Blame Tracking at Higher Fidelity

Jakub Zalewski

Master of Science by Research
CDT Pervasive Parallelism
School of Informatics
University of Edinburgh
2016
Abstract

Type systems help to ensure correct software as sound type checkers will statically reject incorrect programs. Dependent types and refinement types allow for specifying stronger constraints and expressing specification of the program. Flanagan introduced a breakthrough idea of hybrid type checking, which allows for expressive refinements using the same language as the program.

Gradual typing allows refinement types to be added gradually as they are needed. Wadler and Findler introduced blame calculus and extended gradual typing the notion of blame and blame safety, which allow for precise analysis of runtime errors.

We combine the strengths of each — hybrid type checking has no notion of blame safety, while blame calculus lacks dependent function types. In addition, we want to remove the restriction of refinement types to base types in both systems, and remove the technicalities required to break circularities in hybrid type checking.

We make blame analysis more precise by introducing a novel notion of subtyping with witness to achieve blame tracking at higher fidelity.
Acknowledgements

I thank Simon Fowler for his feedback, incessant support, and tolerance of my quirky grammar. I thank my supervisors, Phil and Garrett, for their guidance and their feedback on my research work during the last year. I thank Michel Steuwer for providing feedback and more pragmatic point of view on earlier version of this work. Finally, I want to thank my CDT cohort and the whole office 1.07 (and 1.09) in Informatics Forum for the amazing year.

This work was supported by the EPSRC Centre for Doctoral Training in Pervasive Parallelism, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L01503X/1).
Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Jakub Zalewski)
Moin rodzicom.
Contents

1 Introduction ......................................................... 1
  1.1 Contributions .................................................. 2

2 Background ......................................................... 3
  2.1 Dependent Types ............................................... 4
    2.1.1 Refinement Types ...................................... 4
    2.1.2 Dependent Functions .................................... 5
  2.2 Dynamically Checked Decidable Predicates .................... 5
  2.3 Hybrid Type Checking ......................................... 5
  2.4 Gradual Typing ................................................ 6
  2.5 Blame Calculus ................................................ 6
    2.5.1 Casts ...................................................... 7
    2.5.2 Blame ....................................................... 7
    2.5.3 Blame Theorem ........................................... 7

3 Dependent Blame Calculus (λdB) .................................. 9
  3.1 Compile-time language ........................................ 9
    3.1.1 Compile-time Typing .................................... 11
  3.2 Run-time Language ............................................. 14
    3.2.1 Run-time Typing ......................................... 14
    3.2.2 Soundness and Completeness of Typing Judgements .... 17
  3.3 Dynamic Semantics ............................................. 18
  3.4 Type Safety ................................................... 18
    3.4.1 Diamond Property ....................................... 20

4 Blame Safety at Higher Fidelity ................................ 23
  4.1 Subtyping with a Witness .................................... 24
    4.1.1 Semantic Judgement .................................... 25
List of Figures

3.1 Compile-time Language, $\lambda dB$ ........................................ 10
3.2 Typing rules for compile-time language. ................................. 11
3.3 Compatibility for types. ..................................................... 12
3.4 Well-formed types. ........................................................... 13
3.5 Context validity judgement ................................................ 14
3.6 Run-time language, $\lambda dB$ ................................................. 15
3.7 Typing rules for target language. ......................................... 16
3.8 Reductions ................................................................. 19
4.1 Subtyping with a witness. .................................................... 24
4.2 Semantic Judgement .......................................................... 25
4.3 Closing substitution. ......................................................... 26
4.4 Applying order-preserving closing substitutions ....................... 26
4.5 Positive and negative subtyping ......................................... 27
4.6 Naive Subtyping with a witness .......................................... 28
4.7 Safety for Terms and Types ............................................... 32
Chapter 1

Introduction

Writing parallel and concurrent code is hard due to deadlocks, race conditions, and non-deterministic execution. Types provide a static guarantee that a program will not get stuck on a type error. Dependent types can provide stronger static guarantees about the behaviour of concurrent program, including the absence of race conditions and deadlocks (Brady and Hammond, 2010).

Dependent types are a very powerful programming technique that allows the dynamic behaviour of the program to be encoded within the type system — programs written in dependently typed languages become proofs of their own correctness.

In a dependently typed programming language types become first-class values, allowing a developer to define arbitrary-specific conditions within types. As the programmer wants more complicated properties to be verified the more complex proof obligations the programmer needs to fill. Type inference in a dependently typed programming language becomes undecidable, causing the programmer to write a lot of type annotations simply to satisfy the type checker.

Refinement types are more limited form of dependent types that attach boolean predicate to types. Refinement types are restricted to decidable predicates as their main focus is automatic verification via an external solver, therefore lessening the proof obligations on the developer.

Hybrid type checking (Flanagan, 2006; Knowles and Flanagan, 2010) is a breakthrough idea that allows for refinements with expressive and potentially undecidable predicates; however hybrid type checking restricts refinements to base types as it uses denotational interpretation of types to avoid circularity within typing judgements.
Blame calculus (Wadler and Findler, 2009) uses explicit casts and explicitly tagged refinement-typed values, while blame tracking aids reasoning about runtime exceptions. Existing blame calculi omit dependent functions and fail to show blame safety in some simple cases, handled by hybrid type checking.

We want refinements of arbitrary types to express and reason about specifications (Swamy et al., 2013), and we want blame tracking at higher fidelity to show safety for all casts that will never lead to blame.

1.1 Contributions

The contribution of this thesis are:

- a dependently-typed blame calculus, $\lambda dB$, with dependent function spaces and refinements over arbitrary types.

- an improved notion of blame safety analysis based on the novel presentation of subtyping with a witness.

- proofs of metatheoretical properties. In particular, preservation and progress for $\lambda dB$, and the blame theorem for extended blame analysis.
Chapter 2

Background

An (untrapped) type error represents wrong and undesired behaviour of a program. An untrapped type error in lambda calculus is represented by a term getting “stuck” — the term is not a value, but it cannot evaluate further. An untrapped type error is more severe in real-life applications, as the error might manifest themselves in unexpected places. For simplicity, I will focus on untrapped type errors in lambda calculus.

Sound type systems prevent untrapped type errors by restricting the set of programs evaluated by an interpreter (or a compiler) to programs that can be statically determined to never get stuck on a type error and, as a consequence, rejecting all programs that may get stuck on a type error.

As the type system becomes more powerful, it allows a developer to specify more detailed constraints, which in turn provide greater guarantees to the programmer. With more precise specifications, however, it becomes more difficult to write programs that will be accepted by a type checker. In particular, any specification that is either undecidable or ambiguous should be rejected by a sound static type checker.

Recent research in programming languages aims to address the aforementioned aspect by deferring undecidable properties until runtime and verifying them dynamically either by providing more flexible system with statically decidable constraints (Ou et al., 2004; Tanter and Tabareau, 2015) or by providing a system that can deal with ambiguous specifications by verifying them during runtime (Knowles and Flanagan, 2010; Greenberg et al., 2010).


2.1 Dependent Types

Just as types in a non-dependent type systems verify programs against a set of static specifications, dependent types in a dependent type systems verify programs against a set of specifications that encode the dynamic behaviour of the program.

For instance consider the code for concatenation in a non-dependently typed functional programming language, Haskell, (taken from GHC (2015))

\[
(\vdash : [a] \rightarrow [a] \rightarrow [a])
\]

\[
(\vdash []) \rightarrow ys = ys
\]

\[
(\vdash (x:x) \rightarrow ys = x : xs ++ ys)
\]

with concatenation in a dependently typed language, Agda, (taken from Abel (2009)).

\[
append : \{A : Set\}{n m : Nat} \rightarrow
\]

\[
\quad Vect A n \rightarrow Vect A m \rightarrow
\quad Vect A (m + n)
\]

\[
append vnil ys = ys
\]

\[
append (vcons x xs) ys =
\quad vcons x (append xs ys)
\]

Note that, superficially, both functions perform the same operation, but the dependently typed version encodes additional guarantee that the result’s size should be the size of the two input lists. Consequently, an incorrect implementation which simply returns the empty list will be accepted by the first specification, but rejected by the second.

2.1.1 Refinement Types

Freeman and Pfenning (1991) introduced refinement types which allow expand the types with binary predicates that express the specification of the program. We write refinement types using the syntax of Swamy et al. (2013):

\[
(x : B\{P\})
\]

for some type \(B\) and some boolean predicate acting on \(x\) of type \(B\).

For instance, we can define a type for positive integers:

\[
(x : \text{int}\{x > 0\}).
\]
2.1.2 Dependent Functions

Dependent function spaces (Martin-Löf, 1998) allow for the return type to depend on the value of the argument. We write dependent functions using the syntax of Swamy et al. (2013):

\[(x : A) \to B\]

for some types \(A\) and \(B\) and some \(x\) of type \(A\) bound in \(B\).

For instance we can specify the type for strictly increasing (dependent) functions over integers:

\[(x : \text{int}) \to (y : \text{int})\{y > x\}.

or the append for vectors, assuming that \(\text{Vec}\) is type of vectors:

\[(x : \text{Vec}) \to (y : \text{Vec}) \to (\text{len} : \text{Vec} \to \text{int}) \to (z : \text{Vec})\{\text{len}(z) = \text{len}(x) + \text{len}(y)\}

2.2 Dynamically Checked Decidable Predicates

Ou et al. (2004) introduced a notion of dynamic typing with dependent types. The authors provided a language that would annotate regions of the program as either simply-typed or dependently-typed and the compiler would insert additional checks during runtime at the boundary between simply-typed and dependently typed code to verify the predicates specified in the dependently typed program. Their system was ground-breaking in that fewer proof burdens were placed on the developer.

In order to ensure that their type checking is decidable, Ou et al. (2004) restrict predicates in their refinements to be drawn from decidable logic and provide a set of axioms for the theorem prover — which restricts the expressiveness of their predicates.

2.3 Hybrid Type Checking

Flanagan (2006); Knowles and Flanagan (2010) introduced hybrid type checking which handles undecidable predicates by deferring some verification until runtime, but resorts to denotational interpretation of refinement types to avoid circularity in the typing judgements. Furthermore, it relies on general beta-reduction to simplify its meta-theory. Greenberg et al. (2010) fix the order of evaluation, but they
still rely on denotational interpretation of refinement types. Finally, Thiemann (2016) shows how to simplify redundant predicates, but restricts predicates to be drawn from a decidable logic, and has 3 different rules for applying functions under a cast, complicating his system. All of these systems restrict refinements to base types.

