Design and Evaluation of Contracts for Gradual Typing

Jack Williams

Doctor of Philosophy

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2019

Abstract

Gradual typing aims to improve the correctness of dynamically typed programs by incrementally adding type information. Sound gradual typing performs static type checking and inserts run-time checks when a type cannot be guaranteed statically. This form of gradual typing offers many features, but also requires that the programmer uses a language with a specialised gradual type system. A lightweight form of gradual typing uses contracts to enforce types at run-time, assigning blame when a type assertion fails. Contracts can be implemented as a library, without requiring a specialised gradual type system. Contracts provide a lower barrier of entry into sound gradual typing.

This thesis investigates the design and evaluation of contracts for gradual typing, focusing on bridging the gap between JavaScript (dynamic) and TypeScript (static). There are two key outcomes regarding theory and practice. Contracts for higher-order intersection and union types can be designed in a uniform way, using blame to derive the semantics of contracts satisfaction. Contracts and gradual typing can be evaluated using the DefinitelyTyped repository, where JavaScript libraries are annotated with TypeScript definition files.

Contract composition is the fundamental method for building complex type assertions. Intersection and union types are well suited for describing patterns common to dynamically typed programs. Our first contribution is to present a calculus of contracts for intersection and union types with blame assignment, giving a uniform treatment to both operators.

A correct model of contracts must include a definition of contract satisfaction. Our second contribution is to show that contract satisfaction can be defined using blame: satisfying programs are those that do not elicit blame when monitored. We define a series of properties mandating how contract satisfaction should compose, ensuring that a contract for a type behaves as one would expect for that type.

Building on our technical developments, our third contribution is a practical evaluation of gradual typing using the DefinitelyTyped repository. We show that contracts can be used to enforce conformance to a definition file, detecting errors in the specification. Our evaluation also reveals that technical concerns associated with implementing contracts using JavaScript proxies are a problem in practice.

Lay Summary

The purpose of a software system determines which tools are best suited for building that system. *Statically typed* programming languages facilitate correctness and require that the design of the system is explicit about the shape, or *type*, of the data that is used. *Dynamically typed* programming languages facilitate flexibility and allow the design of the system to make no assumptions about the type of the data.

The challenge faced by many companies is that the purpose of a software system can change over time. A prototype written in a dynamically typed language may become an important system that would be better suited to a statically typed language. Companies are burdened with the laborious task of rewriting their systems in languages better suited to long-term maintenance.

Gradually typed languages have been developed to ease the migration from dynamic typing to static typing by allowing the same program to combine dynamically typed and statically typed fragments. With gradual typing, a prototype can be built using predominantly dynamic typing, and then incrementally upgraded to use predominantly static typing. A crucial aspect of gradual typing is that the boundary between dynamically and statically typed fragments is monitored at run-time. Monitoring ensures that data flowing into a statically typed fragment has the correct type.

This thesis studies a monitoring technique used in gradual typing known as *software contracts*. A contract describes a program property that is enforced during the execution of a program; if a program breaks the contract a violation is raised that identifies the faulty code.

The first part of this thesis investigates how we can compose contracts to enforce richer properties, and how we can verify that contracts enforce the properties they describe. We focus on two mechanisms to compose contracts: intersection and union. The intersection of two contracts enforces both contracts; the union of two contracts enforces that at least one contract applies.

The second part of this thesis investigates how contracts and gradual typing can be used with JavaScript, the dominant programming language for the web. We focus on two questions. First, is contract enforcement justified by detecting many violations in real code? Second, can contract enforcement be correctly implemented using existing technology in JavaScript?

Acknowledgements

My first thanks are to Philip Wadler my supervisor. He showed me what it takes to be a great researcher, and I consider it a privilege to have been his student. I would also like to thank Garrett Morris and Sam Lindley who also contributed to my supervision over the past four years.

Thanks to Microsoft for enabling me to pursue my studies through their PhD Scholarship. Thanks to Andy Gordon for hosting me at Microsoft Research as an intern and giving me the opportunity to work on a new set of problems.

I am grateful to Peter Thiemann and James Cheney for being my examiners and making the viva a positive and rewarding experience.

I would like to thank my friends, both inside the Forum and out, for providing company and support. Finally, I would like to thank Lene and my family. I would not have been able to get this far on my own.

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. Work in this thesis has appeared in the following publications:

- The content in Chapter 3 and Chapter 4 appears in *The Root Cause of Blame: Contracts for Intersection and Union Types* (Williams et al., 2018).
- The content in Chapter 5 and Chapter 6 appears in *Mixed Messages: Measuring Conformance and Non-Interference in TypeScript* (Williams et al., 2017a).

I declare that I was the primary author for both publications.

(*Jack Williams*)

Table of Contents

Chapter 1

Introduction

Renovating a house is a hard task; renovating a house that you are living in is even harder. Software development suffers from a similar problem. Upgrading a legacy code base is a hard task; upgrading a legacy code base that you actively depend upon is even harder. This is an issue that many large companies face. A software application is written to satisfy short-term goals, but goes on to become a vital piece of infrastructure that requires long-term maintenance. Program renovation is interspersed with bug fixes and new features, and must be done with minimal downtime.

A particular manifestation of this problem is through the choice of programming language used to build an application. Dynamically typed languages place weak constraints on the specification of a program; a programmer is free to compose the program in such a way that might be unsafe at run-time. This liberal approach can make it faster to prototype an application, but if the prototype grows into a significant application then this freedom can become a burden. Weak constraints provide weak guarantees. A dynamically typed language provides few assurances about the correctness of the program.

Statically typed languages are rigorous in their construction; a static checker will reject programs that may elicit certain kinds of unsafe behaviour. A statically typed language *knows* more about a program, so a statically typed language can *say* more about a program. Documentation, tooling, and testing are all improved by type information, making statically typed languages suitable for large applications that require long-term maintenance.

Programmers would like to renovate an application from dynamically typed to statically typed when it reaches critical mass. This renovation has to be done in full: the entire program must to be converted from dynamic to static and cannot be left in

an intermediate state. The transition requires a significant upfront investment and for large applications the cost can be too much to bear. To make progress, programmers must be able to *pay as they go*.

Gradual typing is an approach to programming language design that facilitates the incremental transition from dynamically typed to statically typed. With gradual typing a programmer can add *some* type annotations to a dynamically typed program, but significantly, they are not required to fully annotate the program. At every point in the transformation the program is a valid gradually typed program that remains usable. This is the salient feature that make gradual typing suitable for incremental renovation of legacy code. Over time, the program may be transformed from fully dynamically typed to fully statically typed.

A gradually typed program consists of three key components: dynamically typed fragments, statically typed fragments, and boundaries between these fragments. The boundary is where interesting things happen, as data from one realm crosses over to the other. At the boundary is where the opposing realms of dynamic and static meet, and where their opposing views are reconciled.

Some approaches to reconciliation are strict: the boundary between dynamic and static is enforced at run-time and values must be checked before crossing. This sound approach uses contracts to implement dynamic type checking. Contracts ensure that type annotations mediating between dynamic and static realms are meaningful. A value crossing the boundary must conform to the annotated type, otherwise an exception is raised.

Some approaches to reconciliation are permissive: the boundary between dynamic and static is erased at run-time and values cross unchecked. Such an approach is easy to implement, but type annotations mediating between dynamic and static realms are reduced to superficial documentation. There is no guarantee that a value crossing the boundary conforms to the annotated type.

Theoretical and practical approaches to gradual typing are not in alignment. The theory of gradual typing favours soundness and strict enforcement of the boundary. The practice of gradual typing has yet to commit to soundness and favours erasure of the dynamic-static boundary.

We study the theory and practice of gradual typing with the aim to bring them closer in alignment. The thesis of this work is that intersection and union contracts can be used to apply sound gradual typing to the largest repository of gradually typed programs—the DefinitelyTyped repository. Type annotations are frequently incorrect

and sound gradual typing can be used to mitigate this. However, implementation issues taken for granted in theory occur in practice, reducing the effectiveness of sound gradual typing. Strengthening the connection between theory and practice requires effort from both sides. Researchers of gradual typing must consider practical concerns during design. Practitioners of gradual typing must consider the consequence of widespread conformance violation during implementation.

1.1 Background

Contracts A vital feature of sound gradual typing is the use of dynamic type checking to enforce type conformance, but gradual typing did not spawn this idea. Contracts that dynamically assert program invariants were first proposed by Meyer (1988, 1992), where contracts enforce function pre-conditions and post-conditions.

Early work was limited to first-order function contracts that could be immediately verified at the call-site of a function. This restriction was lifted by the development of higher-order function contracts by Findler and Felleisen (2002); their work was a significant contribution in the development of gradual typing. The insight of Findler and Felleisen (2002) was *wrapping*: a higher-order function contract cannot be verified immediately so it must be wrapped, where a wrapped function applies its domain and codomain contracts on demand. Their work introduced the concept of *blame*. A violated contract assigns blame to the party responsible. There are two parties to a function contract, and consequently a function contract can assign two kinds of blame: positive and negative. Positive blame indicts the subject of the contract and negative blame indicts the context of the contract.

Contract operators facilitate contract composition, allowing a programmer to describe a rich set of invariants from reusable components. Findler and Felleisen (2002) created a spark that ignited a flurry of work, and contracts have subsequently been extended to support features including parametric polymorphism (Guha et al., 2007), intersection and union (Keil and Thiemann, 2015a), temporal properties (Disney et al., 2011), and session types (Jia et al., 2016; Melgratti and Padovani, 2017).

Gradual Typing in Theory The origins of gradual typing can be traced back to the work by Henglein (1994), later extended by Siek and Taha (2006) and Tobin-Hochstadt and Felleisen (2006). Siek and Taha (2006) developed the first system that supports sound, fine-grained interaction between dynamically and statically typed code. Casts

mediate between dynamically and statically typed code and cast composition implements dynamic type checking. For example:

$$((V : \mathtt{I} \Rightarrow \mathsf{any}) : \mathsf{any} \Rightarrow \mathtt{I}) \longrightarrow V$$

Composing a cast from integer type `I` to dynamic type any, with a cast from dynamic type any to integer type `I`, allows a value from dynamically typed code to safely flow into statically typed code. Illegal cast composition will trigger a cast error. For example:

$$((V : \mathtt{I} \Rightarrow \mathsf{any}) : \mathsf{any} \Rightarrow \mathtt{B}) \longrightarrow \mathit{CastError}$$

The static realm is expecting a value of boolean type `B`, however the dynamic realm has provided a value of integer type `I`. A cast error is raised that indicates a violation at the boundary between dynamic and static.

A key development in the work by Siek and Taha (2006) is the *consistency* relation. Existing work on dynamic-static integration struggled to get the balance right when using casts to mediate between types. Too conservative and the system prevents desirable inter-operation between dynamic and static; too liberal and the type system collapses in on itself by permitting casts between nonsensical types. Siek and Taha (2006) use the consistency relation to define meaningful casts. The salient trait of consistency is that the relation is not transitive. The relation is just flexible enough to allow desirable inter-operation, but not so flexible that the type system degenerates.

Tobin-Hochstadt and Felleisen (2006) concurrently developed an alternate approach to gradual typing. Their style takes a wider perspective, focusing on the interaction between dynamically and statically typed modules. Statically typed modules use *contracts* to ensure that dynamically typed modules conform to the desired type. Contracts record provenance and assign *blame* when violated, indicating the source of the violation. The insight provided by Tobin-Hochstadt and Felleisen (2006) is that blame tracking can be used to prove strong guarantees about where contract violations occur: *"code in typed modules can't go wrong"*.

Wadler and Findler (2009) brought harmony to the work of Siek and Taha (2006) and Tobin-Hochstadt and Felleisen (2006), combining a fine-grained cast calculus with blame tracking—*blame calculus*. The soundness guarantee offered by blame calculus is that *"well-typed programs can't be blamed"*, shown by decomposing subtyping into positive and negative variants. Positive and negative subtyping then recombine to produce naive subtyping, a strictly covariant form of subtyping.

Gradual typing research has subsequently flourished. A variety of type system features have been extended to gradual typing, including objects (Chung et al., 2018; Siek and Taha, 2007), references (Siek et al., 2015c), parametric polymorphism (Ahmed et al., 2017; Igarashi et al., 2017a; Toro et al., 2019), session types (Igarashi et al., 2017b; Thiemann, Peter, 2014), intersection and union types (Castagna and Lanvin, 2017), and linear types (Fennell and Thiemann, 2013).

Gradual Typing in Practice The theory of contracts, blame, and gradual typing combine to form a significant body of work aimed at integrating dynamically and statically typed code. Practitioners in industry have begun to adopt the ideas of gradual typing, however they are yet to fully embrace sound gradual typing.

