad \Delta, P \vdash M \ \mathsf{safe}\ p \qquad \Delta, \neg P \vdash N \ \mathsf{safe}\ p}{\Delta \vdash (\mathtt{if}\ P\ \mathtt{then}\ M\ \mathtt{else}\ N) \ \mathsf{safe}\ p}$$

$$\frac{\Delta \vdash M \ \mathsf{safe}\ p \qquad \Delta \vdash M : A <:^{+} B \qquad \Delta \vdash B \ \mathsf{safe}\ p}{\Delta \vdash (M : A \overset{p}{\Rightarrow} B) \ \mathsf{safe}\ p}$$

$$\frac{\Delta \vdash M \ \mathsf{safe}\ p \qquad \Delta \vdash M : A <:^{-} B \qquad \Delta \vdash B \ \mathsf{safe}\ p}{\Delta \vdash (M : A \overset{-p}{\Rightarrow} B) \ \mathsf{safe}\ p}$$

$$\frac{\Delta \vdash M \ \mathsf{safe}\ p \qquad |q| \neq |p| \qquad \Delta \vdash B \ \mathsf{safe}\ p}{\Delta \vdash (M : A \overset{q}{\Rightarrow} B) \ \mathsf{safe}\ p} \qquad \frac{\Xi \vdash V \ \mathsf{safe}\ p}{\Xi \vdash (V : G \Rrightarrow \bigstar) \ \mathsf{safe}\ p}$$

$$\frac{\Xi \vdash V \ \mathsf{safe}\ p \qquad \Xi \vdash P \ \mathsf{safe}\ p}{\Xi \vdash (V : A \Rrightarrow (x : A)\{P\}) \ \mathsf{safe}\ p} \qquad \frac{q \neq p}{\Delta \vdash (\mathtt{blame}\ q) \ \mathsf{safe}\ p} \qquad \frac{\text{there is no } \eta \text{ such that } \eta : \Delta}{\Delta \vdash (\mathtt{blame}\ p) \ \mathsf{safe}\ p}$$

Fig. 10. Blame Safety for Contexts, Types, and Terms

6 RELATED WORK

Hybrid Type Checking. Hybrid type checking [Flanagan 2006] allows for writing dependently-typed programs whose type-checking is potentially undecidable — hybrid type checker will insert dynamic casts to ensure that the typing discipline is enforced during run-time and provides a

guarantee that if the program is indeed well-typed it will reduce successfully. However, hybrid type checking only permits refinements and dependent functions over simple types.

Manifest Contracts. Greenberg et al. [2010, 2012] focus on the interplay between contracts and hybrid type checking. Our language λdB is similar to their λH as we both fix the order of evaluation to be call-by-value, however they need to define denotational semantics of types and kinds for their closing substitutions to break a circularity problem that would arise from the subsumption rule, whereas we forego subsumptions and define closing substitutions by composing context morphisms and substitutions arising by evaluating closed environments.

Gradual Refinement Types. Lehmann and Tanter [2017] present a gradually typed language with refinement types. Like Ou et al. [2004], Flanagan [2006], and Wadler and Findler [2009], refinements are only over base types, and like Ou et al. [2004], refinements must be decidable (they use a SAT solver), and have a strongly restricted syntax rather than ranging over arbitrary terms. Their work is novel in that they consider unknown refinements and apply the methodology of AGT [Garcia et al. 2016]. They do not consider blame.

Dependent Interoperability. Tanter and Tabareau [2015] and Dagand et al. [2016, 2018] present an interoperability framework added as a library to Coq. They do not support refinement types *per se*, but do support the related notion of sigma types and are not restricted to base types. They do not consider blame.

Approximate Normalization for Gradual Dependent Types. Eremondi et al. [2019] present a gradual type theory. Their work is novel in that they consider unknown values as well as unknown types, and again they apply the methodology of AGT [Garcia et al. 2016]. As with the previous work, they do not support refinement types *per se*, but do support the related notion of sigma types and are not restricted to base types. They do not consider blame.

REFERENCES

Pierre-Evariste Dagand, Nicolas Tabareau, and Éric Tanter. 2016. Partial type equivalences for verified dependent interoperability. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 298–310.

Pierre-Evariste Dagand, Nicolas Tabareau, and Éric Tanter. 2018. Foundations of dependent interoperability. *Journal of Functional Programming* 28 (2018).

Peter Dybjer. 1995. Internal Type Theory. In *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers (Lecture Notes in Computer Science)*, Stefano Berardi and Mario Coppo (Eds.), Vol. 1158. Springer, 120–134. https://doi.org/10.1007/3-540-61780-9_66

Joseph Eremondi, Éric Tanter, and Ronald Garcia. 2019. Approximate Normalization for Gradual Dependent Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 88 (July 2019), 30 pages. https://doi.org/10.1145/3341692

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-order Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM, New York, NY, USA, 48–59.

Cormac Flanagan. 2006. Hybrid Type Checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 245–256.

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. ACM, New York, NY, USA, 429–442.

Healfdene Goguen and James McKinna. 1997. *Candidates for Substitution.* Technical Report ECS-LFCS-97-358. LFCS, University of Edinburgh. Available at http://www.lfcs.inf.ed.ac.uk/reports/97/ECS-LFCS-97-358/ECS-LFCS-97-358.pdf.

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2010. Contracts Made Manifest. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 353–364.

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2012. Contracts made manifest. *Journal of Functional Programming* 22 (5 2012), 225–274. Issue 03.

Matthias Keil and Peter Thiemann. 2015. Blame Assignment for Higher-order Contracts with Intersection and Union. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 375–386.

Kenneth Knowles and Cormac Flanagan. 2010. Hybrid Type Checking. *ACM Trans. Program. Lang. Syst.* 32, 2, Article 6 (Feb. 2010), 34 pages.

Kenneth Knowles, Aaron Tomb, Jessica Gronski, Stephen N Freund, and Cormac Flanagan. 2006. SAGE: Unified hybrid checking for first-class types, general refinement types, and dynamic (extended report). (2006).

Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 775–788. https://doi.org/10.1145/3009837.3009856

Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-language Programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 3–10.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. Dynamic Typing with Dependent Types. In *Exploring New Frontiers of Theoretical Informatics: IFIP 18th World Computer Congress TC1 3rd International Conference on Theoretical Computer Science (TCS2004) 22–27 August 2004 Toulouse, France*, Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell (Eds.). Springer US, Boston, MA, 437–450.

Andrew M. Pitts. 2000. Categorical Logic. In *Handbook of Logic in Computer Science, Volume 5. Algebraic and Logical Structures*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum (Eds.). Oxford University Press, Chapter 2, 39–128. http://www.oup.co.uk/isbn/0-19-853781-6

Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2013. Secure distributed programming with value-dependent types. *J. Funct. Program.* 23, 4 (2013), 402–451.

Éric Tanter and Nicolas Tabareau. 2015. Gradual Certified Programming in Coq. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS 2015)*. ACM, New York, NY, USA, 26–40.

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 964–974.

Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can't be blamed. In *Programming Languages and Systems*. Springer, 1–16.