2.4 Gradual Typing

Siek and Taha (2006) introduced gradual typing that allows for combining the advantages from two seemingly separate words of statically and dynamically typed worlds. Gradual typing allows for adding type annotations to portions of the code that will be verified statically, whereas it defers the check of the remaining portions of the program to the runtime.

Type compatibility (also referred to as type consistency) is a new relation introduced by Siek and Taha (2006) to allow for casts between statically and dynamically typed code. Type $A$ may be cast to type $B$ if they are compatible with each other, written as

$$A \sim B.$$  

The compatibility relation is different from subtyping relation in that compatibility is symmetric, but not transitive, while subtyping relation is transitive and anti-symmetric. The symmetry of compatibility relation allows for transitioning between more-precisely typed code and less-precisely typed code, while subtyping permits only transition from more-precisely typed to less-precisely typed.

Gradual Certified Programming

Recently, Tanter and Tabareau (2015) introduced a library for gradual typing in Coq proof assistant. They also restrict their system to decidable propositions, but for simplicity they forego blame tracking — consequently it is not possible to verify which portion of the code is responsible for a failed cast and therefore ensure that the error lies within the less-typed portion of the code.
2.5 Blame Calculus

Wadler and Findler (2009) introduced the blame calculus, which extends gradual typing (Siek and Taha, 2006) — which seamlessly mixes more-precisely typed code and less-precisely typed code — with blame tracking of contracts (Findler and Felleisen, 2002) and proves the Blame Theorem, that a failed cast between types will always blame the less-precisely typed code, by showing blame safety which guarantees that a term will never evaluate to a particular type of blame.

2.5.1 Casts

Blame calculus integrates the less-precisely typed code and more precisely typed code via explicit casts:

\[ x : A \xrightarrow{p} B \]

for some \( x \) of type \( A \), some type \( B \) (compatible with \( A \)), and some blame label \( p \). Blame label are unique references to the locations of the program \( l \); blame label can be either positive or negative.

2.5.2 Blame

A failed cast evaluates to term

\[ \text{blame} \ p \]

where \( p \) was the label annotating the cast.

If \( p \) was a positive blame label, then the term enclosed in the cast annotated with \( p \) caused the cast to fail, whereas if \( p \) was a negative blame label, then the context in which the term was used caused the cast to fail.

2.5.3 Blame Theorem

For instance, a program containing the following cast from \( A \) to \( B \) will never evaluate to \text{blame} \ p \ if \( A \prec B \) we assume \( x : A \).

While blame calculus omits dependent functions, both Thiemann (2016) and Knowles and Flanagan (2010) omit blame tracking; Greenberg et al. (2012) introduce blame tracking, but do not complement their blame labels (Wadler, 2015). The lack of blame tracking or complement to blame in those systems prevents formulating blame theorem and showing blame safety.
Chapter 3

Dependent Blame Calculus ($\lambda dB$)

We present dependently typed blame calculus, $\lambda dB$, which integrates dependently-typed code and simply-typed code using casts, and incorporates refinements over arbitrary types and dependent functions for dependent-typing.

We present $\lambda dB$ in the form of compile-time language (Figure 3.1) without refinement-typed terms, but with decidable type-checking, and in the form of run-time language (Figure 3.6) with refinement-typed values. The run-time language is an extension of the compile-time language, therefore the embedding of the compile-time language in the run-time language is straightforward. We grey out the parts of the compile-time language when presenting the rules for run-time language.

3.1 Compile-time language

$\lambda dB$ is influenced by $\lambda^H$ of Knowles and Flanagan (2010), blame calculus of Wadler and Findler (2009), and the syntax of refinement types of Swamy et al. (2013).

Let $\iota$ range over base types. A base type is either a type for integers $\text{int}$ or a type for booleans $\text{bool}$.

Let $A, B, C, D$ range over refinement types. A refinement type is either a base type $\iota$, a refinement $(x : A)\{P\}$, or a dependent function $(x : A) \rightarrow B$. We generalise refinement types to be specified over arbitrary types, not just base types.

Let $p, q$ range over blame labels. A blame label is either positive blame label from an unique location $+l$ or a negative blame label from a location $-l$. We use
the standard notion of a blame label with an involutive negation on the blame labels:

\[-(+l) = -l\]

\[-(-l) = +l.\]

Let \(P, Q\) range over predicates which are terms of the language.

Let \(L, M, N\) range over terms. Terms are either constants \(c\), built-in operators \(op(M)\), variables \(x\), dependent lambda abstractions \(\lambda x:A. N\), applications of dependent functions to values \(LV\), non-dependent let bindings \(\text{let } x = M \text{ in } N\), if-then-else expressions \(\text{if } N \text{ then } M \text{ else } L\), casts on values \(V : A \xrightarrow{p} B\), and blame \(\text{blame } p\). We restrict the applications of functions to values, similarly to Swamy et al. (2013) for value-dependency, and we restrict the applications of casts to values.

Let \(V, W\) range over values. Values are either constants \(c\), variables \(x\), lambda abstractions \(\lambda x:A. N\), or values wrapped in a function cast \(V : (x : A) \xrightarrow{p} (y : C) \rightarrow D\).

Let \(\Gamma\) range over environments. Environments are either empty environments \(\cdot\), extensions with a name binding to a type \(\Gamma, x : A\), or extensions with a name binding to a term \(\Gamma, x = M\). Our environments are similar to those of Knowles et al. (2006), but we permit bindings to arbitrary terms, whereas Knowles et al. restrict their bindings to values.
### 3.1.1 Compile-time Typing

We write $\Gamma \vdash M : A$ (Figure 3.2) to indicate that in the environment $\Gamma$ term $M$ has type $A$, and we write $\Gamma \vdash A : \text{tp}$ (Figure 3.4) to indicate that in the environment $\Gamma$ type $A$ is well-formed. We provide an additional judgement $\Gamma \text{ctx}$ (Figure 3.5) to indicate that environment $\Gamma$ a valid typing context and contains only well-formed types.

![Typing Rules](image)

Each constant $c$ has a type defined by $\text{type}(c)$. Each operator built-in operator $\text{op}$ is defined by a total meaning function $[[\text{op}]]$ that preserves types (types of all the operands are the same as the type of result).

Lambda abstractions are dependently typed and the argument is bound in
Chapter 3. Dependent Blame Calculus ($\lambda dB$)

the result type:

\[ \Gamma, x : A \vdash N : B \]

\[ \Gamma \vdash (\lambda x : A. N) : (x : A) \rightarrow B \]

We restrict our applications to values as value-dependency is a well-understood technique to reason about side-effects (Swamy et al., 2013); since the function argument is bound in the result type $B$ we substitute $x$ for the argument $V$ in the result type $B$

\[ \Gamma \vdash L : (x : A) \rightarrow B \quad \Gamma \vdash V : A \]

\[ \Gamma \vdash LV : B[x := V] \]

We restrict our let-bindings to be non-dependent via the additional clause $\Gamma \vdash B : tp$ which ensures that $B$ does not depend on $x$.

\[ \Gamma \vdash M : A \quad \Gamma, x = M \vdash N : B \quad \Gamma \vdash B : tp \]

\[ \Gamma \vdash (let \ x = M \ in \ N) : B \]

By restricting our function applications to be value-dependent and by restricting our let-bindings 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.

\[
\begin{array}{|c|c|c|c|}
\hline
A \sim B & A \sim (x : B)\{P\} & A \sim B & C \sim A & B \sim D \\
\hline
\sim \sim  & A \sim (x : B)\{P\} & (x : A)\{P\} \sim B & (x : A) \rightarrow B \sim (y : C) \rightarrow D \\
\hline
\end{array}
\]

Figure 3.3: Compatibility for types.

We restrict our casts to values which will facilitate blame analysis in Chapter 4. We write $A \sim B$ to indicate that type $A$ is compatible with type $B$ and we extend the standard notion of to include dependent functions (Figure 3.3). We further restrict our casts to be defined between well-formed types, via via the additional clause $\Gamma \vdash B : tp$ which ensures that the target type is well formed

\[ \Gamma \vdash M : A \quad \Gamma \vdash A \sim B \quad \Gamma \vdash B : tp \]

\[ \Gamma \vdash (V : A \xrightarrow{p} B) : B \]
We restrict our blame terms to have only well-formed types and occur only in well-formed environments

$$
\Gamma_{ctx} \quad \Gamma \vdash A : \text{tp} \\
\Gamma \vdash \text{blame } p : A
$$

**Well-Formed Types**

![Well-formed types](image)

Figure 3.4: Well-formed types.

All base types are well-formed within a valid typing context as they contain no-predicates

$$
\Gamma_{ctx} \\
\Gamma \vdash \iota : \text{tp}
$$

Refinements are well-formed within a valid typing context if they are specified over a well-formed type $A$ and their predicate may contain additional binding for $x$ of type $A$

$$
\Gamma_{ctx} \quad \Gamma \vdash A : \text{tp} \quad \Gamma, x : A \vdash P : \text{bool} \\
\Gamma \vdash (x : A) \{P\} : \text{tp}
$$

Dependent functions are well-formed within a valid typing context if the argument type $A$ is well formed within a valid typing context and the result type $B$ is well-formed within a valid context extended with an $x$ of type $A$

$$
\Gamma_{ctx} \quad \Gamma \vdash A : \text{tp} \quad \Gamma, x : A \vdash B : \text{tp} \\
\Gamma \vdash (x : A) \rightarrow B : \text{tp}
$$

**Valid Contexts**

Instead of implicitly requiring that every environment is a valid typing context we provide an explicit judgement for valid contexts.
Chapter 3. Dependent Blame Calculus ($\lambda dB$)

<table>
<thead>
<tr>
<th>(·) ctx</th>
<th>$\Gamma$ ctx</th>
<th>$\Gamma \vdash A : \text{tp}$</th>
<th>$\Gamma$ ctx</th>
<th>$\Gamma \vdash M : A$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>($\Gamma, x : A$) ctx</td>
<td></td>
<td>($\Gamma, x = M$) ctx</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.5: Context validity judgement

An empty context is trivially valid

$$
(\cdot) \text{ctx}.
$$

An extension of a valid context with a binding to a type which is well-formed in such context is valid

$$
\Gamma \text{ctx} \quad \Gamma \vdash A : \text{tp} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (\Gamma, x : A) \text{ctx}.
$$

An extension of a valid context with a binding to a term that which is typeable in such context is valid

$$
\Gamma \text{ctx} \quad \Gamma \vdash M : A \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (\Gamma, x = M) \text{ctx}.
$$

3.2 Run-time Language

During run-time we extend our language with constructs to handle refinement-typed values.