Industry demand for combining dynamic and static typing is evident. There is a wide variety of industry tools focused on integrating dynamically and statically typed code: Hack[1] by Facebook, a gradually typed language for integration with PHP; Pytype[2] by Google, a type checker for Python that exploits type hints; Flow by Facebook, a type checker for JavaScript (Chaudhuri et al., 2017); and TypeScript by Microsoft, a gradually typed superset of JavaScript with a focus on tooling and documentation (Bierman et al., 2014).

Every tool listed takes the permissive stance when reconciling the boundary between dynamically and statically typed code, coined the *erasure embedding* by Greenman and Felleisen (2018). All type annotations are erased at run-time causing the boundary between dynamically and statically typed components to evaporate. Type annotations at the boundary are purely descriptive and have no run-time effect.

TypeScript is the most popular of these tools and the integration of TypeScript code with legacy JavaScript libraries is emblematic of the erasure embedding. A JavaScript library is paired with a *definition file*: a hand written specification of the library interface using TypeScript types. Definition files are consumed by TypeScript clients and prompt auto-completion and type checking for a client's conformance to the definition. TypeScript definition files are vital to the ecosystem: the Definitely-Typed repository now contains definitions for over 4000 JavaScript libraries.

Figure 1.1 presents a basic example of a definition file, the corresponding JavaScript library, and a TypeScript client. The client expects to receive a value of type `Box` after calling the library function `makeBox`, and TypeScript will provide auto-completion for

[1] `https://hacklang.org`
[2] `https://opensource.google.com/projects/pytype`

Definition: box.d.ts

```
1 interface Box {
2   content: number;
3 }
4 export function makeBox(v: number): Box;
```

Library: box.js

```
1 module.exports.makeBox = v => ({ contents: v });
```

Client: box-client.ts

```
1 import * as basic from "./box";
2 const box: Box = basic.makeBox(42);
3 const result: number = box.content + 100;
```

Figure 1.1: Example Definition, Library, and Client

field `content` of value `box`.

TypeScript compiles to JavaScript and erases all types, including the types originating from the definition file. After erasure, any semblance of a dynamic-static boundary is lost. There is no way to determine if the library behaves in conformance with the definition file at run-time because there are no types at run-time. In the example the library implementation subtly differs from the definition file; the former uses the field name `contents` while the latter uses the field name `content`. This error is concealed to the client because it originates from the dynamic side of the boundary; consequently, client code will evaluate in an unexpected manner. The client expects `result` to evaluate to `142` but `result` will evaluate to `NaN` instead.

The choice to erase the dynamic-static boundary is motivated by two factors: performance and implementation complexity. The performance regression introduced by contracts used in sound gradual typing can be significant (Takikawa et al., 2016). Recent progress has been made to reduce performance degradation, however this is yet to trickle down to mainstream implementations (Bauman et al., 2017; Feltey et al., 2018; Muehlboeck and Tate, 2017). The second limiting factor is that implementing contracts capable of monitoring all boundary interactions is difficult. TypeScript exploits intersection and union types in definition files to describe overloaded functions and uncertainty. Findler and Felleisen (2002) introduced higher-order function

contracts before the rise of gradual typing, but it was not until much later that intersection and union contracts were added by Keil and Thiemann (2015a). The design and correctness criteria of intersection and union contracts may be solved, but their implementation still presents technical challenges.

There is significant momentum behind both the theory and practice of gradual typing, however the trajectories of the two tracks are yet to fully align. This is, in part, due to a lack of research aimed at applying sound gradual typing principles in real-world contexts.

1.2 Contributions

This work makes contributions to the theory and application of contracts for sound gradual typing. First, by presenting a new design and implementation of higher-order intersection and union contracts. Second, by conducting an empirical evaluation of sound gradual typing against the DefinitelyTyped repository. Concretely, the contributions of this work are:

- Extending the untyped lambda calculus with the first implementation of higher-order intersection and union contracts where each operator has a single rule of decomposition. We provide a blame assignment algorithm that supports uniform rules of decomposition.
- Presenting the first monitoring-oriented semantics for contract satisfaction that supports intersection and union. We define contract satisfaction using blame assignment.
- Applying sound gradual typing to libraries in the DefinitelyTyped repository. We monitor the library-client boundary using contracts and measure conformance to the definition file. There were 122 libraries that fulfilled our evaluation criteria, of which 62 exhibited violations of conformance.
- Conducting a systematic evaluation of non-interference for contracts used to implement sound gradual typing in JavaScript. We measure multiple forms of interference, including object identity altered by function wrappers and data sealed by polymorphic contracts. There were 122 libraries that fulfilled our evaluation criteria, of which 22 exhibited violations of non-interference.

1.3 Outline

This thesis is structured as follows:

- Chapter 2 reviews the fundamental concepts in contract monitoring and blame assignment. We start with higher-order function contracts and the work of Findler and Felleisen (2002), then we review higher-order intersection and union contacts and the work of Keil and Thiemann (2015a).

- Chapter 3 presents the untyped lambda calculus extended with intersection and union contracts. We give an operational semantics and blame algorithm characterised by blame assignment and blame resolution.

- Chapter 4 describes a new definition of contract satisfaction using blame assignment. The programs that satisfy a contract are those that are never assigned blame when monitored using that contract. We present a new contract soundness criterion and provide a series of sound monitoring properties that a contract system should satisfy.

- Chapter 5 puts our theory into practice. We introduce *The Prime Directive*: a tool that monitors libraries and clients for conformance to a TypeScript definition file. We show how JavaScript proxies are used to implement higher-order contracts including parametric polymorphic contracts.

- Chapter 6 presents the results of applying *The Prime Directive* to 122 libraries in the DefinitelyTyped repository, recording all violations of conformance and non-interference. We discuss alternate contract designs aimed at ameliorating violations of non-interference in the presence of our results.

- Chapter 7 concludes this work by providing a summary our results and discussion of future work.

Chapter 2

Contracts and Blame

Blame has been an integral component to the development of contracts and gradual typing. From a theoretical perspective blame gives precise guarantees about where run-time type errors can occur. The phrase by Wadler and Findler (2009): *well-typed programs can't be blamed*, fundamentally relies on blame tracking to prove the innocence of statically typed code. From a practical perspective blame gives programmers insightful information when run-time errors occur such as indicating whether a module was at fault, or its client.

In this chapter we give an overview of existing work regarding blame for higher-order contracts: first, function contracts, then intersection and union contracts. The extension of contracts to support intersection and union reveals another vital role that blame has. Correctly implementing higher-order intersection and union contracts *depends* upon blame.

2.1 Anatomy of a Contract

In the purest sense a contract is the dynamic assertion of a program invariant; commonly, the invariant describes a type. A contract application can be written:

$$M@^{p}A$$

When describing these examples we let M and N range over program terms, V and W range over program values, p and q range over blame nodes, and A and B range over types. There are four key constituents that make up a contract application $M@^{p}A$ and we describe each in turn.

Subject The subject of a contract, in this example M. A contract application will assert that the subject conforms to the specification of the contract.

Blame Tracking The blame node of a contract, in this example p. A contract application uses a blame node to track the provenance of a contract and correctly assign blame should the contract fail. In the context of this work we use the term *provenance* to informally refer to the origin of a contract, and not as a direct reference to the formal study of provenance (Bose and Frew, 2005; Cheney, 2011; Cheney et al., 2009a,b; Simmhan et al., 2005). In section 3.4 we discuss existing work on provenance in relation to the form of blame tracking we present.

The literature of blame tracking contains different representations and descriptions for the blame node annotating a contract. Findler and Felleisen (2002) use two *obligation* variables: one to track the subject and one to track consumers of the subject. Wadler and Findler (2009) use a single blame *label* with a polarity. We use a single *blame node*. When extending contracts to intersection and union types simple obligations or labels are insufficient. Instead, we use a blame node that represents contract provenance and keeps a trace of evaluation; the detail of blame nodes will be revealed at the relevant points.

Contract The contract itself, in this example A. A contract describes a specification or obligation to enforce. Contracts can be constructed from arbitrary boolean returning functions, and then composed using a set of contract operators or combinators. In this work we focus on contracts that describe types, using a fixed set of base types that can be composed using operators. In the coming examples we write `I`, `B`, and `S` to denote contracts for integers, booleans, and strings; $\rightarrow$ to denote the function operator; and $\cap$ and $\cup$ to denote the intersection and union operators.

Context The last key component was hiding in plain sight. A contract application does not happen in a vacuum, but in a *context*. To make this explicit we write the contract application as a configuration:

$$\langle K, M@^{p}A \rangle$$

We write K to denote a continuation that consumes the result of evaluating a term. Alternate systems may choose to use evaluation contexts instead, written as $\mathcal{E}[M@^{p}A]$. The context of a contract is important to distinguish because some contracts place

obligations on the context, in addition to the subject. In some examples we choose to omit the context from a reduction, implicitly assuming the existence of the context.

Putting Them Together A basic contract application will evaluate the subject and ensure that the subject fulfils the contract. For example:

$$42@^{p}\mathtt{I} \longrightarrow 42$$

Applying the integer contract to 42 will immediately return the value because the contract is satisfied. However, consider another example:

$$42@^{p}\mathtt{B} \longrightarrow \mathtt{blame}\,p$$

The integer value does not conform to the boolean contract and blame is correspondingly assigned to the blame node p.

These examples are simple programs using simple contracts. In reality a programmer would like to build rich contracts for more complex programs. The primary way to do this is by using contract operators to compose contracts; function contracts are exemplary of contract composition.

2.2 Higher-order Function Types

Checking conformance of a value against a function contract is, in general, undecidable. A function contract is unable to guarantee that a value conforms to a given function type, but a function contract can ensure that every use of that function conforms to the type. This is achieved by *wrapping* (Findler and Felleisen, 2002).

Contract Monitoring A function contract will bond to a value, creating a wrapped function. When a wrapped function is applied the contract will check that the argument conforms to the domain of the contract, and also that the result of the application conforms to the codomain of the contract. For example, the wrap rule for a function contract evaluates as follows:

$$(V@A \to B)\,W \longrightarrow (V\,(W@A))@B$$

Blame tracking is temporarily omitted, and will be restored later. The *wrap* rule first ensures that argument W is checked against domain type A, then function V is applied, finally the application result is checked against codomain type B.

Functional and multi-paradigm programming makes heavy use of first-class functions. Contracts extend to this programming style by composing function contracts to build specifications for higher-order functions. For example:

$$(\lambda f.f\ 1 > 0)(\lambda y.y)$$

The expression applies a higher order function to the identity function. Argument f is applied to 1 and the result is checked to determine if the value is positive. A contract for this function can be defined using the higher order function contract $(\mathtt{I} \to \mathtt{I}) \to \mathtt{B}$.

Example ▷ Wrapping Higher-order Functions Evaluating wrapped functions can trigger multiple invocations of the wrap rule when higher-order values and contracts are used.

$$\begin{aligned}
& ((\lambda f.f\ 1 > 0)@(\mathtt{I} \to \mathtt{I}) \to \mathtt{B})(\lambda y.y) \\
\overset{wrap}{\longrightarrow}\ & ((\lambda f.f\ 1 > 0)((\lambda y.y)@\mathtt{I} \to \mathtt{I}))@\mathtt{B} \\
\longrightarrow\ & (((\lambda y.y)@\mathtt{I} \to \mathtt{I})\ 1 > 0)@\mathtt{B} \\
\overset{wrap}{\longrightarrow}\ & (((\lambda y.y)\ (1@\mathtt{I}))@\mathtt{I} > 0)@\mathtt{B} \\
\longrightarrow\ & (((\lambda y.y)\ 1)@\mathtt{I} > 0)@\mathtt{B} \\
\longrightarrow\ & (1@\mathtt{I} > 0)@\mathtt{B} \\
\longrightarrow\ & (1 > 0)@\mathtt{B} \\
\longrightarrow\ & \mathtt{true}@\mathtt{B} \\
\longrightarrow\ & \mathtt{true}
\end{aligned}$$

The top-level function contract $(\mathtt{I} \to \mathtt{I}) \to \mathtt{B}$ wraps the argument $\lambda y.y$ with another function contract $\mathtt{I} \to \mathtt{I}$. When the wrapped argument is applied to 1 another wrapping will take place. The example illustrates a successful evaluation where the function and its argument both conform to the contract. Reality may not be so perfect. A programmer could write code that fails to conform to a contract, and in that event, a violation should be raised that *blames* the party at fault.

Blame Blame assignment for higher-order functions was first presented by Findler and Felleisen (2002). Their work identified that function contracts have two parties and should therefore have two sets of obligations. The two parties to a contract are the *subject* and the *context*; the obligations are sometimes referred to as server and

client obligations, or positive and negative obligations. We adopt the latter. Correspondingly, when the subject violates the contract positive blame is assigned, and when the context violates the contract negative blame is assigned.

The dichotomy of contract obligations has been present since the inception of contracts: *The precondition binds the client; the postcondition binds the class* (Meyer, 1988, 1992). Tracking obligations for first-order functions with preconditions and postconditions is straightforward because errors are localised to the application of the guarded function. Applying the following wrapped function to argument `true` will raise an error at the function call-site because of the violated domain contract, and the client is always at fault for violations of the domain contract.

$$(\lambda x.x > 0)@\mathtt{I} \to \mathtt{B}$$

Applying the following wrapped function to any legal argument will raise an error at the function call-site because of the violated codomain contract, and the subject is always at fault for violations of the codomain contract.

$$(\lambda x.x + 1)@\mathtt{I} \to \mathtt{B}$$

When higher-order functions are introduced obligation (or blame) tracking becomes non-trivial. Errors may be deferred for higher-order contracts, occurring at a location beyond the initial call-site. Applying the following wrapped function will not trigger a violation until all the subsequent abstractions have been applied, which may happen at a location distant from the initial call-site.

$$(\lambda x.\lambda y.\lambda z.x + y + z)@\mathtt{I} \to \mathtt{I} \to \mathtt{I} \to \mathtt{B}$$

A further challenge associated with blame tracking for higher-order functions is that negative and positive obligations do not immediately partition in direct correspondence with the domain and codomain contracts. We may not immediately assign negative blame when the domain contract of function type $(\mathtt{I} \to \mathtt{I}) \to \mathtt{B}$ is violated because the contract $\mathtt{I} \to \mathtt{I}$ has positive *and* negative obligations.

A violation could occur because the context provides an illegal argument. In the following expression the context is obligated to provide an argument of type $\mathtt{I} \to \mathtt{I}$, though clearly $\lambda y.\mathtt{true}$ does not conform to this type.

$$((\lambda f.f\ 1 > 0)@(\mathtt{I} \to \mathtt{I}) \to \mathtt{B})\ (\lambda y.\mathtt{true})$$

A violation could also occur because the subject uses the argument in an illegal way, also read as the context of the argument violating the contract. In the following expression the subject is obligated to respect the negative obligations of domain contract $\mathtt{I} \to \mathtt{I}$, however supplying the argument `true` clearly violates the obligation.

$$((\lambda f.f\ \mathtt{true} > 0)@(\mathtt{I} \to \mathtt{I}) \to \mathtt{B})(\lambda y.y)$$

The insight by Findler and Felleisen (2002) was to propagate blame information (or obligations) through applications of wrapped functions. Blame propagates through the domain contract in a *contravariant* way, while blame propagates through the codomain contract in a *covariant* way. Findler and Felleisen (2002) implemented contravariant propagation by reversing the obligations associated with a contract. We now restore blame tracking to our examples.

$$(V@^{p,n} A \to B)\, W \longrightarrow (V\,(W@^{n,p} A))@^{p,n} B$$

Each contract has a pair of obligation identifiers, positive (p) and negative (n). When wrapping an application of a function contract the identifiers are swapped in the domain. In this work we adopt the style of Wadler and Findler (2009) that uses a single blame node with a polarity. Instead of swapping a pair of identifiers, we *negate*, or complement, a single blame node.

$$(V@^{p} A \to B)\, W \longrightarrow (V\,(W@^{-p} A))@^{p} B$$

The negation operation $-p$ on blame nodes is involutive such that $-(-p) \equiv p$. By negating the blame node we indicate that the obligations associated with that contract are reversed. The obligation to provide a legal argument to an application does not belong to the function itself, but the context of the function.

Example ▷ Wrapping Higher-order Functions with Blame Distinguishing positive and negative blame nodes and negating nodes as they flow through domain contracts make it possible to correct assign blame for higher order function contracts.

$$\begin{aligned}
& ((\lambda f.f\ \mathtt{true} > 0)@^{p}(\mathtt{I} \to \mathtt{I}) \to \mathtt{B})(\lambda y.y) \\
\overset{wrap}{\longrightarrow}\ & ((\lambda f.f\ \mathtt{true} > 0)((\lambda y.y)@^{-p}\mathtt{I} \to \mathtt{I}))@^{p}\mathtt{B} \\
\longrightarrow\ & (((\lambda y.y)@^{-p}\mathtt{I} \to \mathtt{I})\ \mathtt{true} > 0)@^{p}\mathtt{B} \\
\overset{wrap}{\longrightarrow}\ & (((\lambda y.y)(\mathtt{true}@^{p}\mathtt{I}))@^{-p}\mathtt{I} > 0)@^{p}\mathtt{B} \\
\longrightarrow\ & (((\lambda y.y)\,\mathtt{blame}\,p)@\mathtt{I} > 0)@\mathtt{B} \\
\longrightarrow\ & \mathtt{blame}\,p
\end{aligned}$$

The first invocation of the *wrap* rule bonds a contract of type $\mathtt{I} \to \mathtt{I}$ to the argument $\lambda y.y$, but importantly, negates the corresponding blame node p. The second invocation of the *wrap* rule applies the wrapped argument to boolean `true`. The blame node for the integer domain contract is negated, but observe that the blame node is *already* a negated blame node. The double negation means that the integer contract applied to argument `true` is annotated with blame node p, indicating an obligation that belongs to the subject of the initial contract. The integer contract fails, assigning blame to p. The fault of the violation is with the code *inside* the initial function contract annotated with p. In particular, the function body applies its argument of declared type $\mathtt{I} \to \mathtt{I}$ to a value of type boolean.

Example ▷ Higher-order Negative Blame Another example indicates how negative blame can arise from the use of higher-order contracts.

$$
\begin{array}{rl}
 & ((\lambda f.f\,1 > 0)@^{p}(\mathtt{I} \to \mathtt{I}) \to \mathtt{B})(\lambda y.\mathtt{true}) \\
\overset{wrap}{\longrightarrow} & ((\lambda f.f\,1 > 0)((\lambda y.\mathtt{true})@^{-p}\mathtt{I} \to \mathtt{I}))@^{p}\mathtt{B} \\
\longrightarrow & (((\lambda y.\mathtt{true})@^{-p}\mathtt{I} \to \mathtt{I})\,1 > 0)@^{p}\mathtt{B} \\
\overset{wrap}{\longrightarrow} & (((\lambda y.\mathtt{true})(1@^{p}\mathtt{I}))@^{-p}\mathtt{I} > 0)@^{p}\mathtt{B} \\
\longrightarrow & (((\lambda y.\mathtt{true})\,1)@^{-p}\mathtt{I} > 0)@^{p}\mathtt{B} \\
\longrightarrow & (\mathtt{true}@^{-p}\mathtt{I} > 0)@^{p}\mathtt{B} \\
\longrightarrow & (\mathtt{blame}\ -p > 0)@^{p}\mathtt{B} \\
\longrightarrow & \mathtt{blame}\ -p
\end{array}
$$

The first invocation of the *wrap* rule bonds a contract of type $\mathtt{I} \to \mathtt{I}$ to the argument $\lambda y.\mathtt{true}$, negating the corresponding blame node p. The second invocation of the *wrap* rule applies the wrapped argument to integer 1. The integer contract for argument 1 succeeds and β-reduction is applied to the inner application, yielding boolean `true`. The integer codomain contract for the argument function is applied to the result. As the codomain contract of the argument function is in a contravariant position relative to the initial function contract annotated with p, the integer contract is annotated with blame node $-p$. The integer contract fails, assigning blame to $-p$. The fault of the violation is with the context of the initial function contract annotated with p. In particular, the context provides a function which returns a boolean when then declared type is $\mathtt{I} \to \mathtt{I}$.

2.3 Dissecting Blame

Findler and Felleisen (2002) revealed many subtleties associated with propagating blame information during contract evaluation. The tracking of blame contains nuance, however the process of handling contract violations has been traditionally simple: when a contract fails the program halts without an opportunity for reparation. Adopting the approach that instantly throws a blame error will not work when extending contracts to support intersection and union, as we will show in Section 2.4 and Section 2.5. To facilitate the explanation and implementation of intersection and union contracts we dissect blame into two phases: *assignment* and *resolution*. Revealing these two concepts that were previously implicit in contract design also has benefits for systems that do not implement intersection or union.

When a contact $V@^{p}A$ is violated blame is *raised* on p. Existing systems of contracts or casts with blame considered raising blame as an atomic operation (Findler and Felleisen, 2002; Wadler and Findler, 2009). We consider the process to consist of two phases which we refer to as blame assignment and blame resolution. First is blame assignment. When a contract annotated with blame node p is violated then blame assignment is the process of determining whether p can be legitimately assigned fault, of blame, for the violation. If raising blame is raising a question of guilt, then assigning blame is delivering a guilty verdict. Second is blame resolution. When a contract annotated with blame node p is violated and blame is assigned to p, then blame resolution is the process of determining what to do with that blame. If raising blame is raising a question of guilt, then resolving blame is handing out the sentence.

In the existing literature assignment and resolution are implicitly implemented in an immediate fashion. When a contract is violated then blame assignment always ascribes fault to the implicated blame node; when a contract is violated then blame resolution always throws a blame error. This implementation of assignment and resolution is sound, but tightly coupled. Blame assignment may become incorrect if blame resolution is changed.

A contract implementation may choose not to throw an error when blame is assigned during a contract violation. We selected this approach when conducting the evaluation of gradual typing presented in Chapter 6 because we wanted to detect as many violations as possible in a single execution of a program, without having to fix code or amend contracts. Another example that warrants a different design are contracts implemented as asynchronous processes (Swords et al., 2018), where blame may

not be fully resolved before other violations are raised. We demonstrate how blame may be incorrectly assigned when changing resolution with the following example:

$$((\lambda f.f\,1)@^{p}(\mathtt{I}\to\mathtt{I})\to\mathtt{I})\,(\lambda y.\mathtt{true})$$

From a short inspection of the expression it should be apparent that the function $\lambda f.f\,1$ conforms to the type $(\mathtt{I}\to\mathtt{I})\to\mathtt{I}$, while the argument $\lambda y.\mathtt{true}$ does not conform to the type $\mathtt{I}\to\mathtt{I}$.

Example ▷ Resolution that Throws Starting with the traditional implementation of blame—where blame resolution throws an error—then blame is correctly assigned to blame node $-p$.

$$\begin{aligned}
& ((\lambda f.f\,1)@^{p}(\mathtt{I}\to\mathtt{I})\to\mathtt{I})\,(\lambda y.\mathtt{true}) \\
\longrightarrow\ & ((\lambda f.f\,1)\,((\lambda y.\mathtt{true})@^{-p}\mathtt{I}\to\mathtt{I}))@^{p}\mathtt{I} \\
\longrightarrow\ & (((\lambda y.\mathtt{true})@^{-p}\mathtt{I}\to\mathtt{I})\,1)@^{p}\mathtt{I} \\
\longrightarrow\ & (((\lambda y.\mathtt{true})\,(1@^{p}\mathtt{I}))@^{-p}\mathtt{I})@^{p}\mathtt{I} \\
\longrightarrow\ & (((\lambda y.\mathtt{true})\,1)@^{-p}\mathtt{I})@^{p}\mathtt{I} \\
\longrightarrow\ & \mathtt{true}@^{-p}\mathtt{I}@^{p}\mathtt{I} \\
\longrightarrow\ & \mathtt{blame}\,-p@^{p}\mathtt{I} \\
\longrightarrow\ & \mathtt{blame}\,-p
\end{aligned}$$

Example ▷ Resolution that Logs However suppose resolution is implemented differently, returning the subject of the contract and logging the violation instead.

$$\begin{aligned}
& ((\lambda f.f\,1)@^{p}(\mathtt{I}\to\mathtt{I})\to\mathtt{I})\,(\lambda y.\mathtt{true}) \\
\longrightarrow\ & ((\lambda f.f\,1)\,((\lambda y.\mathtt{true})@^{-p}\mathtt{I}\to\mathtt{I}))@^{p}\mathtt{I} \\
\longrightarrow\ & (((\lambda y.\mathtt{true})@^{-p}\mathtt{I}\to\mathtt{I})\,1)@^{p}\mathtt{I} \\
\longrightarrow\ & (((\lambda y.\mathtt{true})\,(1@^{p}\mathtt{I}))@^{-p}\mathtt{I})@^{p}\mathtt{I} \\
\longrightarrow\ & (((\lambda y.\mathtt{true})\,1)@^{-p}\mathtt{I})@^{p}\mathtt{I} \\
\longrightarrow\ & \mathtt{true}@^{-p}\mathtt{I}@^{p}\mathtt{I} \\
& \textbf{\textit{Log blame for:}}\ -p \\
\longrightarrow\ & \mathtt{true}@^{p}\mathtt{I} \\
& \textbf{\textit{Log blame for:}}\ p \\
\longrightarrow\ & \mathtt{true}
\end{aligned}$$

Blame is first assigned and logged for node $-p$ because the argument function $\lambda y.\texttt{true}$ is expected to return an integer, but returns a boolean. Execution continues. If we adopt the approach of unconditionally assigning blame then blame will be assigned and logged for node p because the outer function is expected to return an integer, but returns a boolean. The contract for type $(\texttt{I} \to \texttt{I}) \to \texttt{I}$ assigns positive blame to the function: an incorrect guilty verdict.