Let terms $L, M, N$ be extended with explicitly tagged refinement-typed values $V_{(x: A)\{P\}}$ and casts-in-progress $\Delta b^R V_{(x: A)\{P\}}$.

Let values $V, E$ be extended with explicitly tagged refinement-typed values $V_{(x: A)\{P\}}$.

We write $\mathcal{E}$ to denote evaluation contexts which are standard evaluation contexts extended with an evaluation context for cast-in-progress $\Box b^R V_{(x: A)\{P\}}$ which allows the predicate within cast-in-progress to evaluate.

3.2.1 Run-time Typing

We extend the typing judgement to handle refinement-typed terms during run-time.
### 3.2. Run-time Language

| **Base Types** | \( \iota ::= \text{bool} \mid \text{int} \) |
| **Refinement Types** | \( A, B, C, D ::= \iota \mid (x: A)\{P\} \mid (x: A) \rightarrow B \) |
| **Predicates** | \( P, Q ::= M \) |
| **Terms** | \( L, M, N ::= c \mid \text{op}(\vec{M}) \mid x \mid \lambda x: A. N \mid L V \mid \text{let } x = M \text{ in } N \mid \text{if } N \text{ then } M \text{ else } L \mid V : A \xrightarrow{P} B \mid \text{blame } p \mid \vec{V}_{(x:A)}\{P\} \mid Q \xrightarrow{p} \vec{V}_{(x:A)}\{P\} \) |
| **Blame labels** | \( p, q ::= +l \mid -l \) |
| **Environments** | \( \Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, x = M \) |
| **Values** | \( V, W ::= c \mid x \mid \lambda x: A. N \mid \vec{V} : (x : A) \rightarrow B \xrightarrow{p} (y : C) \rightarrow D \mid \vec{V}_{(x:A)}\{P\} \) |
| **Evaluation Contexts** | \( \mathcal{E} ::= \Box \mid \text{op}(\vec{V}, \Box, \vec{M}) \mid \Box V \mid \text{let } x = \Box \text{ in } M \mid \text{if } \Box \text{ then } M \text{ else } L \mid \Box \xrightarrow{p} \vec{V}_{(x:A)}\{P\} \) |

*Figure 3.6: Run-time language, \( \lambda dB \)*
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma, x : A \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, x : A \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, y : B \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, y : B \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, x : A \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, x : A \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, y : B \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, y : B \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, x : A \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, x : A \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, y : B \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, y : B \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, x : A \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, x : A \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, y : B \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, y : B \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, x : A \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, x : A \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, y : B \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, y : B \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, x : A \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, x : A \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, y : B \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, y : B \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, x : A \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, x : A \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, y : B \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, y : B \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, x : A \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, x : A \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, y : B \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, y : B \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, x : A \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, x : A \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, y : B \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, y : B \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, x : A \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, x : A \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, y : B \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, y : B \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, x : A \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, x : A \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, y : B \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, y : B \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, x : A \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, x : A \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, y : B \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, y : B \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, x : A \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, x : A \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, y : B \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, y : B \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, x : A \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, x : A \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, y : B \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, y : B \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, x : A \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, x : A \vdash \lambda x : A. M$</td>
</tr>
<tr>
<td>$\Gamma, y : B \vdash \lambda x : A. M : B$</td>
<td>$\Gamma \vdash M : A$ and $\Gamma, y : B \vdash \lambda x : A. M$</td>
</tr>
</tbody>
</table>

Figure 3.7: Typing rules for target language.
Explicitly tagged refinement-typed values are values that do not take the predicate of the refinement to \texttt{false}. We prevent stuck predicates via clause 

\[ \cdot \vdash V : A \] 

which ensures that value is closed in the empty typing context, and via clause 

\[ \cdot \vdash (x : A)\{P\} \] 

which ensures that predicate \( P \) does not refer to any variables other than \( x \) of type \( A \)

\[
\Gamma \text{ctx} \quad \cdot \vdash V : A \quad \cdot \vdash (x : A)\{P\} : \text{tp} \quad P[x := V] \not\rightarrow^* \text{false}
\]

\[ \Gamma \vdash (V_{(x : A)\{P\}}) : (x : A)\{P\} \]

Casts-in-progress verify the predicate for value that is cast to a refinement type. We provide two typing rules for casts-in-progress to type cast-in-progress within a valid context and within empty context. Within arbitrary valid context we restrict casts-in-progress to predicates on the left-hand side to match the predicate of the refinement.

\[
\Gamma \text{ctx} \quad \Gamma \vdash P : \text{bool} \quad \Gamma \vdash V : A \quad \Gamma \vdash (x : A)\{P\} : \text{tp}
\]

\[ \Gamma \vdash (P^{\cdot}V_{(x : A)\{P\}}) : (x : A)\{P\} \]

Within empty context we restrict the predicate on the left-hand side via the additional clause 

\[ P[x := V] \not\rightarrow^* Q \] 

which ensures that \( Q \) is a result of reducing \( P \) with the value \( V \) substituted for \( x \).

\[
\cdot \vdash Q : \text{bool} \quad \cdot \vdash V : A \quad \cdot \vdash (x : A)\{P\} : \text{tp} \quad P[x := V] \not\rightarrow^* Q
\]

\[ \cdot \vdash (Q^{\cdot}V_{(x : A)\{P\}}) : (x : A)\{P\} \]

Since an empty context is a valid context it may seem that the typing rules overlap the rules may overlap if \( x \notin \text{FV}(P) \). In that case our typing rules are coherent as after applying the typing rule for casts-in-progress within arbitrary typing context \( \cdot \text{ctx} \) holds trivially, while after applying the rule for empty contexts, from \( x \notin \text{FV}(P) \) we know that \( P[x := V] = P \), so \( P \not\rightarrow^* P \) holds trivially.

### 3.2.2 Soundness and Completeness of Typing Judgements

We show that our judgement for well-formed types is sound and complete with respect to typing judgement for terms: we show that every typeable term in our language has a well-formed type and we show that every well-formed type is inhabited.

**Proposition 1** (Every typeable term has a well-formed type). If \( \Gamma \vdash M : A \) then \( \Gamma \vdash A : \text{tp} \).
Chapter 3. Dependent Blame Calculus ($\lambda dB$)

**Proof.** By induction on the typing derivation. Detailed proof can be found in Appendix A. □

**Proposition 2** (Every well-formed type is inhabited). If $\Gamma \vdash A : tp$, then there exists $M$ such that $\Gamma \vdash M : A$.

**Proof.** blame $p$ inhabits all well-formed types. □

We also show that terms that do not contain the blame term have unique types.

**Proposition 3** (Unicity of non-blame terms). If $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$ and blame $p \notin M$ then $A = B$.

**Proof.** By induction on the typing derivation of $\Gamma \vdash M : A$. □

### 3.3 Dynamic Semantics

We define reduction rules for $\lambda dB$ in terms of reductions on terms and reductions on configurations (Figure 3.8). We fix the order of evaluation to call-by-value and we provide show that our language is deterministic and satisfies the diamond property.

Our reduction rules are similar to that of Wadler and Findler (2009) with one exception, for the evaluation of function casts we add two fresh identifiers $e$ and $f$ to avoid any accidental variable capture.

### 3.4 Type Safety

**Lemma 4** (Substitution). Substitution preserves typing for terms and types:

- If $\Gamma \vdash V : A$ and $\Gamma, x : A \vdash M : B$, then $\Gamma \vdash M[x := V] : B$.
- If $\Gamma \vdash V : A$ and $\Gamma, x : A \vdash B : tp$, then $\Gamma \vdash B[x := V] : tp$.

**Proof.** Proof by mutual induction on the structure of derivation of $\Gamma, x : A \vdash M : B$ and $\Gamma, x : A \vdash B : tp$. Detailed proof can be found in Appendix A. □

**Lemma 5** (Canonical forms). Let $V$ be a value that is well typed in the empty context then:
Reduction on terms

\[\text{op}(\vec{V}) \rightarrow \text{\llbracket \text{op} \rrbracket}(\vec{V})\]

\[(\lambda x : A. N) V \rightarrow N[x := V]\]

\[\text{let } x = V \text{ in } N \rightarrow N[x := V]\]

if true then \(M\) else \(L\) \(\rightarrow M\)

if false then \(M\) else \(L\) \(\rightarrow L\)

\(V : A \overset{p}{\Rightarrow} A \rightarrow V\)

\(V : A \overset{p}{\Rightarrow} (y : B)(Q) \rightarrow \text{let } y = V : A \overset{p}{\Rightarrow} B \text{ in } (Q \overset{p}{\Rightarrow} y : B)(Q)\)

\((V(x : A)(P)) : (x : A)(P) \overset{p}{\Rightarrow} B \rightarrow V : A \overset{p}{\Rightarrow} B\)

\((V : (x : A) \rightarrow B \overset{p}{\Rightarrow} (y : C) \rightarrow D) W \rightarrow \text{let } e = W : C \overset{\bar{p}}{\Rightarrow} A \text{ in}\)

\[\text{let } f = (V e) \text{ in}\]

\[f : B[x := e] \overset{p}{\Rightarrow} D[y := W]\]

where

\(e \notin FV(V, (x : A) \rightarrow B, (y : C) \rightarrow D)\), and

\(f \notin FV((x : A) \rightarrow B, (y : C) \rightarrow D)\)

\[\text{true} \overset{p}{\Rightarrow} V(x : A)(P) \rightarrow V(x : A)(P)\]

\[\text{false} \overset{p}{\Rightarrow} V(x : A)(P) \rightarrow \text{blame } p\]

Reduction on configurations

\[
\begin{array}{c}
\frac{M \rightarrow N}{\mathcal{E}[M] \rightarrow \mathcal{E}[N]} \\
\frac{\mathcal{E} \neq \Box}{\mathcal{E}[\text{blame } p] \rightarrow \text{blame } p}
\end{array}
\]

Figure 3.8: Reductions
− If ⊢ V : τ, then V = c with type(c) = τ.