Always assigning blame is not a sound approach in general because it depends upon program execution terminating at the first violation, preventing subsequent contract violations. A more general approach is inspired by viewing function types as logical implication (Curry, 1934). A function contract for type $A \to B$ should only enforce the codomain contract B under the assumption that the domain contract A is satisfied. In other words: a subject is only required to conform to a function contract under the assumption that the context conforms to the same contract. In our example, assigning negative blame to $-p$ indicates that the context has violated the contract, and consequently, the subject should not longer be expected to conform to that contract. Assigning blame to $-p$ renders blame assignment to p void.

In Chapter 3 we give a semantics for blame that tracks violations by the context to ensure that blame is correctly assigned to the subject. When doing so, care has to be taken to distinguish multiple applications of the same function. This is achieved by enriching the structure of blame nodes to record the provenance of the contract they annotate. Our blame semantics is not the first to track violations by the context when assigning blame; Keil and Thiemann (2015a) use the technique when implementing intersection and union contracts. Our characterisation of blame in terms of assignment and resolution is novel, and we argue the distinction is clarifying for contract design in general.

2.4 Intersection Types

Intersection types have long history and were first introduced by Coppo and Dezani-Ciancaglini (1978) and Coppo et al. (1981); recently, intersection types have seen a renaissance in modern programming languages. They have proven to be a popular mechanism for describing multiple inheritance and mixins, and are used in languages such as Ceylon (Muehlboeck and Tate, 2018) and Scala (Rompf and Amin, 2016). Intersection types have also been popular in languages that widely exploit structural subtyping to assign types to previously untyped code, including TypeScript (Bierman

et al., 2014) and Flow for JavaScript (Chaudhuri et al., 2017).

A further use of intersection types is describing overloaded functions (Pierce, 1992). A common pattern in untyped languages is to perform a dynamic type test on the arguments to a function. Each branch of the type test can return a different type, emulating the selection of an overload. For example:

```
1 function negateNumOrBool(x) {
2   return (typeof x === "number") ? -x : !x;
3 }
```

When the function `negateNumOrBool` is applied to a number the function will return the negation of that number; when the function `negateNumOrBool` is applied to a boolean the function will return the negation of that boolean. The type of `negateNumOrBool` can be describe using an intersection type $(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B})$. Intersection types make a natural addition to a contract library's repertoire of operators.

Contract Monitoring Keil and Thiemann (2015a) present the first account of contracts for higher-order intersection types with blame assignment. Their implementation of intersection contracts is not uniform, and requires contract normalisation to be dynamically applied. The normalisation process extracts all immediate contracts nested within an intersection and evaluates them first. Evaluating intersections of function contracts is delayed until function application. This behaviour is implemented using three rules:

$$
\begin{array}{lcl}
V@^{p}(\mathcal{K}[I] \cap B) & \longrightarrow & (V@^{p_1} I)@^{p_2}\mathcal{K}[B] \\
V@^{p}(Q \cap \mathcal{K}[I]) & \longrightarrow & (V@^{p_1} I)@^{p_2}\mathcal{K}[Q] \\
(V@^{p} Q \cap R)\, W & \longrightarrow & ((V@^{p_1} Q)@^{p_2} R)\, W \\
\text{where} \quad A, B ::= \mathit{flat}(M) \mid A \to B \mid A \cap B \mid A \cup B & & \\
\qquad\qquad Q, R ::= A \to B \mid Q \cap R & & \\
\qquad\qquad \mathcal{K} ::= \Box \mid \mathcal{K} \cap B \mid Q \cap \mathcal{K} & &
\end{array}
$$

The first rule extracts a base (or immediate) contract from within the left branch of an intersection. The second rules extracts a base contract from within the right branch of an intersection if the left branch consists only of intersections of function types. The third rules applies an intersection of function contracts, separately applying each branch. The intent of the three rules is to ensure that state for blame assignment is correctly aggregated across applications of intersection contracts.

Blame The design of intersection contracts and blame assignment should accurately reflect the static semantics of intersection types. Basing blame assignment on the behaviour of static type checking assists programmers to develop an intuition for how contracts behave, whilst also allowing contracts to be used in systems that employ gradual typing. Introduction (I) and elimination (E) rules for intersection types are presented below:

$$\text{I-}\cap\ \frac{M : A \qquad M : B}{M : A \cap B} \qquad \text{E-}\cap_1\ \frac{M : A \cap B}{M : A} \qquad \text{E-}\cap_2\ \frac{M : A \cap B}{M : B}$$

Introduction rules for a type govern how values of that type are produced. Elimination rules for a type govern how values of that type are consumed. Correspondingly, the introduction rules inform how positive blame is assigned for intersection contracts, while the elimination rules inform how negative blame is assigned for intersection contracts. Some static type systems such as those by Pierce (1992) and Davies and Pfenning (2000) favour the use of a subtyping relation to describe intersection types rather than explicit introduction and elimination rules. We present introduction and elimination forms to reinforce the correspondence with positive and negative blame assignment.

The introduction rule states that for a term, or subject, to conform to the intersection type $A \cap B$ then the subject must individually conform to both A and B. The elimination rules state that a context may consume an intersection type $A \cap B$ at type A or type B, eliminating the other. An alternate reading is to say for a context to conform to an intersection type $A \cap B$ then the context must conform to A or B. While type rules pertain to conformance, blame assignment pertains to violation. In general, a contract cannot prove that a value conforms to a type, however, a contract can prove that a value fails to conform to a type by producing a counter-example. Using these rules, an interpretation of blame assignment for intersection can be described.

+ Positive blame is assigned to an intersection type $A \cap B$ if positive blame is assigned to A or positive blame is assigned to B. A subject that conforms to an intersection type must conform to both branches, and therefore assigning blame to one branch is sufficient to show that the subject does not conform to the intersection.

- Negative blame is assigned to an intersection type $A \cap B$ if negative blame is assigned to A and negative blame is assigned to B. A context that conforms to an

intersection type must conform to at least one branch, and therefore assigning blame to both branches is sufficient to show that the context does not conform to the intersection.

Positive Blame The presented definition delivers the expected behaviour for positive blame assignment.

$$((\lambda x.x > 3)@^{p}(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{I} \to \mathtt{B}))\,42$$

The function should satisfy both branches of the intersection and should therefore return a result that satisfies both codomain types. No value can satisfy both the integer and boolean contracts and consequently positive blame is assigned to the function. In this example the function returns a boolean that violates the integer contract, assigning positive blame to the left branch that propagates to the intersection.

Negative Blame When considering the behaviour of negative blame assignment observe that the definition permits *some* contract violations. Specifically, a negative blame violation may be allowed for one branch provided that the other branch is not assigned negative blame. We are no longer allowed to assume that blame resolution can *always* throw an error.

The presented definition appears to deliver the expected behaviour for negative blame assignment.

$$((\lambda x.x)@^{p}(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B}))\,42$$

The context should satisfy one branch of the intersection, and is free to violate the other; a negative violation can be viewed as an elimination. No value can satisfy both the integer and boolean contracts from the function domains, and consequently negative blame is assigned to function in the right branch. The negative blame assigned to the right branch denotes an elimination of the right branch; the context is choosing to use the type $\mathtt{I} \to \mathtt{I}$. When the function is applied the value 42 flows through the application to both codomain contracts. The boolean contract in the codomain of the right branch will attempt to raise blame because the contract is violated, but positive blame should *not* be assigned to the function. When the context decides to eliminate the right branch—by supplying an illegal value—the subject is no longer bound to the contract in that branch. Negative blame from the context makes assigning positive blame to the subject void. Here we see importance of distinguishing blame assignment which tracks past violations. Without using past knowledge of violations,

positive blame would be assigned to the right branch which then propagates to the intersection, wrongly indicating that the function does not conform to the type.

Suppose the example is changed to use a different argument:

$$((\lambda x.x)@^{p}(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B}))\,\text{"foo"}$$

In this instance negative blame is assigned to both branches of the intersection because argument "foo" does not conform to the integer and boolean contracts—negative blame is correspondingly assigned to the intersection.

A further subtlety arises when intersection contracts are used more than once (Keil and Thiemann, 2015a). Take the following example:

$$\begin{aligned}&\mathsf{let}\ f = (\lambda x.x)@^{p}(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B})\ \mathsf{in}\\ &\mathsf{if}\ f\,\mathtt{true}\ \mathsf{then}\ f\,1\ \mathsf{else}\ f\,0\end{aligned}$$

The example binds the contracted identity function to variable f, then applies f in multiple locations: first in the "if" condition and then in each branch. The first application of f uses a boolean argument that will cause negative blame to be assigned to the **left** branch of the intersection. The condition will evaluate to true and proceed to evaluate the "then" clause, applying f again. The second application of f uses an integer argument that will cause negative blame to be assigned to the **right** branch of the intersection. At this point negative blame has been assigned to **both** branches of the intersection: our current definition tells us that negative blame should be assigned to the intersection itself. This use of an intersection contract is valid because each application conforms to one of the branches; the proposed definition of negative blame is too naive.

Our diagnosis begins by observing that the negative blame violations of the intersection contract have been accumulated across all uses of the contract. To aggregate violations across all applications is to assume that the context chose which branch to use at the creation of the contract, therefore binding that choice to every application. This assumption is correct in *some* cases, but will not be correct in *all* cases.

Examining the static elimination rules again we see that the rule has no syntax direction and can be invoked at any point in a program. Furthermore, if we have a variable of intersection type that occurs in multiple locations then we are free to eliminate the variable in multiple ways. In the face of many choices the only option is to be as conservative as possible: an intersection contract must assume that the elimination choice is made as late as possible, and each elimination choice is fresh. The corrected definition of negative blame assignment is as follows:

- Negative blame is assigned to an intersection type $A \cap B$ if negative blame is assigned to A and negative blame is assigned to B in the *same* elimination context.

With this definition the example that includes multiple applications behaves as one would expected.

$$\begin{aligned}&\mathsf{let}\ f = (\lambda x.x)@^{p}(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B})\ \mathsf{in}\\&\mathsf{if}\ f\ \mathtt{true}\ \mathsf{then}\ f\ 1\ \mathsf{else}\ f\ 0\end{aligned}$$

The first application of f uses a boolean argument that will cause negative blame to be assigned to the **left** branch of the intersection in the **first** application. The condition will evaluate to true and proceed to evaluate the "then" clause, applying f again. The second application of f uses an integer argument that will cause negative blame to be assigned to the **right** branch of the intersection in the **second** application. At this point negative blame has been assigned to both branches of the intersection, however blame has not been assigned to both branches in the *same* elimination context. The second application successfully evaluates and the expression reduces to the integer 1.

The behaviour of aggregating negative blame *per* application motivates the monitoring rules by Keil and Thiemann (2015a). An intersection of function contracts is only decomposed at the point of application, generating fresh monitoring state. The cost of this design is that multiple rules are required. In Chapter 3 we present an alternate semantics that relies on more detailed blame tracking to implement intersection contracts using one uniform rule.

2.5 Union Types

Union types (Barbanera et al., 1995) are the natural dual to intersection types; intersection represents certainty, while union represents uncertainty. It is the uncertainty denoted by union types that has led them to become a popular feature in languages that bridge the gap between static and dynamic languages, such as Typed Racket (Tobin-Hochstadt and Felleisen, 2006), TypeScript (Bierman et al., 2014), and Flow for JavaScript (Chaudhuri et al., 2017). A conditional expression that returns values of one type in one branch, and values of another type in the other branch, can be assigned a type that is the union of both branch types.

When referring to union types we mean untagged unions, or "true" unions, rather than tagged unions, or variants. Untagged unions are suitable for describing untyped

code as they do not require values to be guarded by data constructors, or tags. Variants may be encoded using untagged unions by assigning a tag field with a unique type to each branch of the union.

A common pattern in untyped languages is to have a conditional expression with values of different types in each branch.

```
1 function zeroAsString(x) {
2   return x ? "0" : 0;
3 }
```

The function `zeroAsString` accepts an argument `x` and returns `"0"` when `x` is true and `0` when `x` is false. A contract of type $\mathsf{B} \to (\mathsf{S} \cup \mathsf{I})$ can be used to guard this function. We may add even more union types to this contract! Some systems encode the boolean type as union of two singleton types, denoting true and false respectively (Kent et al., 2016). An alternate contract for function `zeroAsString` would be $(\mathsf{T} \cup \mathsf{F}) \to (\mathsf{S} \cup \mathsf{I})$. The specification for the function could be strengthened even further by using the intersection type $(\mathsf{T} \to \mathsf{S}) \cap (\mathsf{F} \to \mathsf{I})$, however in general these types are not equivalent.

Contract Monitoring Keil and Thiemann (2015a) paired the first presentation of higher-order intersection contracts with higher-order union contracts. There is a single monitoring rule for union contracts, though the rule is not a simple decomposition. Instead, the rule implements distribution of intersection over union:

$$
\begin{array}{lll}
V@^{p}\mathcal{K}[A \cup B] & \longrightarrow & (V@^{p_1}\mathcal{K}[A])@^{p_2}\mathcal{K}[B] \\
\text{where} & A, B ::= \mathit{flat}(M) \mid A \to B \mid A \cap B \mid A \cup B & \\
& Q, R ::= A \to B \mid Q \cap R & \\
& \mathcal{K} ::= \Box \mid \mathcal{K} \cap B \mid Q \cap \mathcal{K} &
\end{array}
$$

The monitoring rule prioritises decomposition of union contracts over intersection. When a union contract exists in a context of intersections the union is extracted and the constituents are monitored in distinct contexts. The blame identifiers p_1 and p_2 are related to p by a union constraint (which we omit from the presentation). In essence, this rule implements the distributive law:

$$(A \cup B) \cap C \equiv (A \cap C) \cup (B \cap C)$$

Blame To present blame assignment for higher-order union contracts we adopt the same approach as intersection contracts and present the associated static type rules.

We present the introduction and elimination rules by Dunfield and Pfenning (2003).

$$\text{I-}\cup_1 \frac{M : A}{M : A \cup B} \qquad\qquad \text{I-}\cup_2 \frac{M : B}{M : A \cup B}$$

$$\text{E-}\cup \frac{\Gamma \vdash M : A \cup B \qquad \Gamma, x : A \vdash \mathcal{E}[x] : C \qquad \Gamma, y : B \vdash \mathcal{E}[y] : C}{\Gamma \vdash \mathcal{E}[M] : C}$$

The introduction rules state that for a term, or subject, to conform to the union type $A \cup B$ then the subject must conform to A or B. The elimination rule states that for an elimination context to conform to the union type $A \cup B$ then the context must conform to A and B. This is where the uncertainty manifests. A context cannot be sure whether a value of type $A \cup B$ will conform to type A or type B, and must therefore be prepared to accept both. Using these rules, an interpretation of blame assignment for union can be described.

+ Positive blame is assigned to a union type $A \cup B$ if positive blame is assigned to A and positive blame is assigned to B. A subject that conforms to a union type must conform to at least one branch, and therefore assigning blame to both branches is sufficient to show that the subject does not conform to the union.

- Negative blame is assigned to a union type $A \cup B$ if negative blame is assigned to A or negative blame is assigned to B. A context that conforms to a union type must conform to both branches, and therefore assigning blame to one branch is sufficient to show that the context does not conform to the union.

The blame assignment rules for union are very close to the dual of intersection, as to be expected. The one subtlety that prevents assignment for union being implemented by negating intersection assignment is in the duality between introduction and elimination. Negative blame for intersection contracts is aggregated per use of the intersection, while positive blame for union contracts is aggregated across all uses of the union. This is because a value of intersection type can be eliminated in multiple places and in multiple ways, while a value of union type is only introduced once. Consequently, the choice of which branch to satisfy persists across all uses of a value of union type.

As was the case with intersection contracts, the use of union contracts means that we cannot assume blame resolution raises an error, terminating the program. For

example, no value can satisfy both the integer contract $\mathtt{I}$ and the boolean contract $\mathtt{B}$, however a value can satisfy the union of the two. Evaluating the contract $\mathtt{I} \cup \mathtt{B}$ will always result in at least one local violation, but that does not mean the program as a whole has the violated the contract.

Positive Blame A function contract with a union type in the codomain may return one of two types when applied.

$$((\lambda x.\mathsf{if}\ x\ \mathsf{then}\ \texttt{"0"}\ \mathsf{else}\ 0)@^{p}\mathtt{B} \to (\mathtt{S} \cup \mathtt{I}))\,\mathtt{true}$$

Applying the contracted function will evaluate the conditional and return the string "0". When evaluating the codomain contracts the string contract will succeed, while the integer contract will fail and assign positive blame to the right branch of the union. As only one branch has been assigned positive blame the conditions are not sufficient to assign positive blame to the union.

$$((\lambda x.\mathsf{if}\ x\ \mathsf{then}\ \mathtt{false}\ \mathsf{else}\ 0)@^{p}\mathtt{B} \to (\mathtt{S} \cup \mathtt{I}))\,\mathtt{true}$$

In this example the contracted function will return the result `false` to which the union contract is applied. The contract in the left branch of the union will be violated, assigning positive blame, however this is not sufficient to blame the union. The contract in the right branch of the union will then be violated, assigning positive blame. At this point both branches have been assigned positive blame and therefore the union is assigned positive blame. This correctly indicates that the function failed to return a value that satisfies the union type $\mathtt{S} \cup \mathtt{I}$.

If the example used a union of function types then the outcome would be similar.

$$((\lambda x.\mathsf{if}\ x\ \mathsf{then}\ \mathtt{false}\ \mathsf{else}\ 0)@^{p}(\mathtt{B} \to \mathtt{S}) \cup (\mathtt{B} \to \mathtt{I}))\,\mathtt{true}$$

The returned value `false` violates the codomain contract $\mathtt{S}$ in the left branch of the union and is assigned positive blame. The return value `false` also violates the codomain contract $\mathtt{I}$ in the right branch of the union and is assigned positive blame. After evaluation of the second contract both functions in the union have been assigned positive blame, and therefore positive blame is assigned to the union itself.

Unions of function types have a subtle and interesting property; a union of function types can require multiple applications to detect a violation (Keil and Thiemann, 2015a). Take the following example:

$$((\lambda x.\mathsf{if}\ x\ \mathsf{then}\ \texttt{"0"}\ \mathsf{else}\ 0)@^{p}(\mathtt{B} \to \mathtt{S}) \cup (\mathtt{B} \to \mathtt{I}))\,\mathtt{true}$$

Evaluating this expression will assign positive blame to the right branch of the union because the returned value "0" does not satisfy the type `I`, but positive blame will not be assigned to the union because the left branch is satisfied. Suppose we change the argument to the function:

$$((\lambda x.\text{if } x \text{ then "0" else } 0)@^{p}(\mathsf{B} \to \mathsf{S}) \cup (\mathsf{B} \to \mathsf{I}))\,\texttt{false}$$

Evaluating this expression will assign positive blame to the left branch of the union because the returned value 0 does not satisfy the type `S`, but positive blame will not be assigned to the union because the right branch is satisfied.

The function does not conform to the union type however no single application of the contracted function will be sufficient to detect a violation. Recall that when a union type is introduced the choice of branch is fixed; a value may not "flip-flop" between union branches. For a union of function types there is no way to detect this "flip-flop" in a single application, and consequently the contract may only assign positive blame when the function is applied twice:

$$\begin{aligned}&\text{let } f = (\lambda x.\text{if } x \text{ then "0" else } 0)@^{p}(\mathsf{B} \to \mathsf{S}) \cup (\mathsf{B} \to \mathsf{I}) \text{ in}\\ &(f\,\texttt{true}, f\,\texttt{false})\end{aligned}$$

The first application assigns positive blame to the right branch but not the left; the contract assumes that the value has selected the left branch to satisfy. The second application assigns positive blame to the left branch but not the right. When blame is aggregated across both applications then both the left and right branches have been assigned positive blame, and therefore the union is assigned positive blame. A union of function contracts is not free to satisfy different branches at different applications.

The correct contract for this function is $\mathsf{B} \to (\mathsf{S} \cup \mathsf{I})$. Each application of the function produces a fresh union contract that may be satisfied by a different branch.

Negative Blame The nature of negative blame assignment for union contracts is more ruthless than the positive counterpart, making the decision process easier. For a union of function contracts the context must supply an argument that satisfies both domain types because the context cannot be sure which branch the value implements. For example:

$$(\lambda x.\text{if } x = 10 \text{ then "0" else } 0)@^{p}(\mathsf{Pos} \to \mathsf{S}) \cup (\mathsf{Even} \to \mathsf{I})$$

The contract Pos accepts positive integers and the contract Even accepts even integers. Any context that uses this function must provide an argument that is both positive and even, otherwise negative blame will be assigned to the context.

This restrictive nature of union contracts means that most practical contracts will be between functions that have the same domain type, for example $(\mathsf{B} \to \mathsf{S}) \cup (\mathsf{B} \to \mathtt{I})$. We cannot simplify this contract by pushing the union inwards, for example $\mathsf{B} \to (\mathsf{S} \cup \mathtt{I})$. As we have previously shown, these contracts have different behaviour.

Chapter 3

Contracts for Intersection and Union Types

In this chapter we take the intuition regarding blame assignment that we developed in Chapter 2 and make it concrete. We extend the untyped lambda calculus to support contracts for higher-order intersection and union types.

We define the calculus $\lambda^{\cap\cup}$, an untyped lambda calculus with contracts for intersection and union types. The primary result of our calculus is a pair of monitoring rules for intersection and union contracts that uniformly decompose. To enable uniform monitoring we enrich the structure of blame nodes to record a history of the evaluation that led to the construction of a given contract. When defining blame we make explicit the phases of assignment and resolution, as characterised in Chapter 2. We begin by defining the syntax and operational semantics for $\lambda^{\cap\cup}$, then we add the semantics of blame.

3.1 Syntax

The syntax of $\lambda^{\cap\cup}$ is given in Figure 3.1. Our calculus is based on the *CK* machine by Felleisen and Friedman (1986), a choice motivated by the presentation of contract semantics defined in Chapter 4. We distinguish four broad syntactic categories: contract types, program terms, blame tracking, and program configurations. We discuss each in turn.

Contract Types Let A and B range over contract types. A contract type is either an intersection type $A \cap B$, a union type $A \cup B$, a function type $A \to B$, a base type ι,

Types	A, B	::=	$A \cap B \mid A \cup B \mid A \to B \mid \iota \mid \mathsf{any}$
Base Types	ι	::=	$\mathtt{I} \mid \mathtt{B}$
Terms	M, N	::=	$x \mid k \mid \lambda x.M \mid M\,N \mid \mathtt{blame}\ \pm\ell \mid M@^{p}A$
Values	V, W	::=	$k \mid \lambda x.M \mid V@^{p}A \to B$
Frames	F	::=	$\Box\, N \mid V\, \Box \mid \Box\, @^{p}A$
Continuations	K	::=	$Id \mid K \circ F$
Wrap Indices	$n \in \mathbb{N}$		
Blame Labels	ℓ		
Branch Directions	d	::=	$\mathsf{left} \mid \mathsf{right}$
Branch Types	$\diamond$	::=	$\cap \mid \cup$
Blame Contexts	c	::=	$\mathsf{dom} \mid \mathsf{cod}$
Blame Paths	P	::=	$\mathsf{nil} \mid c_n / P$
Blame Nodes	$p, q \in \mathcal{P}$	::=	$\pm\ell[P] \mid p \bullet d^{\pm}_{\diamond}[P]$
Blame States	Φ	::=	$\emptyset \mid \{\ p\ \} \mid \Phi \cup \Phi'$
Context Trackers	Δ	=	$\mathcal{P} \rightharpoonup \mathbb{N}$
Configurations		::=	$\langle \Phi, \Delta, K, M \rangle$

Figure 3.1: Syntax

or the type any. Base types are either the integer type I or the boolean type B.

The type any matches all values and is analogous to the dynamic type in gradually typed cast languages, written $\star$ or ? (Siek and Taha, 2006; Wadler and Findler, 2009). In a language that supports user-defined contracts any can be defined using a predicate that always returns true. We include any in our language because it offers ergonomic benefits. In Chapter 4 we discuss splitting types into their positive and negative obligations and having access to any produces a succinct definition.