− If ⊢ V : (x : A)\{P\}, then V = W_{(x:A)}\{P\} with ⊢ W : A and P[x = W] \not\rightarrow^* false.

− If ⊢ V : (x : A) → B then either
  + V = λx:A. N with x : A ⊢ N : B, or
  + V = W : (y : C) → D \xrightarrow{p} (x : A) → B with ⊢ W : (y : C) → D.

Proof. Follows from case analysis.

**Proposition 6** (Preservation). If Γ ⊢ M : A and M \rightarrow N then Γ ⊢ N : A.

Proof. By cases on the reduction sequence. Detailed proof can be found in Appendix A.

**Proposition 7** (Progress). If ⊢ M : A then either:

− M is a value.

− M → N for some term N.

− M is blame p for some blame label p.

Proof. By induction on the typing derivation. Detailed proof can be found in Appendix A.

### 3.4.1 Diamond Property

**Proposition 8** (Diamond property). If Γ ⊢ M : A and M \rightarrow N and M \rightarrow N', then either N = N' or there exists L such that such that N → L and N' → L and Γ ⊢ L : A.

Proof. The only interesting case is

\[ Γ \vdash ((V_{(x:A)}\{P\}) : (x : A)\{P\} \xrightarrow{p} (y : B)\{Q\}) : (y : B)\{Q\} \]

which can be either evaluated by rule 7 or rule 8:
– by first applying reduction rule 7, then by applying the reduction rule 8:

\[
\begin{align*}
(V_{x:A}\{P\}) : (x : A)\{P\} &\xrightarrow{p} (y : B)\{Q\} \\
\xrightarrow{} \text{let } y = (V_{x:A}\{P\}) : (x : A)\{P\} &\xrightarrow{p} B \text{ in } (Q \triangleright^p y_{(y:B)}\{Q\}) \\
\xrightarrow{} \text{let } y = V &\xrightarrow{p} B \text{ in } (Q \triangleright^p y_{(y:B)}\{Q\})
\end{align*}
\]

– by first evaluating via rule 8, then by applying the reduction rule 7:

\[
\begin{align*}
(V_{x:A}\{P\}) : (x : A)\{P\} &\xrightarrow{p} (y : B)\{Q\} \\
\xrightarrow{} V : A &\xrightarrow{p} (y : B)\{Q\} \\
\xrightarrow{} \text{let } y = V &\xrightarrow{p} B \text{ in } (Q \triangleright^p y_{(y:B)}\{Q\})
\end{align*}
\]

\[\Gamma \vdash \text{let } y = V \xrightarrow{p} B \text{ in } (Q \triangleright^p y_{(y:B)}\{Q\}) \text{ holds via preservation (Proposition 6).} \]

\[\square\]
Chapter 4

Blame Safety at Higher Fidelity

The ordinary subtyping can only show that the following program is safe for the first cast (Wadler and Findler, 2009), while our approach shows both casts are safe:

\[
\begin{align*}
&\text{let } z = y : (x : \text{int})\{x > 0\} \Rightarrow \text{int} \text{ in} \\
&z : \text{int} \Rightarrow (x : \text{int})\{x > 0\}
\end{align*}
\]

we assume \(y : (x : \text{int})\{x > 0\}\).

We show that thanks to our novel presentation our subtyping rules for refinement types over arbitrary types are coherent, reflexive, and transitive. We also note that previous work on hybrid type checking required an reflexivity axiom for arbitrary refinements, whereas we show that our rules are reflexive in an absence of such axiom.

Our new presentation of subtyping permits more fine-grained blame safety analysis, i.e. blame tracking at higher fidelity. Our contributions to blame safety analysis are as follows:

- We introduce a novel presentation of subtyping, in which the subtyping judgement is a three-way relation among the value \(V\), its type \(A\), and a supertype \(B\).

\[
\Gamma \vdash V : A \ll B
\]

where \(V\) witnesses subtyping from \(A\) to \(B\) (Section 4.1). We prove that our presentation of subtyping is coherent, reflexive, and transitive. We use it to prove the blame theorem for \(\lambda dB\).

- We use a different notion of semantic judgement (Section 4.1.1), inspired by Keil and Thiemann (2015), that allows us to define positive subtyping
for refinements over arbitrary cases, and we show how it handles cases in which previous definition of semantic judgement fell short.

– Finally, we provide an improved notion of blame safety that permits blame tracking at higher-fidelity (Section 4.3). Furthermore we show that we avoid the circularity present in hybrid type systems by forgoing subsumption in favour of explicitly tagged casts — we use subtyping only in the blame safety analysis.

### 4.1 Subtyping with a Witness

The standard method in the literature is to provide separate judgement for subtyping and a separate judgement for subsumption, while we opt for explicit casts between types and extending our subtyping judgements with the notion of witness. A witness is the value the subtyping judgement will be performed on.

\[
\begin{align*}
\Gamma \vdash V : \iota
\end{align*}
\]

\[
\begin{align*}
\frac{
\Gamma \vdash V : \iota}
{\Gamma \vdash V : \iota : \iota}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash V : (x : A) \rightarrow B
\quad
\Gamma \vdash (y : C) \rightarrow D : \text{tp}
\quad
\Gamma, g : C \vdash g : C : A
\quad
\Gamma, g : C, e = (g : C \rightarrow A), f = (V \ x) \vdash f : B[x := e] : D[y := g]
\quad
e, g \notin \text{FV}(V, (x : A) \rightarrow B, (y : C) \rightarrow D)
\quad
f \notin \text{FV}((x : A) \rightarrow B, (y : C) \rightarrow D)
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash V : (x : A) \rightarrow B : (y : C) \rightarrow D
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash V : (x : A)\{P\}
\quad
\Gamma, f = (V : (x : A)\{P\} \Rightarrow A) \vdash f : A : B
\quad
\Gamma \vdash B : \text{tp}
\quad
f \notin \text{FV}(A, B)
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash V : (x : A)\{P\} : B
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash V : A : B
\quad
\Gamma \vdash (y : B)\{Q\} : \text{tp}
\quad
\Gamma, y = (V : A \Rightarrow B) \models Q
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash V : A : (y : B)\{Q\}
\end{align*}
\]

Figure 4.1: Subtyping with a witness.

We write \(\Rightarrow\) to indicate that a choice of blame label does not matter.
For rule 3 and 4 of our notion of subtyping with a witness we introduce fresh identifiers \( e, f, \) and \( g \) to avoid accidental variable capture.

### 4.1.1 Semantic Judgement

We use a notion of semantic judgement inspired by Keil and Thiemann (2015), where instead of requiring that a predicate has to evaluate to \texttt{true}, we require that it does not evaluate to \texttt{false}. For the runtime representation of refinement-typed values we additionally restrict that the predicate is closed via the following clause (Figure 3.7)

\[ \cdot \vdash (x : A)\{P\} : \text{tp}. \]

Without the above restriction any value would satisfy any open term.

\[
\begin{array}{c}
\Gamma \vdash P : \text{bool} \\
\forall \sigma. (\Gamma \vdash \sigma \circ \sigma(P) \not\to^{*} \texttt{false}) \\
\hline
\Gamma \vdash P
\end{array}
\]

Figure 4.2: Semantic Judgement

Our definition of semantic modelling is simpler than that of Knowles and Flanagan (2010) and (Greenberg et al., 2012), while allowing to semantically model non-terminating functions. Although a non-terminating function will trivially satisfy any predicate, the \textit{cast-in-progress} operator will never evaluate to a runtime representation of such-refinement typed function, so we do not consider that a problem in our system.

**Order-Preserving Closing Substitution**

Flanagan (2006) define the notion of a \textit{consistent substitution} that models the environment by binding arbitrary well-typed values to matching identifiers in the environment, then they define that subtyping relation holds whenever for all possible closing substitutions the predicate of subtype implies the predicate of the supertype. However, their approach suffers from circular typing judgement in the negative position of an implication.

We define \textit{order-preserving closing substitutions}, that a similar to \textit{consistent substitution} of Flanagan (2006), but do not suffer from circularity in typing rules
as we use closing substitutions during blame safety analysis. Thanks to our notion of extended environment our closing substitutions take into account the previous history of casts on the value.

\[
\begin{array}{c}
\Gamma \vdash V : A \\
\sigma \in \Gamma \\
\Gamma, x : A \vdash \sigma, x = V \\
\Gamma, x = M \vdash \sigma, x = M
\end{array}
\]

Figure 4.3: Closing substitution.

We give semantics of applying closing substitutions in terms of let-bindings to preserve the order of effects (both blame and non-termination) that are associated with names bound in the environment.

\[
\begin{align*}
\sigma(P) &= P \\
(\sigma, x = M)(P) &= (\sigma)(\text{let } x = M \text{ in } P)
\end{align*}
\]

Figure 4.4: Applying order-preserving closing substitutions

We present positive and negative subtyping with a witness. Positive subtyping indicates that a cast from a positive subtype to positive supertype will never yield positive blame, whereas negative subtyping indicates that a cast from a negative subtype to a negative supertype will never yield negative blame.