Another motivation for the type any is that programmers may wish to partially enforce types for a program, for example, adding a contract for a function codomain while leaving the domain unconstrained. This can be encoded with a function contract of the form $\mathsf{any} \to B$. Inlining the codomain contract might appear a tempting alternative to wrapping the function in a contract of type $\mathsf{any} \to B$, however inlining is not a general solution. For instance, the behaviour of the contract $(\mathsf{any} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B})$ cannot be replicated by inlining the domain and codomain contracts. For a function to satisfy this contract it must return an integer or diverge for all non-boolean ar-

guments, and diverge for all boolean arguments. Divergence is required because a boolean argument satisfies both domain types, therefore the function must return a value that satisfies both codomain types; no value satisfies the integer and boolean contracts so the only way to avoid blame is to diverge.

Program Terms Let M and N range over terms. A term is either a variable x, a constant k, an abstraction $\lambda x.M$, an application $M\,N$, a blame error $\texttt{blame}\ \pm\ell$, or a contract application $M@^{p}A$. A blame error specifies a blame label: an identifier associated with a violated contract from the source program. When we refer to source contracts, or top-level contracts, we mean contracts that were in the source program prior to evaluation. A source contract is not a sub-contract of any other contract.

Let V and W range over values. A value is either a constant k, an abstraction $\lambda x.M$, or a value wrapped in a function contract $V@^{p}A \to B$. Recall that a function contract cannot be immediately tested against a value for conformance. Instead, a function contract bonds to a value and wraps every application of the value.

Let F range over frames and let K range over continuations, where a continuation is a stack of frames that will consume a value. A frame is either an argument frame $\Box\,N$ that will consume a value, applying the value to argument N; a function frame $V\,\Box$ that will consume a value, passing the value as argument to V; or a contract frame $\Box\,@^{p}A$ that will consume a value, applying the contract A to the value. A continuation is either the identity continuation Id that will consume a value and immediately return the same value, or $K \circ F$, a continuation with a frame appended. Once frame F has consumed a value and the resulting term has been evaluated to a new value, that value is then consumed by the remaining continuation K.

The operational semantics are not tied to the use of continuations and can be defined in terms of evaluation contexts using the standard bijection between continuations K and evaluation contexts $\mathcal{E}$, defined by Felleisen and Friedman (1986).

Blame Tracking Let p and q range over blame nodes where a blame node denotes the provenance of a contract. We write $\mathcal{P}$ for the set of all blame nodes. A blame node is either a root node $\pm\ell[P]$ or a branch node $p \bullet d^{\pm}_{\diamond}[P]$. The terms root and branch refer to the interpretation of contract types as binary trees. An intersection or union type denotes a node in a tree where the constituent types are connected via branches. A blame node denotes a path through the binary tree, syntactically represented as a list.

A root node $\pm\ell[P]$ features a polarity, positive (+) or negative (−), that encodes

the variance of the type associated with the contract. The variance of the type is taken with respect to the source contract from which it originated. When presenting blame nodes we make use of *contextual negation*. If we write $\pm\ell[P]$ for a root node that may be positive or negative, then we write $\mp\ell[P]$ for the same root node with the polarity negated. The same behaviour applies to branch nodes. A root node also features a blame label ℓ drawn from a set of identifiers.

Every root node has a blame path. Let P range over blame paths where a path is either the empty path nil, or a blame context prepending a path. Let c range over blame contexts, dom denoting a domain contract, or cod denoting a codomain contract. Every blame context is indexed by a wrap index drawn from the set of naturals. The wrap index denotes how many times the function contract associated with the blame context has been applied. For example, dom_1 denotes the domain contract created in the second application of a particular function contract. A blame path encodes the provenance of some sub-contract of a function contract, detailing the wrap operations that led to the creation of the sub-contract. For example:

$$\mathsf{let}\ f = V@^{p}\mathsf{B} \to \mathtt{I} \to \mathsf{B}\ \mathsf{in}$$
$$(f\ \mathtt{true}\ 0, f\ \mathtt{false}\ 1)$$

Identifier f is bound to a wrapped function contract that is applied in each element of a tuple; the contract associated with f is the higher-order type $\mathsf{B} \to \mathtt{I} \to \mathsf{B}$. Application of f will produce sub-contracts for the domain and codomain types, each associated with a distinct blame path. The application of f to `true` wraps the argument in a boolean contract associated with the blame path $\mathsf{dom}_0/\mathsf{nil}$. The wrap index 0 indicates that this is the first (*0-th*) application of f. The application of f to `false` wraps the argument in a boolean contract associated with the blame path $\mathsf{dom}_1/\mathsf{nil}$. The wrap index 1 indicates that this is the second application of f. The function f returns a result that is also a wrapped function; the contract associated with the result is the type $\mathtt{I} \to \mathsf{B}$. The application of the result to 1 wraps the argument in an integer contract associated with the blame path $\mathsf{cod}_1/\mathsf{dom}_0/\mathsf{nil}$.

We assume that all source contracts are annotated with a positive root node with an empty blame path, as in $+\ell[\mathsf{nil}]$, where the blame label ℓ is unique within the program context.

A branch node $p \bullet d^{\pm}_{\diamond}[P]$ consists of a pointer to parent node p and information about the branch. Branch nodes annotate the sub-contracts of intersection or union contracts, and the parent p represents the blame node annotating the intersection or

union contract at the point the contract was decomposed. A branch node features a polarity, like a root node, that encodes the variance of the type associated with the contract. The variance of the type is taken with respect to the parent intersection or union contract from which it originated. Every branch has a direction d, either left or right, and is also indexed by the type of the parent contract, either intersection $\cap$ or union $\cup$, ranged over by $\diamond$. All branch nodes have a blame path, like a root node. The blame path of a branch node is taken with respect to the parent intersection or union contract from which it originated.

We present examples of contracts and the corresponding blame nodes that will be assigned to the sub-contracts. Starting with the source contract:

$$V@^{+\ell[\mathsf{nil}]}(\mathtt{B} \cup \mathtt{I}) \to ((\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B}))$$

we detail the contracts that arise from the function domain in the first application of the source contract, using wrap index 0 to indicate the first application.

Contract	Blame Node
$(\mathtt{B} \cup \mathtt{I}) \to ((\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B}))$	$+\ell[\mathsf{nil}]$
$\mathtt{B} \cup \mathtt{I}$	$-\ell[\mathsf{dom}_0/\mathsf{nil}]$
$\mathtt{B}$	$-\ell[\mathsf{dom}_0/\mathsf{nil}] \bullet \mathsf{left}^+_\cup[\mathsf{nil}]$
$\mathtt{I}$	$-\ell[\mathsf{dom}_0/\mathsf{nil}] \bullet \mathsf{right}^+_\cup[\mathsf{nil}]$

The node annotating the contract $\mathtt{B}$ has a positive polarity at the branch, but a negative polarity at the root. This is because the type $\mathtt{B}$ is in a positive position with respect to the type $\mathtt{B} \cup \mathtt{I}$, but a negative position with respect to the type $(\mathtt{B} \cup \mathtt{I}) \to ((\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B}))$.

We now detail the contracts that arise from the function codomain in the first application of the source contract, using wrap index 0 to indicate the first application.

Contract	Blame Node
$(\mathtt{B} \cup \mathtt{I}) \to ((\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B}))$	$+\ell[\mathsf{nil}]$
$(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B})$	$+\ell[\mathsf{cod}_0/\mathsf{nil}]$
$\mathtt{I} \to \mathtt{I}$	$+\ell[\mathsf{cod}_0/\mathsf{nil}] \bullet \mathsf{left}^+_\cap[\mathsf{nil}]$
$\mathtt{B} \to \mathtt{B}$	$+\ell[\mathsf{cod}_0/\mathsf{nil}] \bullet \mathsf{right}^+_\cap[\mathsf{nil}]$

We extend the table with the contracts that arise in the first application of the resulting function contracts, similarly using wrap index 0 to indicate the first application of the

resulting contracts.

Contract	Blame Node
$(\mathtt{B} \cup \mathtt{I}) \to ((\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B}))$	$+\ell[\mathsf{nil}]$
$(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B})$	$+\ell[\mathsf{cod}_0/\mathsf{nil}]$
$\mathtt{I} \to \mathtt{I}$	$+\ell[\mathsf{cod}_0/\mathsf{nil}] \bullet \mathsf{left}^+_\cap[\mathsf{nil}]$
$\mathtt{I}$ (domain)	$-\ell[\mathsf{cod}_0/\mathsf{nil}] \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]$
$\mathtt{I}$ (codomain)	$+\ell[\mathsf{cod}_0/\mathsf{nil}] \bullet \mathsf{left}^+_\cap[\mathsf{cod}_0/\mathsf{nil}]$
$\mathtt{B} \to \mathtt{B}$	$+\ell[\mathsf{cod}_0/\mathsf{nil}] \bullet \mathsf{right}^+_\cap[\mathsf{nil}]$
$\mathtt{B}$ (domain)	$-\ell[\mathsf{cod}_0/\mathsf{nil}] \bullet \mathsf{right}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]$
$\mathtt{B}$ (codomain)	$+\ell[\mathsf{cod}_0/\mathsf{nil}] \bullet \mathsf{right}^+_\cap[\mathsf{cod}_0/\mathsf{nil}]$

The blame paths are split between the root node and the branch node. The contract $\mathtt{I}$ is in the domain of the contract $\mathtt{I} \to \mathtt{I}$, which in turn exists within the codomain of the contract $(\mathtt{B} \cup \mathtt{I}) \to ((\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B}))$.

An observant reader may identify that the polarity of a root or branch node can be entirely reconstructed from the paths of the children. A node has a negative polarity if there are an odd number of domain contexts in the paths of all the children. We explicitly track the polarity because it makes the presentation clearer and avoids recomputing the polarity.

Program Configurations Let $\langle \Phi, \Delta, K, M \rangle$ denote a program configuration. A configuration is a tuple that consists of a blame state Φ, a context tracker Δ, a continuation K, and a term M.

A blame state is a set of blame nodes, where every node in the set has been assigned blame for a contract violation. We write $\emptyset$ for the empty blame state.

A context tracker is a partial map from blame nodes to wrap indices, where a mapping $p \mapsto n$ indicates that the function contract annotated with blame node p has been applied n times. The context tracker is not total with respect to function contracts in the program. If a blame node p is not in the domain of the context tracker it means that either p does not annotate a function contract, or the function contract has not been applied yet. We write $[\,]$ for the empty context tracker.

An initial program configuration always has the form $\langle \emptyset, [\,], \mathit{Id}, M \rangle$ where M represents the initial program.

3.2 Operational Semantics

Figure 3.2 presents the operational semantics for $\lambda^{\cap\cup}$, with auxiliary definitions presented in Figure 3.3. The operational semantics are presented as a binary relation ($\longrightarrow$) on program configurations, where one configuration *reduces* to another, written:

$$\langle \Phi, \Delta, K, M \rangle \longrightarrow \langle \Phi', \Delta', K', M' \rangle$$

When a component of a configuration is omitted from a reduction we assume that the component is unchanged in the reduction. Typically, we only choose to omit blame states and context trackers from a reduction, always presenting the continuation even if it remains the same. We write ($\longrightarrow^*$) to denote the reflexive and transitive closure of relation ($\longrightarrow$). We split reductions into four broad categories and we discuss each in turn.

Standard Reduction A configuration of the form $\langle K, M\,N \rangle$ reduces by creating an argument continuation that consumes a value, applying the value to N, and continues as continuation K. The configuration proceeds to evaluate M.

A configuration of the form $\langle K \circ \Box\, N, V \rangle$ reduces by creating a function continuation that consumes a value, applying V to the value, and continues as continuation K. The configuration proceeds to evaluate N.

A configuration of the form $\langle K \circ (\lambda x.M)\,\Box, V \rangle$ reduces by applying β-reduction that substitutes value V for bound variable x in function body M. We write $M[x := V]$ for the capture-avoiding substitution of x for V in M. The configuration proceeds to evaluate the function body (post substitution) with continuation K.

A configuration of the form $\langle K, M @^p A \rangle$ reduces by creating a contract continuation that consumes a value, applying the contract A to the value, and continues as continuation K. The reduction only happens under the condition that M is not a value. The configuration proceeds to evaluate M.

A configuration of the form $\langle K \circ \Box\, @^p A, V \rangle$ applies contract A to value V. The configuration will proceed to either decompose the contract or evaluate the contract.