We present a notion of naive subtyping with a witness (Figure 4.6), which for dependent functions is both covariant on the argument type and covariant on the range type. Naive subtyping indicates type precision, a naive subtype is a more precise type than a naive supertype. A cast from naive subtype to naive supertype will never blame the source type.
4.1. Subtyping with a Witness

<table>
<thead>
<tr>
<th>Positive subtyping with a witness</th>
<th>$\Gamma \vdash V : A &lt;^{+} B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash V : \iota$</td>
<td>$\Gamma \vdash V : \iota &lt;^{+} \iota$</td>
</tr>
<tr>
<td>$\Gamma \vdash V : (x : A) \rightarrow B$</td>
<td>$\Gamma \vdash (y : C) \rightarrow D : \text{tp}$</td>
</tr>
<tr>
<td>$\Gamma, g : C \vdash g : C &lt;^{-} A$</td>
<td>$\Gamma \vdash (y : C) \rightarrow D : \text{tp}$</td>
</tr>
<tr>
<td>$\Gamma, g : C, e = (g : C \Rightarrow A), f = (V x) \vdash f : B[x := e] &lt;^{+} D[y := g]$</td>
<td>$\Gamma \vdash (y : C) \rightarrow D : \text{tp}$</td>
</tr>
<tr>
<td>$e, g \notin \text{FV}(V, (x : A) \rightarrow B, (y : C) \rightarrow D)$</td>
<td>$f \notin \text{FV}(x : A) \rightarrow B, (y : C) \rightarrow D$</td>
</tr>
<tr>
<td>$\Gamma \vdash V : (x : A) \rightarrow B &lt;^{+} (y : C) \rightarrow D$</td>
<td>$\Gamma \vdash V : (x : A) \rightarrow B &lt;^{+} (y : C) \rightarrow D$</td>
</tr>
<tr>
<td>$\Gamma, f = (V : (x : A){P} \Rightarrow A) \vdash f : A &lt;^{+} B$</td>
<td>$\Gamma, g = (V : A \Rightarrow B) \vdash g : B$</td>
</tr>
<tr>
<td>$\Gamma \vdash V : (x : A){P} &lt;^{+} B$</td>
<td>$\Gamma, g = (V : A \Rightarrow B) \vdash g : B$</td>
</tr>
<tr>
<td>$\Gamma \vdash V : A &lt;^{+} B$</td>
<td>$\Gamma, f = (V : (x : A){Q} \Rightarrow B) \vdash f : B$</td>
</tr>
<tr>
<td>$\Gamma \vdash (y : B){Q} : \text{tp}$</td>
<td>$\Gamma, f = (V : (x : A){Q} \Rightarrow B) \vdash f : B$</td>
</tr>
<tr>
<td>$\Gamma \vdash V : A &lt;^{+} (y : B){Q}$</td>
<td>$\Gamma \vdash V : A &lt;^{+} (y : B){Q}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Negative subtyping with a witness</th>
<th>$\Gamma \vdash V : A &lt;^{-} B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash V : \iota$</td>
<td>$\Gamma \vdash V : \iota &lt;^{-} \iota$</td>
</tr>
<tr>
<td>$\Gamma \vdash V : (x : A) \rightarrow B$</td>
<td>$\Gamma \vdash (y : C) \rightarrow D : \text{tp}$</td>
</tr>
<tr>
<td>$\Gamma, f = (V : (x : A){P} \Rightarrow A) \vdash f : A &lt;^{-} B$</td>
<td>$\Gamma \vdash (y : C) \rightarrow D : \text{tp}$</td>
</tr>
<tr>
<td>$\Gamma \vdash V : (x : A){P} &lt;^{-} B$</td>
<td>$\Gamma \vdash (y : C) \rightarrow D : \text{tp}$</td>
</tr>
<tr>
<td>$\Gamma \vdash V : A &lt;^{-} B$</td>
<td>$\Gamma, f = (V : (x : A){Q} \Rightarrow B) \vdash f : B$</td>
</tr>
<tr>
<td>$\Gamma \vdash (y : B){Q} : \text{tp}$</td>
<td>$\Gamma, f = (V : (x : A){Q} \Rightarrow B) \vdash f : B$</td>
</tr>
<tr>
<td>$\Gamma \vdash V : A &lt;^{-} (y : B){Q}$</td>
<td>$\Gamma \vdash V : A &lt;^{-} (y : B){Q}$</td>
</tr>
</tbody>
</table>

Figure 4.5: Positive and negative subtyping
Figure 4.6: Naive Subtyping with a witness.

Lemma 9 (Semantic equivalence for unfolding refinement casts).

\[ \Gamma, f = (V : (x : A)\{P\} \rightarrowrightarrow A) \vdash f : A <:_{n} B \ \ \ \ \ \Gamma \vdash B : tp \ \ \ f \notin FV(A,B) \]

\[ \Gamma \vdash V : (x : A)\{P\} <:_{n} B \]

\[ \Gamma \vdash V : (x : A)\{P\} \vdash (y : B)\{Q\} : tp \ \ \ \ \ \ \Gamma, y = (V : (x : A)\rightarrowrightarrow B) \models Q \]

\[ \Gamma \vdash V : A <:_{n} (y : B)\{Q\} \]

Lemma 10 (Context equivalence for typing). Binding in context that does not occur in terms or in types can be weakened:

- If \( \Gamma, x : A \vdash M : B \) and \( x \notin FV(M,B) \) then \( \Gamma \vdash M : B \).
- \( \Gamma, x = N \vdash M : B \) and \( x \notin FV(M,B) \) then \( \Gamma \vdash M : B \).
4.1. Subtyping with a Witness

\[ \Gamma, x : A \vdash B : \text{tp} \text{ and } x \notin \text{FV}(B) \text{ then } \Gamma \vdash B : \text{tp}. \]

\[ \Gamma, x = N \vdash B : \text{tp} \text{ and } x \notin \text{FV}(B) \text{ then } \Gamma \vdash B : \text{tp}. \]

Proof. By mutual induction on the structure of typing derivation. 

There is an overlap in our subtyping rules for refinement types and there are two possible typing derivations of the following subtyping judgement

\[ \Gamma \vdash V : (x : A)\{P\} <: (y : B)\{Q\}. \]

By inspecting those typing derivations we show that our subtyping judgement is coherent

- by firstly applying the 3rd rule for subtyping:

\[
\begin{align*}
\Gamma, f &= (V : (x : A)\{P\} \Rightarrow A) \vdash f : A <: B \\
\Gamma, f &= (V : (x : A)\{P\} \Rightarrow A) \vdash (y : B)\{Q\} : \text{tp} \\
\Gamma, f &= (V : (x : A)\{P\} \Rightarrow A), y = (f : A \Rightarrow B) \vdash Q \\
\Gamma, f &= (V : (x : A)\{P\} \Rightarrow A) \vdash f : A <: (y : B)\{Q\} \\
\Gamma &\vdash V : (x : A)\{P\} <: \text{tp} \\
\Gamma &\vdash (y : B)\{Q\} : \text{tp} \quad f \notin \text{FV}((y : B)\{Q\}) \\
\Gamma &\vdash V : (x : A)\{P\} <: (y : B)\{Q\}
\end{align*}
\]

- by secondly applying the 4th rule for positive subtyping:

\[
\begin{align*}
\Gamma &\vdash V : (x : A)\{P\} \\
\Gamma, f &= (V : (x : A)\{P\} \Rightarrow A) \vdash f : A <: B \quad f \notin \text{FV}(B) \\
\Gamma &\vdash B : \text{tp} \\
\Gamma &\vdash V : (x : A)\{P\} <: B \\
\Gamma &\vdash (y : B)\{Q\} : \text{tp} \\
\Gamma, y &= (V : (x : A)\{P\} \Rightarrow B) \vdash Q \\
\Gamma &\vdash V : (x : A)\{P\} <: (y : B)\{Q\}
\end{align*}
\]

In the first derivation we get an additional premise:

\[ \Gamma, f = (V : (x : A)\{P\} \Rightarrow A) \vdash (y : B)\{Q\} : \text{tp}, \]

which via \( f \notin \text{FV}((y : B)\{Q\}) \) and Lemma 10 is equivalent to

\[ \Gamma \vdash (y : B)\{Q\} : \text{tp}. \]
In the second derivation we get an additional premise:

\[ \Gamma \vdash B : \text{tp}, \]

which is derivable by inversion from

\[ \Gamma \vdash (y : B)\{Q\} : \text{tp}. \]

Finally we show that the semantic judgement from the second derivation

\[ \Gamma, y = (V : A \Rightarrow B) \models Q \]

implies the semantic judgement from the first derivation

\[ \Gamma, f = (V : (x : A)\{P\} \Rightarrow A), y = (V : A \Rightarrow B) \models Q \]

via \( f \notin \text{FV}(V, (x : A)\{P\}, (y : B)\{Q\}) \) and Lemma 9.

Similarly for \(<:^+, <:-, <:_n\).

**Proposition 11** (Subtyping with a witness is reflexive and transitive). For the subtyping with a witness relation \(<:\)

\[- \Gamma \vdash V : A <: A \text{ for all } \Gamma \vdash V : A.\]

\[- \text{If } \Gamma \vdash V : A <: B, \text{ and } \Gamma, x = (V : A \Rightarrow B) \vdash x : B <: C \text{ then } \Gamma \vdash V : A <: C. \]

Similarly for \(<:^+, <:-, \text{ and } <:_n.\)

**Proof.**  – Reflexivity is proven by induction on \( \Gamma \vdash V : A <: A.\)

– Transitivity is proven by induction on \( \Gamma, x = (V : A \Rightarrow B) \vdash x : B <: C.\)

Detailed proofs can be found in Appendix A.

**Proposition 12** (Tangram with a witness). We can define subtyping and naive subtyping in terms of positive and negative subtyping:

\[- \Gamma \vdash V : A <: B \iff \Gamma \vdash V : A <:^+ B \text{ and } \Gamma \vdash V : A <:- B. \]

\[- \Gamma \vdash V : A <:_n B \iff \Gamma \vdash V : A <:^+ B \text{ and } \Gamma \vdash V : B <:- A. \]

**Proof.** By induction on the derivation of subtyping.
4.2 Ordinary Subtyping

We declare that ordinary subtyping holds for all possible witnesses

**Proposition 13** (Trade of all jacks). If for all $\Gamma \vdash V : A$, $\Gamma \vdash V : A <: B$ then $\Gamma \vdash A <: B$. Similarly for $<:^+, <:-, and <:_n$.

*Proof.* Straightforward. □

**Proposition 14** (Ordinary subtyping is reflexive and transitive). For the subtyping relation $<:$$

- $\Gamma \vdash A <: A$ for all $\Gamma \vdash A : tp$, and

- if $\Gamma \vdash A <: B$ and $\Gamma \vdash B <: C$ then $\Gamma \vdash A <: C$ for all $\Gamma \vdash A : tp$, $\Gamma \vdash B : tp$, and $\Gamma \vdash C : tp$.

Similarly for $<:^+, <:-, and <:_n$.

*Proof.* Follows from Proposition 13 and Proposition 11. □

**Proposition 15** (Ordinary Subtyping and compatibility). If $\Gamma A <: B$ then $A \sim B$. Similarly for $<:^+, <:-, and <:_n$.

*Proof.* By induction on the subtyping judgement. □

**Proposition 16** (Tangram). We can define subtyping and naive subtyping in terms of positive and negative subtyping:


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

*Proof.* Follows from Proposition 12 and Proposition 13. □
### 4.3 Blame Safety

<table>
<thead>
<tr>
<th>Safety for Terms</th>
<th>$\Gamma \vdash M \text{ safe } p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash \bar{M} \text{ safe } p$</td>
<td>$\Gamma \vdash M \text{ safe } p$</td>
</tr>
<tr>
<td>$\Gamma \vdash c \text{ safe } p$</td>
<td>$\Gamma \vdash \text{op}(\bar{M}) \text{ safe } p$</td>
</tr>
<tr>
<td>$\Gamma \vdash L \text{ safe } p$</td>
<td>$\Gamma \vdash V \text{ safe } p$</td>
</tr>
<tr>
<td>$\Gamma \vdash \bar{L} \bar{V} \text{ safe } p$</td>
<td>$\Gamma \vdash \text{let } x = M \text{ in } N \text{ safe } p$</td>
</tr>
</tbody>
</table>

| $\Gamma \vdash \text{if } N \text{ then } M \text{ else } L \text{ safe } p$ |

<table>
<thead>
<tr>
<th>Safety for types</th>
<th>$\Gamma \vdash A \text{ safe } p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash x \text{ safe } p$</td>
<td>$\Gamma \vdash (x : A) { P } \text{ safe } p$</td>
</tr>
</tbody>
</table>

| $\Gamma \vdash A \text{ safe } p$ | $\Gamma, x : A \vdash P \text{ safe } p$ |
| $\Gamma \vdash \iota \text{ safe } p$ | $\Gamma \vdash (x : A) \{ P \} \text{ safe } p$ |

| $\Gamma \vdash (x : A \rightarrow B) \text{ safe } p$ |

Figure 4.7: Safety for Terms and Types
4.4 Blame Theorem

**Proposition 17** (Preservation of safe terms). For any well-typed term $M$ and blame label $p$, if $\Gamma \vdash M \text{ safe } p$ and $M \rightarrow N$, then $\Gamma \vdash N \text{ safe } p$.

*Proof.* By cases on the reduction rules. □

**Proposition 18** (Progress of safe terms). For any well-typed term $M$ and blame label $p$, if $\cdot \vdash M \text{ safe } p$, then $M \not\rightarrow \text{ blame } p$.

*Proof.* The only relevant reduction rule is rule 11, as it reduces to $\text{ blame } p$; the left-hand side is not safe for $p$. □
Chapter 5

Conclusion

We present a dependently-typed blame calculus, $\lambda dB$, with refinements over arbitrary types and fine-grained blame safety analysis.

$\lambda dB$ uses fixed order of evaluation and value-dependency to facilitate reasoning about blame. Furthermore, the typing for the compile-time language is decidable, which together with fixed order of evaluation facilitates implementation.

Refinements over arbitrary allow to express and reason about specifications.

5.1 Future Work

We plan to investigate how our presentation of subtyping can be used to optimise redundant casts, and how $\lambda dB$ relates to other value-dependent systems (Swamy et al., 2013) and to the polymorphic blame calculus (Ahmed et al., 2011; Siek and Wadler, 2016).
Appendix A

Proofs

Proposition 1 (Every typeable term has a well-formed type). If $\Gamma \vdash M : A$ then $\Gamma \vdash A : \text{tp}$.

Proof. By induction on the derivation of $\Gamma \vdash M : A$ we get the following cases:

- $\Gamma \vdash c : \text{type}(c)$, by the assumption that types for constants are well founded.
- $\Gamma \vdash op(\overline{M}) : B$, by the assumption that types for operators are well founded.
- $\Gamma, x : A \vdash x : A$, by assumption.
- $\Gamma, x = M \vdash x : A$, by applying the inductive hypothesis.
- $\Gamma, y : B \vdash x : A$, by induction.
- $\Gamma, y = M \vdash x : A$, by induction.
- $\Gamma \vdash (\lambda x : A. N) : (x : A) \rightarrow B$, given

$$
\begin{array}{c}
\frac{
\Gamma \text{ ctx} \quad \Gamma \vdash A : \text{tp} \quad \Gamma, x : A \vdash N : B
}{
\Gamma \vdash (\lambda x : A. N) : (x : A) \rightarrow B
}
\end{array}
$$

by applying the inductive hypothesis $\Gamma, x : A \vdash N : B$ we get $\Gamma, x : A \vdash B : \text{tp}$, which by applying the rule for well-formed function types shows that $\Gamma \vdash (x : A) \rightarrow B : \text{tp}$.

- $\Gamma \vdash LV : B[x := V]$, from inductive hypothesis we get

$$
\begin{array}{c}
\frac{
\Gamma \text{ ctx} \quad \Gamma \vdash A : \text{tp} \quad \Gamma, x : A \vdash B : \text{tp}
}{
\Gamma \vdash (x : A) \rightarrow B : \text{tp}
}
\end{array}
$$

and show that $\Gamma \vdash B[x := V] : \text{tp}$ via the substitution lemma (Lemma 4).
Appendix A. Proofs

- $\Gamma \vdash (\text{let } x = M \text{ in } N) : B$, by assumption.
- $\Gamma \vdash (\text{if } N \text{ then } M \text{ else } L) : A$, by induction.
- $\Gamma \vdash (V : A \Rightarrow B) : B$, by assumption.
- $\Gamma \vdash \text{blame } p : A$, by assumption.
- $\Gamma \vdash (V : A \text{ occurs in } P): (x: A)\{P\}$, by assumption.
- $\Gamma \vdash (Q : A \Rightarrow B) : (x : A)\{P\}$, by assumption.

Lemma 4 (Substitution). Substitution preserves typing for terms and types:

- If $\Gamma \vdash V : A$ and $\Gamma, x : A \vdash M : B$, then $\Gamma \vdash M[x := V] : B$.
- If $\Gamma \vdash V : A$ and $\Gamma, x : A \vdash B : \text{tp}$, then $\Gamma \vdash B[x := V] : \text{tp}$.

Proof. Proof by mutual induction on the structure of derivation of $\Gamma, x : A \vdash M : B$ and $\Gamma, x : A \vdash B : \text{tp}$.

Lemma 5 (Canonical forms). Let $V$ be a value that is well typed in the empty context then:

- If $\vdash V : \tau$, then $V = c$ with $\text{type}(c) = \tau$.
- If $\vdash V : (x : A)\{P\}$, then $V = W(x : A)\{P\}$ with $\vdash W : A$ and $P[x = W] \not\rightarrow^* \text{false}$.
- If $\vdash V : (x : A) \rightarrow B$ then either
  + $V = \lambda x : A. N$ with $x : A \vdash N : B$, or
  + $V = W : (y : C) \rightarrow D \Rightarrow (x : A) \rightarrow B$ with $\vdash W : (y : C) \rightarrow D$.

Proof.

Lemma 19. If $\Gamma \vdash \mathcal{E}[M] : A$ then there is $\Gamma \vdash B$ such that $\Gamma \vdash M : B$ for all $\Gamma \vdash N : B$, $\Gamma \vdash \mathcal{E}[N] : A$.

Proof. By induction on the structure of $\mathcal{E}$. 

\[\square\]
Proposition 6 (Preservation). If $\Gamma \vdash M : A$ and $M \rightarrow N$ then $\Gamma \vdash N : A$.

Proof. By cases on the reduction sequence. In case we first apply Lemma 19 which handles the inductive case:

- $\mathcal{E}[op(\vec{V})] \rightarrow \mathcal{E}[[op](\vec{V})]$, which is trivially true.
- $\mathcal{E}[(\lambda x:A.N)\,V] \rightarrow \mathcal{E}[N[x:=V]]$, which holds by substitution lemma.
- $\mathcal{E}[\text{let } x=V \text{ in } N] \rightarrow \mathcal{E}[N[x:=V]]$, which holds by substitution lemma.
- $\mathcal{E}[(\text{if true then } M \text{ else } L)] \rightarrow \mathcal{E}[M]$, type derivation does not change.
- $\mathcal{E}[(\text{if false then } M \text{ else } L)] \rightarrow \mathcal{E}[L]$, type derivation does not change.
- $\mathcal{E}[V:A \xrightarrow{p} A] \rightarrow \mathcal{E}[V]$, the right hand side derivation is a sub-derivation of the left hand side.

- $\mathcal{E}[V:A \xrightarrow{p} (y:B)\{Q\}] \rightarrow \mathcal{E}[\text{let } y=V:A \xrightarrow{p} B \text{ in } Q\xrightarrow{p} V_{(y:B)\{Q\}}]$, given

\[
\begin{array}{c}
\Gamma \vdash V:A \\
\Gamma \vdash (y:B)\{Q\} : \text{tp}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash (V:A \xrightarrow{p} (y:B)\{Q\}) : (y:B)\{Q\}
\end{array}
\]

to build the derivation on the right-hand side

\[
\begin{array}{c}
\text{ctx} \\
\Gamma \vdash V:A \\
\Gamma \vdash (y:B)\{Q\} : \text{tp} \\
\Gamma, y=V:A \xrightarrow{p} B \xrightarrow{p} Q\xrightarrow{p} V_{(y:B)\{Q\}} : (y:B)\{Q\}
\end{array}
\]

$\Gamma \text{ ctx}$, $\Gamma \vdash V:A$, and $\Gamma \vdash (y:B)\{Q\} : \text{tp}$ hold by assumption, we show that the remaining premise holds by applying the typing rule for casts-in-progress:

\[
\begin{array}{c}
(\Gamma, y=V:A \xrightarrow{p} B) \text{ ctx} \\
\Gamma, y=V:A \xrightarrow{p} B \vdash Q: \text{bool} \\
\Gamma, y=V:A \xrightarrow{p} B \vdash y:B
\end{array}
\]

\[
\begin{array}{c}
(\Gamma, y=V:A \xrightarrow{p} B) \text{ ctx} \\
\Gamma, y=V:A \xrightarrow{p} B \vdash Q\xrightarrow{p} y_{(y:B)\{Q\}} : (y:B)\{Q\}
\end{array}
\]

we show that the premises hold:

+ $(\Gamma, y=V:A \xrightarrow{p} B) \text{ ctx}$ by applying the rule for well-formed contexts
* $\Gamma \text{ ctx}$ holds by assumption,
* while by applying the rule for typing casts:

$$
\Gamma \text{ ctx} \quad \Gamma \vdash V : A \quad A \sim B \quad \Gamma \vdash B : \text{tp} \\
\hline
\Gamma \vdash (V : A \rightarrow B) : B
$$

which premises:

- $\Gamma \text{ ctx}$ holds by assumption
- $\Gamma \vdash V : A$ holds by assumption
- $A \sim B$ is a sub-derivation of $A \sim (y : B)\{Q\}$ from the left-hand side, and
- $\Gamma \vdash B : \text{tp}$ is a sub-derivation of $\Gamma \vdash (y : B)\{Q\} : \text{tp}$ from the left-hand side.