Contract Decomposition A configuration of the form $\langle \Delta, K \circ (V @^p A \to B)\,\Box, W \rangle$ will invoke the *wrap* rule. First, the wrap index n is computed by the operation $\delta(\Delta, p)$ defined in Figure 3.3. The operation queries the context tracker for the wrap index

Reduction $\boxed{\langle \Phi, \Delta, K, M\rangle \longrightarrow \langle \Phi', \Delta', K', M'\rangle}$

$$
\begin{array}{lcl}
\langle K, M\,N\rangle & \longrightarrow & \langle K \circ \square\, N, M\rangle \\
\langle K \circ \square\, N, V\rangle & \longrightarrow & \langle K \circ V\, \square, N\rangle \\
\langle K \circ (\lambda x.M)\, \square, V\rangle & \longrightarrow & \langle K, M[x := V]\rangle \\
\langle K, M@^p A\rangle & \longrightarrow & \langle K \circ \square\, @^p A, M\rangle \\
\quad \text{if } M \neq V & & \\
\langle K \circ \square\, @^p A, V\rangle & \longrightarrow & \langle K, V@^p A\rangle \\
\langle \Delta, K \circ (V@^p A \to B)\, \square, W\rangle & \longrightarrow & \langle \Delta', K, (V\,(W@^{-p \gg \mathsf{dom}_n} A))@^{p \gg \mathsf{cod}_n} B\rangle \\
\quad \text{where } (\Delta', n) = \delta(\Delta, p) & & \\
\langle K, V@^p A \diamond B\rangle & \longrightarrow & \langle K, (V@^{p \bullet \mathsf{left}^+_\diamond[\mathsf{nil}]} A)@^{p \bullet \mathsf{right}^+_\diamond[\mathsf{nil}]} B\rangle \\
\langle K, V@^p \mathsf{any}\rangle & \longrightarrow & \langle K, V\rangle \\
\langle K, V@^p \iota\rangle & \longrightarrow & \langle K, V\rangle \\
\quad \text{if } V : \iota & & \\
\langle \Phi, K, V@^p \iota\rangle & \longrightarrow & \langle \Phi', K, M\rangle \\
\quad \text{otherwise, where } \Phi', M = \mathit{blame}(p, \Phi, V) & & \\
\langle K, \mathtt{blame}\ \pm\ell\rangle & \longrightarrow & \langle \mathit{Id}, \mathtt{blame}\ \pm\ell\rangle \\
\quad \text{if } K \neq \mathit{Id} & &
\end{array}
$$

Figure 3.2: Operational Semantics (Auxiliary Definitions in Figure 3.3)

of blame node p, returning an updated context tracker and the corresponding index. The new context tracker will increment the index associated with p, indicating that the function contract annotated with p has been applied an additional time. The wrap rule decomposes the function contract, wrapping the argument and result with their corresponding contracts and *extended* blame nodes. Blame node extension, written $\gg$, is defined in Figure 3.3. The operation extends a blame node with a new blame context to record the provenance of the sub-contract; extension appends a blame context to the path of a blame node. The blame node for the domain contract is negated to reflect the inversion of contract obligations while the blame node for the codomain contract retains the same polarity. Negation is defined in Figure 3.3 and recursively negates all polarities in a blame node. Like existing definitions for negation of blame labels (Wadler and Findler, 2009), negation for blame nodes is also an involution.

Proposition 3.2.1 (Involution)**.** $--p = p$.

A point of interest regarding function contracts is that they never check that the

Wrap Index $\boxed{\delta(\Delta, p)}$

$$\delta(\Delta, p) = \begin{cases} (\Delta[p \mapsto 1], 0) & \text{if } p \notin dom(\Delta) \\ (\Delta[p \mapsto n+1], n) & \text{where } n = \Delta(p) \end{cases}$$

Blame Node Negation $\boxed{-p}$

$$\begin{array}{lcl} -(\pm\ell[P]) & = & \mp\ell[P] \\ -(p \bullet d^{\pm}_{\diamond}[P]) & = & -p \bullet d^{\mp}_{\diamond}[P] \end{array}$$

Path Extension $\boxed{p \gg c_n \text{ and } P \gg c_n}$

$$\begin{array}{lcl} \pm\ell[P] \gg c_n & = & \pm\ell[P \gg c_n] \\ p \bullet d^{\pm}_{\diamond}[P] \gg c_n & = & p \bullet d^{\pm}_{\diamond}[P \gg c_n] \\ \mathsf{nil} \gg c_n & = & c_n/\mathsf{nil} \\ (c_n/P) \gg c'_n & = & c_n/(P \gg c'_n) \end{array}$$

Value Conformance $\boxed{V : A}$

$$\frac{V \in \mathbb{Z}}{V : \mathtt{I}} \qquad \frac{V \in \{\mathtt{true}, \mathtt{false}\}}{V : \mathtt{B}} \qquad \frac{V : \iota}{V@^{p}A \to B : \iota}$$

Figure 3.3: Auxiliary Definitions for Figure 3.2

wrapped value represents a lambda abstraction; function contracts only guard application behaviour. This choice is made because it simplifies the presentation and semantics by avoiding the need to juggle first-order and higher-order obligations when monitoring a function contract. The traditional interpretation of a function contract which asserts that the wrapped value *is* an abstraction can be implementing using an explicit composition of a first-order and higher-order check. Keil and Thiemann (2015a) make the same design decision in the presentation of their system. The alternate approach interprets $M@^{p}A \to B$ as checking both the first-order tag and higher-order contract. Under this approach $V@^{p}A \to B$ is no longer a value because the first-order component of the contract must be checked; if we wish to syntactically identify values we must introduce another function contract operator that denotes only the higher-order obligations. Consequently, the approach starts to resemble the encoding given Keil and Thiemann (2015a), but at the cost of introducing additional syntax and reduction rules. Our contract implementation discussed in Chapter 5 does use a single function operator that checks the first-order and higher-components—in practice we find that this is convenient for implementation and improves the contract

violation error messages.

A configuration of the form $\langle K, V@^p A \diamond B\rangle$ will immediately decompose the intersection or union contract, monitoring the value V against the constituent contracts. Each new contract is annotated with a branch node where p is the parent. The initial polarity of a branch node is positive because the branch monitors the same subject as the parent contract. The initial path of a branch node is empty because the type of the new contract is the immediate type in the corresponding branch of the intersection or union.

Contract Evaluation A configuration of the form $\langle K, V@^p \mathsf{any}\rangle$ will immediately evaluate the contract and produce a configuration of the form $\langle K, V\rangle$. The type any places no obligations on the subject or context therefore the contract is satisfied by all values.

A configuration of the form $\langle K, V@^p \iota\rangle$ will immediately evaluate the contract and produce a configuration of the form $\langle K, V\rangle$ when value V *conforms* to base type ι. Conformance, written $V : \iota$, is defined in Figure 3.3. An integer constant conforms to the type `I` and a boolean constant conforms to the type `B`. Function wrappers do not affect conformance. If a value V conforms to the type ι then $V@^p A \to B$ also conforms to the type ι. This choice is made with union contracts in mind. Given a term of the form:

$$4@^p(\mathtt{I} \to \mathtt{I}) \cup \mathtt{I}$$

then we would like the value 4 to be viewed as an integer, even with the function wrapper bound to the value. Removing the function wrapper is not an option because we must still monitor the context of the function contract, even if the subject is not a function. For example, given a term of the form:

$$(4@^p(\mathtt{I} \to \mathtt{I}) \cup \mathtt{I})\,\mathtt{true}$$

then we must still raise a violation because the context has inappropriately used the union contract. Consequently we view function wrappers as transparent with respect to conformance.

A configuration of the form $\langle \Phi, K, V@^p \iota\rangle$ will trigger a violation when V does not conform to ι, raising blame and appealing to the operation $\mathit{blame}(p, \Phi, V)$ defined in Figure 3.4. We describe the full semantics of blame in Section 3.3 and here we only consider the interface of the operation.

$$\mathit{blame} : (\mathcal{P} \times \mathit{BlameState} \times \mathit{Value}) \to (\mathit{BlameState} \times \mathit{Term})$$

The operation accepts a blame node, a blame state, and value; the informal meaning of $\mathit{blame}(p, \Phi, V)$ can be taken as:

Raise blame in state Φ *when value* V *violated a contract annotated with blame node* p.

The operation outputs a result consisting of an updated blame state Φ' and a term M; the informal meaning of the result (Φ', M) can be taken as:

Continue in state Φ' *where the result of the contract application is term* M.

The new blame state may differ from the input state by inclusion of p, or a prefix of p, when the blame node and any parent nodes have been assigned blame. The new term M is drawn from the set $\{V, \texttt{blame}\ \pm\ell\}$ where $\pm\ell$ corresponds to the polarity and blame label at the root of blame node p. When the violation is insufficient to trigger a blame error then V is returned as execution continues. When the violation indicates that a source contract has been violated then $\texttt{blame}\ \pm\ell$ is returned as execution halts.

Lifting A configuration of the form $\langle K, \texttt{blame}\ \pm\ell\rangle$ will discard the current continuation when K is not the identity continuation Id. The resulting continuation is of the form $\langle \mathit{Id}, \texttt{blame}\ \pm\ell\rangle$, lifting the blame error to the top-level.

3.3 Blame

Having defined the operational semantics that characterise the monitoring rules for contracts we now turn our attention to blame assignment and resolution: the process of interpreting contract violations. Figure 3.4 presents the semantics to blame for $\lambda^{\cap\cup}$, with auxiliary definitions presented in Figure 3.5. The entry point is the operation *blame*, as introduced in Section 3.2. The operation $\mathit{blame}(p, \Phi, V)$ will attempt to assign blame to node p in the current blame state, appealing to the operation $\mathit{assign}(p, \Phi)$. Blame assignment accepts a blame node and a blame state, returning a new blame state and truth value. The new blame state may differ from the input state by inclusion of p, or a prefix of p, when the blame node and any parent nodes have been assigned blame. The truth value specifies whether blame assignment propagated to a root node, indicating that a source contract has been violated. Blame assignment (*assign*) is a mutually recursive function defined in conjunction with blame resolution

Blame $\boxed{blame(p, \Phi, V)}$

$$blame(p, \Phi, V) = \begin{cases} \Phi', \texttt{blame}\ \pm\ell & \text{if } \Phi', \top = assign(p, \Phi),\ \pm\ell = root(p) \\ \Phi', V & \text{if } \Phi', \bot = assign(p, \Phi) \end{cases}$$

Blame Assignment $\boxed{assign(p, \Phi)}$

$$assign(p, \Phi) = \begin{cases} \Phi, \bot & \text{if } \exists q \in \Phi.\ compat(-p, q) \\ resolve(p, \Phi \cup \{p\}) & \text{otherwise} \end{cases}$$

Blame Resolution $\boxed{resolve(p, \Phi)}$

$$resolve(\pm\ell[P], \Phi) = \Phi, \top$$

$$resolve(p \bullet d^{+}_{\cap}[P], \Phi) = assign(parent(p \bullet d^{+}_{\cap}[P]), \Phi)$$

$$resolve(p \bullet d^{-}_{\cup}[P], \Phi) = assign(parent(p \bullet d^{-}_{\cup}[P]), \Phi)$$

$$resolve(p \bullet d^{-}_{\cap}[P], \Phi) = \begin{cases} assign(parent(p \bullet d^{-}_{\cap}[P]), \Phi) & \text{if } \exists P'.\ p \bullet flip(d)^{-}_{\cap}[P'] \in \Phi \\ & \quad \text{and } elim(P, P') \\ \Phi, \bot & \text{otherwise} \end{cases}$$

$$resolve(p \bullet d^{+}_{\cup}[P], \Phi) = \begin{cases} assign(parent(p \bullet d^{+}_{\cup}[P]), \Phi) & \text{if } \exists P'.\ p \bullet flip(d)^{+}_{\cup}[P'] \in \Phi \\ \Phi, \bot & \text{otherwise} \end{cases}$$

Figure 3.4: Blame Semantics (Auxiliary Definitions in Figure 3.5)

(*resolve*), both sharing the same interface:

$$assign : (\mathcal{P} \times BlameState) \rightarrow (BlameState \times \{\bot, \top\})$$
$$resolve : (\mathcal{P} \times BlameState) \rightarrow (BlameState \times \{\bot, \top\})$$

The operations call each-other in tandem as blame assignment propagates up through branch nodes. We first describe the semantics of blame assignment, then we describe the semantics of blame resolution.