+ we show that $\Gamma, y = V : A \rightarrow B \vdash y : B$ holds by applying the typing rule for variables

$$
\Gamma \text{ ctx} \quad \Gamma \vdash V : A \rightarrow B \vdash B : B \\
\hline
\Gamma, y = V : A \rightarrow B \vdash y : B \vdash y : B
$$

both $\Gamma \text{ ctx}$ and $\Gamma \vdash V : A \rightarrow B : B$ are already proven.

+ we show that $\Gamma, y = V : A \rightarrow B \vdash Q : \text{bool}$ by applying Lemma 20 with $\Gamma \vdash (V : A \rightarrow B) : B$ to get $\Gamma, y = V : A \rightarrow B \vdash Q : \text{bool}$, which is a sub-derivation of $\Gamma \vdash (y : B)\{Q\}$.

$$
- \mathcal{E}[(V_{(x:A)}(P)) : (x : A)\{P\} \rightarrow B] \rightarrow \mathcal{E}[V : A \rightarrow B], \text{ given}
$$

$$
\Gamma \vdash V : A \quad \Gamma \vdash (x : A)\{P\} : \text{tp} \quad P[x := V] \not\rightarrow^* \text{false} \\
\hline
\Gamma \vdash (V_{(x:A)}(P)) : (x : A)\{P\} \sim B \quad \Gamma \vdash B : \text{tp}
$$

$$
\hline
\Gamma \vdash (V_{(x:A)}(P)) : (x : A)\{P\} \rightarrow B
$$

we construct the derivation for the right-hand side

$$
\Gamma \text{ ctx} \quad \Gamma \vdash V : A \quad A \sim B \quad \Gamma \vdash B : \text{tp} \\
\hline
V : A \rightarrow B
$$

by assumption, while $A \sim B$ is a sub-derivation of $(x : A)\{P\} \sim B$. 
we split constructing the derivation into two steps:

+ we start by constructing the derivation of the innermost let-binding:

\[
\begin{array}{c}
\Gamma \text{ ctx} \quad \Gamma \vdash V : (x : A) \rightarrow B \\
(x : A) \rightarrow (y : C) \rightarrow D \quad \Gamma \vdash (y : C) \rightarrow D : \text{tp}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash (V : (x : A) \rightarrow B = (y : C) \rightarrow D) : (y : C) \rightarrow D \\
\Gamma \text{ ctx} \quad \Gamma \vdash W : C \\
\Gamma \vdash (V : (x : A) \rightarrow B = (y : C) \rightarrow D) W : D[y := W]
\end{array}
\]

we split constructing the derivation into two steps:

+ we start by constructing the derivation of the innermost let-binding:

\[
\begin{array}{c}
\Gamma, e = W : C \rightarrow B A \text{ ctx} \\
\Gamma, e = W : C \rightarrow B A \vdash (V e) : B[x := e]
\end{array}
\]

\[
\begin{array}{c}
\Gamma, e = W : C \rightarrow B A, f = (V e) \vdash (f : B[x := e] \rightarrow D[y := W]) : D[y := W] \\
\Gamma, e = W : C \rightarrow B A \vdash D[y := W] : \text{tp}
\end{array}
\]

before we prove the premises, we show that an auxiliary assumption

\[
\Gamma \vdash W : C \rightarrow B A : A \text{ holds by applying the typing rule for casts}
\]

by showing for each premise that:

* \(\Gamma \text{ ctx} \) holds by assumption,
* \(\Gamma \vdash W : C \) holds by assumption,
* \(C \rightarrow A\) is a subderivation of \((x : A) \rightarrow B \rightarrow (y : C) \rightarrow D\), and
* \(\Gamma \vdash A : \text{tp}\) is a subderivation of \(\Gamma \vdash (y : C) \rightarrow D : \text{tp}\).

then we prove the premises of the innermost let-binding:

* \((\Gamma, e = W : C \rightarrow B A \text{ ctx})\) holds as \(\Gamma \text{ ctx}\) and \(\Gamma \vdash W : C \rightarrow B A : A\).
* \(\Gamma, e = W : C \rightarrow B A \vdash (V e) : B[x := e]\), by applying the typing rule for application:

\[
\begin{array}{c}
\Gamma, e = W : C \rightarrow B A \text{ ctx} \\
\Gamma, e = W : C \rightarrow B A \vdash V : (x : A) \rightarrow B \\
\Gamma, e = W : C \rightarrow B A \vdash e : A \\
\Gamma, e = W : C \rightarrow B A \vdash V e : B[x := e]
\end{array}
\]
Appendix A. Proofs

which premises

- \((\Gamma, e = W : C \xrightarrow{p} A)\) \(\text{ctx}\) was already proven,
- \(\Gamma, e = W : C \xrightarrow{p} A \vdash V : (x : A) \rightarrow B\) holds by Lemma 10 with 
  \(e \notin FV(V)\),
- \(\Gamma, e = W : C \xrightarrow{p} A \vdash e : A\)

\* \(\Gamma, e = W : C \xrightarrow{p} A, f = (V e) \vdash (f : B[x := e] \xrightarrow{p} D[y := W]) : D[y := W]\), by applying the typing typing rule for casts

\[(\Gamma, e = W : C \xrightarrow{p} A, f = (V e)) \text{ctx} \]
\[
\Gamma, e = W : C \xrightarrow{p} A, f = (V e) \vdash f : B[x := e]
\]
\[
B[x := e] \ D[y := W]
\]

\[
\Gamma, e = W : C \xrightarrow{p} A, f = (V e) \vdash (f : B[x := e] \xrightarrow{p} D[y := W]) : D[y := W]
\]

the we show that the premises hold

- \(\Gamma, e = W : C \xrightarrow{p} A, f = (V e)) \text{ctx}, holds by construction with 
  \((\Gamma, e = W : C \xrightarrow{p} A) \text{ ctx}\) and \(\Gamma, e = W : C \xrightarrow{p} A \vdash (V e) : B[x := e]\) which are already proven.
- \(\Gamma, e = W : C \xrightarrow{p} A, f = (V e) \vdash f : B[x := e]\) holds by applying typing rule for variables with \(\Gamma, e = W : C \xrightarrow{p} A, f = (V e) \text{ ctx}\) and \(\Gamma, e = W : C \xrightarrow{p} A \vdash V e : B[x := e]\), which are already proven.
- \(B[x := e] \ D[y := W]\) holds by substitution lemma.

\* \(\Gamma, e = W : C \xrightarrow{p} A \vdash D[y := W] : \text{tp}\) holds by substitution lemma

with \(\Gamma, y : C \vdash D : \text{tp}\), then by Lemma 10 with \(e \notin FV((y : C) \rightarrow D)\).

+ then we can construct the derivation for the outermost let-binding:

\[
\Gamma, e = W : C \xrightarrow{p} A \vdash (\text{let } f = (V e) \text{ in } (f : B[x := e] \xrightarrow{p} D[y := W])) : D[y := W]
\]

\[
\Gamma \vdash W : C \xrightarrow{p} A : A \qquad \Gamma \text{ctx} \qquad \Gamma \vdash D[y := W] : \text{tp}
\]

\[
\Gamma \vdash \text{let } e = W : C \xrightarrow{p} A \text{ in } (\text{let } f = (V e) \text{ in } (f : B[x := e] \xrightarrow{p} D[y := W]))
\]

and show that its premises

+ \(\Gamma, e = W : C \xrightarrow{p} A \vdash (\text{let } f = (V e) \text{ in } (f : B[x := e] \xrightarrow{p} D[y := W])) : D[y := W]\) is already proven,
+ \(\Gamma \vdash W : C \xrightarrow{p} A : A\) is already proven,
+ \(\Gamma \text{ctx}, is\ already\ proven,
+$\Gamma \vdash D[y := W] : \text{tp}$, is already proven.

$-\mathcal{E}[\text{true} \triangleright^p V_{(x:A)}\{P\}] \rightarrow \mathcal{E}[V_{(x:A)}\{P\}]$, given

\[
\begin{align*}
\cdot \text{ctx} & \quad \cdot \vdash \text{true} : \text{bool} \\
\cdot \vdash V : A & \quad \cdot \vdash (x : A)\{P\} : \text{tp} & P[x := V] & \rightarrow^* \text{true} \\
\end{align*}
\]

$\Gamma \vdash (\text{true} \triangleright^p V_{(x:A)}\{P\}) : (x : A)\{P\}$

we construct the right-hand side derivation

\[
\begin{align*}
\cdot \text{ctx} & \quad \cdot \vdash V : A & \quad \cdot \vdash (x : A)\{P\} : \text{tp} & P[x := V] & \not\rightarrow^* \text{false} \\
\cdot \vdash (V_{(x:A)}\{P\}) : (x : A)\{P\} \\
\end{align*}
\]

which premises

+$\cdot\text{ctx}$ is trivial,

+$\cdot\vdash V : A$ holds by assumption,

+$\cdot\vdash (x : A)\{P\} : \text{tp}$ holds by assumption,

+$P[x := V] \not\rightarrow^* \text{false}$, holds as $P[x := V] \rightarrow^* \text{true}$.

$-\mathcal{E}[\text{false} \triangleright^p V_{(x:A)}\{P\}] \rightarrow \mathcal{E}[\text{blame } p]$ given

\[
\begin{align*}
\cdot \text{ctx} & \quad \cdot \vdash \text{false} : \text{bool} \\
\cdot \vdash V : A & \quad \cdot \vdash (x : A)\{P\} : \text{tp} & P[x := V] & \rightarrow^* \text{false} \\
\end{align*}
\]

$\cdot \vdash (\text{false} \triangleright^p V_{(x:A)}\{P\}) : (x : A)\{P\}$

we construct the right-hand side derivation

\[
\begin{align*}
\cdot \text{ctx} & \quad \cdot \vdash A : \text{tp} \\
\cdot \vdash \text{blame } p : A \\
\end{align*}
\]

which premises

+$\cdot\text{ctx}$ is trivial

+$\cdot\vdash A : \text{tp}$ is a sub-derivation of of $\cdot\vdash (x : A)\{P\} : \text{tp}$ from the left-hand side.

$\square$

**Proposition 7** (Progress). If $\cdot \vdash M : A$ then either:
− $M$ is a value.

− $M \rightarrow N$ for some term $N$.

− $M$ is blame $p$ for some blame label $p$.

**Proof.** By induction on the typing derivation $\cdot \vdash M : A$ we get the following cases:

− $\cdot \vdash c : \text{type}(c)$, where $c$ is a value.

− $\cdot \vdash op(M) : B$, where either:
  
  + all operands are values $\tilde{V}$, and the whole thing steps to $[[op]](\tilde{V})$,
  
  + one of operands steps, then the whole thing steps.
  
  + one of operands steps to blame $p$, then the whole thing steps to blame $p$.

− $\cdot \vdash \text{(let } x = M \text{ in } N) : B$, then either:
  
  + $M$ is a value $V$ then the whole thing steps to $N[x := V]$,
  
  + $M$ steps then the whole thing steps,
  
  + $M$ steps to blame $p$ then the whole thing steps to blame $p$.

− $\cdot \vdash \lambda x : A. N : (x : A) \rightarrow B$, and $\lambda x : A. N$ is a value.

− $\cdot \vdash LV : B[x := V]$, then either:
  
  + $L$ is a value, then via Lemma 5 it is either:
    
    * $\lambda x : A. N$, so the whole thing steps to $N[x := V]$, or
    
    * $W : (x : A) \rightarrow B \xrightarrow{p} (y : C) \rightarrow D$, so the whole thing steps to
      
      \[
      \text{let } e = V : C \xrightarrow{p} A \text{ in } \\
      \text{let } f = (W e) \text{ in } \\
      f : B[x := e] \xrightarrow{p} D[y := V]
      \]
      
      where $e \notin \text{FV}(W, (x : A) \rightarrow B, (y : C) \rightarrow D)$ and $f \notin \text{FV}((x : A) \rightarrow B, (y : C) \rightarrow D)$.

  + $L$ steps then the whole thing steps,

  + $L$ steps to blame $p$ then the whole thing steps to blame $p$.

− $\cdot \vdash \text{(if } N \text{ then } M \text{ else } L) : A$, then either:
\[ N \text{ is a value } V, \text{ then via Lemma 5 } V \text{ is either:} \]

\[ \text{\ast } \text{true, then the whole thing steps to } M. \]
\[ \text{\ast } \text{false, then the whole thing steps to } L. \]

+ \( N \) steps then the whole thing steps,

+ \( N \) steps to \text{blame } p \text{ then the whole thing steps to } \text{blame } p.

\[ \vdash (V : A \xrightarrow{p} B) : B \text{ then the whole thing steps to either:} \]

+ If \( A = (x : A')\{P\} \) then via Lemma 5, \( V = V'_{(x:A')\{P\}}, \) then the whole thing steps to

\[ V' : A' \xrightarrow{p} B \]

+ If \( B = (y : B')\{Q\} \) then the whole thing steps to

\[ \text{let } y = V : A \xRightarrow{p} B \text{ in } (Q \map_{p} V_{(y:B)\{Q\}}) \]

\[ \vdash \text{blame } p : A \text{ is } \text{blame } p. \]

\[ \vdash (V_{(x:A)\{P\}}) : (x : A) \{P\} \text{ is a value.} \]

\[ \vdash (P \map_{p} V_{(x:A)\{P\}}) : (x : A) \{P\}, \text{ then either:} \]

+ \( P \) is a value \( V \), then via Lemma 5 \( V \) is either:

\[ \text{\ast } \text{true, then the whole thing steps to } V_{(x:A)\{P\}}. \]
\[ \text{\ast } \text{false, then the whole thing steps to } \text{blame } p. \]

+ \( P \) steps then the whole thing steps,

+ \( P \) steps to \text{blame } q \text{ then the whole thing steps to } \text{blame } q.

\[ \vdash (Q \map_{p} V_{(x:A)\{P\}}) : (x : A) \{P\}, \text{ then either:} \]

+ \( Q \) is a value \( V \), then via Lemma 5 \( V \) is either:

\[ \text{\ast } \text{true, then the whole thing steps to } V_{(x:A)\{P\}}. \]
\[ \text{\ast } \text{false, then the whole thing steps to } \text{blame } p. \]

+ \( Q \) steps then the whole thing steps,

+ \( Q \) steps to \text{blame } q \text{ then the whole thing steps to } \text{blame } q.

\[ \square \]
Lemma 9 (Semantic equivalence for unfolding refinement casts).

\[ \Gamma, f = (V : (x : A)\{P\} \Rightarrow A), y = (f : A \Rightarrow B) \models Q \]

if

\[ \Gamma, y = (V : (x : A)\{P\} \Rightarrow B) \models Q \]

for every \( f \not\in \text{FV}(A, B, Q) \).

Proof. By case analysis on the structure of \( \Gamma \).

- for empty environment \( \cdot \), \( V = W_{(x : A)\{P\}} \) via Lemma 5. Then there is a single closing substitution for both the left-hand side and right-hand side, which reduces to the same redex

\[
\text{let } y = W \Rightarrow A \text{ in } Q
\]

which via Proposition 8 should reduce to the same value, which in both cases will not be false, meaning that left-hand side will imply right-hand side and vice-versa.

- for environment extended with a binding to type, a closing substitution will choose an arbitrary value of type \( A \), but for every closing substitution from the left-hand side, there will be a closing substitution on the right-hand side that will chose the same value. Therefore for each closing substitution on the left-hand side, there will be a corresponding closing substitution on the right-hand side that will reduce to the same redex, as in the case above.

- for environment extended with a binding to a term, a closing substitution from the left hand side will correspond to the closing substitution to the right-hand side, and they will have a common redex as in the case above.

\[ \square \]

Lemma 20 (Generalisation for typing). We can always generalise a binding to a term to a binding to a type

- if \( \Gamma, x = N \vdash M : B \) then \( \Gamma, x : A \vdash M : B \), and

- if \( \Gamma, x = N \vdash B : \text{tp} \) then \( \Gamma, x : A \vdash B : \text{tp} \).

for all \( \Gamma, x, N, M, A, B \) such that

\[ \Gamma \vdash N : A. \]
Proof. By mutual induction on the typing derivation. \qed

**Lemma 21** (Context weakening for semantic judgement). *Binding in context that does not occur in predicate can be weakened:*

- If $\Gamma \models P$ then $\Gamma, x : A \models P$.
- If $\Gamma \models P$ then $\Gamma, x = N \models P$.

for every $\Gamma$, $x$, $A$, $N$, and $P$ such that

$x \notin (\Gamma \models P)$,

$\Gamma \vdash A : tP$, and

$\Gamma \vdash N : A$.

Proof. If $x \notin (\Gamma \models P)$, then $x \notin P$, so any binding to $x$ assigned by a closing substitution will not affect the outcome of the judgement. \qed

**Lemma 22** (Semantic equivalence for wrapped functions). *We can unfold applications of dependent functions in the environment:*

$$\Gamma, z = ((V : (x : A) \to B \xrightarrow{\bullet} (y : C) \to D) W) \models P$$

if and only if

$$\Gamma, e = W, x = (y : C \xrightarrow{\bullet} A), f = (V x), z = (f : B \xrightarrow{\bullet} D) \models P$$

assuming

$$\Gamma \vdash V : (x : A) \to B$$

$$\Gamma \vdash (y : C) \to D : tP$$

$$\Gamma, g : C \vdash g : C \triangleleft A$$

$$\Gamma, g : C, e = (g : C \xrightarrow{\bullet} A), f = (V x) \vdash f : B[x := e] \triangleleft D[y := g]$$

$$e, g \notin \text{FV}(V, (x : A) \to B, (y : C) \to D)$$

$$f \notin \text{FV}((x : A) \to B, (y : C) \to D)$$

Proof. For every application of closing substitution satisfying left-hand-side, there is a corresponding application of closing substitution satisfying right-hand-side, up to alpha-equivalence on $e$ and $f$. \qed
**Lemma 23** (Subtyping is defined only over well-formed types). If \( \Gamma \vdash V : A <: B \) then \( \Gamma \vdash A : tp \) and \( \Gamma \vdash B : tp \) for all \( \Gamma, V, A, \) and \( B \). Similarly for \( <:^- \) and \( <:_n \).

**Proof.** By induction on the derivation of \( \Gamma \vdash V : A <: B \) we get the following cases:

- \( \Gamma \vdash V : \iota <:^+ \iota \), which is trivial.
- \( \Gamma \vdash V : (x : A) \rightarrow B <:^+ (y : C) \rightarrow D \), which holds by application of Lemma 1 and assumption.
- \( \Gamma \vdash V : (x : A)\{P\} <:^+ B \), which holds by application of Lemma 1 and induction.
- \( \Gamma \vdash V : A <:^+ (x : B)\{P\} \), which holds by induction and assumption.

We can repeat the proof for \( <:^- \) and \( <:_n \) in the same fashion. \( \square \)

**Proposition 11** (Subtyping with a witness is reflexive and transitive). For the subtyping with a witness relation \(<:\)

- \( \Gamma \vdash V : A <: A \) for all \( \Gamma \vdash V : A \).
- If \( \Gamma \vdash V : A <: B \), and \( \Gamma, x = (V : A \Rightarrow B) \vdash x : B <: C \) then \( \Gamma \vdash V : A <: C \).

Similarly for \( <:^+, \<^- \), and \( <:_n \).

**Proof.** Reflexivity is proven by straightforward induction on \( \Gamma \vdash V : A <: A \).

Transitivity is proven by induction on \( \Gamma, x = (V : A \Rightarrow B) \vdash x : B <: C \) and there are the following cases

- \( \Gamma \vdash x : \iota <: \iota \) which is trivial,
- \( \Gamma \vdash V : (x : A) \rightarrow B <: (y : C) \rightarrow D \) which holds by showing equivalence for expansion of wrapped function via Lemma ??.
- \( \Gamma \vdash x : (x : A)\{P\} <: B \) which holds by the inductive hypothesis
- \( \Gamma \vdash x : A <: (y : B)\{Q\} \) which holds via Lemma 9.

\( \square \)


Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Sym-