3.3.1 Assignment

The purpose of blame assignment is to determine whether a blame node p can be rightfully considered at fault for a contract violation, or whether a previous violation

Compatibility $\boxed{compat(p,q) \text{ and } compat(P,P')}$

$$\frac{compat(P,P')}{compat(\pm\ell[P],\pm\ell[P'])} \qquad \frac{compat(P,P')}{compat(p \bullet d^{\pm}_{\diamond}[P], p \bullet d^{\pm}_{\diamond}[P'])}$$

$$\frac{compat(P,P')}{compat(c_n/P, c_n/P')} \qquad \frac{c \neq c'}{compat(c_n/P, c'_n/P')}$$

Matching Elimination $\boxed{elim(P,P')}$

$$\frac{}{elim(c_n/P, c'_n/P')}$$

Blame Node Parent $\boxed{parent(p)}$

$$\begin{array}{lcl} parent(p \bullet d^{\pm}_{\diamond}[\mathsf{nil}] \bullet d'^{\pm}_{\diamond'}[P']) & = & p \bullet d^{\pm}_{\diamond}[P'] \\ parent(p \bullet d^{\pm}_{\diamond}[P']) & = & p \quad \text{otherwise} \end{array}$$

Direction Flip $\boxed{flip(d)}$

$$\begin{array}{lcl} flip(\mathsf{left}) & = & \mathsf{right} \\ flip(\mathsf{right}) & = & \mathsf{left} \end{array}$$

Blame Node Root $\boxed{root(p)}$

$$\begin{array}{lcl} root(\pm\ell[P]) & = & \pm\ell \\ root(p \bullet d^{\pm}_{\diamond}[P]) & = & root(p) \end{array}$$

Figure 3.5: Auxiliary Definitions for Figure 3.4

by the context has invalidated the contract in question. Blame is not assigned to node p when there exists a node compatible with $-p$, representing the context of p, that has already been assigned blame. The condition is written:

$$\exists q \in \Phi.\ compat(-p, q)$$

Blame node q represents a violation by the context that relieves p of the duty to follow the contract. The binary relation *compat* is defined in Figure 3.5 and is used to relate blame nodes that originate from the same application of a function contract. Two blame nodes are compatible if they are identical up to path compatibility. Two blame paths are compatible if they share a prefix and then diverge at different blame contexts with the *same* wrap index, for example:

$$\frac{\frac{\mathsf{dom} \neq \mathsf{cod}}{\mathit{compat}(\mathsf{dom}_0/\mathsf{nil},\ \mathsf{cod}_0/\mathsf{nil})}}{\mathit{compat}(\mathsf{dom}_0/\mathsf{dom}_0/\mathsf{nil},\ \mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil})}$$

Compatibility is symmetric, but not reflexive. The intuition is that compatibility relates violations by the subject and context, and therefore a reflexive relation would undesirably relate violations from the same source. Similarly, the relation is not transitive because there are subject-context and context-subject pairs that are individually related, but cannot be collapsed to produce a related subject-subject pair. These properties inherit from the use of inequality in the definition of path compatibility.

Proposition 3.3.1. *The relation compat(p, q) is symmetric, but not reflexive or transitive.*

When a blame node compatible with $-p$ *does* exist then blame is *not* assigned to p and the operation returns the unmodified blame state and $\bot$, indicating that a source contract has not been violated. The particular blame polarity of p is not relevant, only the blame polarity in relation to some other blame node is significant. We could have also stated the condition as $\exists(q \in \Phi).\ \mathit{compat}(p, -q)$

Proposition 3.3.2. *compat$(-p, q)$ if and only if compat$(p, -q)$.*

When a blame node compatible with $-p$ *does not* exist then blame *is* assigned to p by adding p to the blame state. Having assigned blame to p, blame must now be resolved for p.

Preliminary Definitions We demonstrate blame assignment with a series of examples. To keep the examples concise we introduce additional reduction rules to avoid the tedious steps of pushing and popping frames when an application immediately involves two values. The new and admissible rules are:

Definition 3.3.3 (Simplified Reduction)**.**

$$\begin{array}{lcl}
\langle K, (\lambda x.M)\, V\rangle & \longrightarrow & \langle K, M[x := V]\rangle \\
\langle \Delta, K, (V @^{p} A \to B)\, W\rangle & \longrightarrow & \langle \Delta', K, (V\,(W @^{-p \gg \mathsf{dom}_n} A)) @^{p \gg \mathsf{cod}_n} B\rangle \\
\quad \textit{where } (\Delta', n) = \delta(\Delta, p) & &
\end{array}$$

Constructing an interesting example of blame assignment without intersection or union contracts is challenging because the program will terminate with a blame error at the first violation. To illustrate the intricacies of blame assignment, without having

to introduce the complexities of intersection and union, we introduce two definitions of blame.

The first definition of blame is the one we present in Figure 3.4, repeated here. We refer to this definition as *blame with exception semantics*.

Definition 3.3.4 (Blame with Exception Semantics)**.**

$$blame(p, \Phi, V) = \begin{cases} \Phi', \texttt{blame}\ \pm\ell & \text{if } \Phi', \top = assign(p, \Phi),\ \pm\ell = root(p) \\ \Phi', V & \text{if } \Phi', \bot = assign(p, \Phi) \end{cases}$$

The second definition is defined here. We refer to this definition as *blame with logging semantics*.

Definition 3.3.5 (Blame with Logging Semantics)**.**

$$blame_L(p, \Phi, V) = \Phi', V \quad \text{where } \Phi', b = assign(p, \Phi),\ b \in \{\bot, \top\}$$

The operation logs violations in the blame state, as was the existing behaviour, however now the truth value is ignored and the operation always returns the value V. In each example we make explicit which definition we are using.

Example ▷ Simple Blame Assignment The first example, presented in Figure 3.6, illustrates a simple case of higher-order blame assignment where a function contract is violated and a blame error is correspondingly raised. Each reduction is discussed in turn.

Reduction (3.1a) applies the *wrap* rule to the application. The context tracker for the program configuration is updated to indicate that the wrapped function annotated with blame node a, or $+\ell[\mathsf{nil}]$, has now been applied once. The blame nodes that annotate the function sub-contracts have had their blame paths extended to track the application context. Reduction (3.1b) creates a contract continuation that applies an integer contract annotated with blame node c, or $+\ell[\mathsf{cod}_0/\mathsf{nil}]$. Reduction (3.1c) applies β-reduction, substituting the wrapped function into the body of abstraction $\lambda y.y\,\texttt{false}$. Reduction (3.1d) applies the *wrap* rule again. In this instance the context tracker is updated with a mapping for blame node b, or $-\ell[\mathsf{dom}_0/\mathsf{nil}]$. Reduction (3.1e) creates another contract continuation, also of integer type, however the annotating blame node is e, or $-\ell[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]$. Reduction (3.1f) creates an argument continuation by pushing the unevaluated function argument $\texttt{false@}^d\mathsf{B}$ to the frame stack. Reduction (3.1g) pops the argument from the frame stack, then pushes

$$
\begin{array}{lll}
 & \textit{let } a = +\ell[\mathsf{nil}] & \\
 & \langle \emptyset, [\,], Id, ((\lambda y.y\,\mathtt{false})@^{a}(\mathtt{B}\to\mathtt{I})\to\mathtt{I})\,\lambda x.\mathtt{true}\rangle & \\
 & \textit{let } b = -\ell[\mathsf{dom}_0/\mathsf{nil}];\quad c = +\ell[\mathsf{cod}_0/\mathsf{nil}] & \\
\longrightarrow & \langle \emptyset, [a\mapsto 1], Id, ((\lambda y.y\,\mathtt{false})\,((\lambda x.\mathtt{true})@^{b}\mathtt{B}\to\mathtt{I}))@^{c}\mathtt{I}\rangle & (3.1a)\\
\longrightarrow & \langle \emptyset, [a\mapsto 1], Id\circ\square@^{c}\mathtt{I}, (\lambda y.y\,\mathtt{false})\,((\lambda x.\mathtt{true})@^{b}\mathtt{B}\to\mathtt{I})\rangle & (3.1b)\\
\longrightarrow & \langle \emptyset, [a\mapsto 1], Id\circ\square@^{c}\mathtt{I}, ((\lambda x.\mathtt{true})@^{b}\mathtt{B}\to\mathtt{I})\,\mathtt{false}\rangle & (3.1c)\\
 & \textit{let } d = +\ell[\mathsf{dom}_0/\mathsf{dom}_0/\mathsf{nil}];\quad e = -\ell[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}] & \\
\longrightarrow & \langle \emptyset, [a\mapsto 1; b\mapsto 1], Id\circ\square@^{c}\mathtt{I}, ((\lambda x.\mathtt{true})\,(\mathtt{false}@^{d}\mathtt{B}))@^{e}\mathtt{I}\rangle & (3.1d)\\
\longrightarrow & \langle \emptyset, [a\mapsto 1; b\mapsto 1], Id\circ\square@^{c}\mathtt{I}\circ\square@^{e}\mathtt{I}, (\lambda x.\mathtt{true})\,(\mathtt{false}@^{d}\mathtt{B})\rangle & (3.1e)\\
\longrightarrow & \langle \emptyset, [a\mapsto 1; b\mapsto 1], Id\circ\square@^{c}\mathtt{I}\circ\square@^{e}\mathtt{I}\circ\square\,(\mathtt{false}@^{d}\mathtt{B}), \lambda x.\mathtt{true}\rangle & (3.1f)\\
\longrightarrow & \langle \emptyset, [a\mapsto 1; b\mapsto 1], Id\circ\square@^{c}\mathtt{I}\circ\square@^{e}\mathtt{I}\circ(\lambda x.\mathtt{true})\,\square, \mathtt{false}@^{d}\mathtt{B}\rangle & (3.1g)\\
\longrightarrow & \langle \emptyset, [a\mapsto 1; b\mapsto 1], Id\circ\square@^{c}\mathtt{I}\circ\square@^{e}\mathtt{I}\circ(\lambda x.\mathtt{true})\,\square, \mathtt{false}\rangle & (3.1h)\\
\longrightarrow & \langle \emptyset, [a\mapsto 1; b\mapsto 1], Id\circ\square@^{c}\mathtt{I}\circ\square@^{e}\mathtt{I}, \mathtt{true}\rangle & (3.1i)\\
\longrightarrow & \langle \emptyset, [a\mapsto 1; b\mapsto 1], Id\circ\square@^{c}\mathtt{I}, \mathtt{true}@^{e}\mathtt{I}\rangle & (3.1j)\\
 & \textbf{\textit{Violation!}}\ \textit{blame}(e,\emptyset,\mathtt{true}) = \{e\}, \mathtt{blame}\ -\ell & \\
\longrightarrow & \langle \{e\}, [a\mapsto 1; b\mapsto 1], Id\circ\square@^{c}\mathtt{I}, \mathtt{blame}\ -\ell\rangle & (3.1k)\\
\longrightarrow & \langle \{e\}, [a\mapsto 1; b\mapsto 1], Id, \mathtt{blame}\ -\ell\rangle & (3.1l)
\end{array}
$$

Figure 3.6: Simple Blame Assignment (with Exception Semantics)

the evaluated abstraction, creating a function continuation. Reduction (3.1h) evaluates the boolean contract that is applied to the constant `false`, successfully validating the contract without incident. Reduction (3.1i) applies β-reduction. Reduction (3.1j) pops the first of the integer contracts from the frame stack, applying the contract to value `true`. Reduction (3.1k) evaluates the integer contract annotated with blame node e, or $-\ell[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]$. The value `true` does not conform to the contract `I` therefore a violation is triggered and the *blame* operation is invoked.

Violation! $\mathit{blame}(e, \emptyset, \mathtt{true}) = \{e\}, \mathtt{blame}\ -\ell$ as
$\quad \mathit{assign}(e, \emptyset) = \mathit{resolve}(e, \{e\})$ as $\nexists q \in \emptyset.\ \mathit{compat}(-e, q)$
$\quad \mathit{resolve}(e, \{e\}) = \{e\}, \top$
$\quad$ where $e = -\ell[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]$

Assignment is trivial because the blame state is empty and therefore no compatible node can be found. Resolution is similarly trivial because the blame node in question is a root. The result $\top$ is returned by the *resolve* operation indicating that a source contract has been violated therefore a blame error should be raised. The result of $\mathit{blame}(e, \emptyset, \mathtt{true})$ is a blame state that implicates e, and a blame error that implicates the context of the initial function contract. Reduction (3.1l) discards the remaining continuation as an error has been raised; the program configuration reaches a terminal state.

Example ▷ Blame Assignment with Logging Semantics We start with the same initial program however now we now use blame_L rather than *blame*. By inspecting the blame state at the end of evaluation we can observe all blame nodes that were assigned blame, or all violations that were "logged". The full reduction sequence is defined in Figure 3.7.

Reduction (3.2a) to (3.2j) proceed exactly as before. Reduction (3.2k) evaluates the integer contract annotated with blame node e, or $-\ell[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]$. The value `true` does not conform to the contract `I` therefore a violation is triggered and the blame_L operation is invoked.

Violation! $\mathit{blame}_L(e, \emptyset, \mathtt{true}) = \{e\}, \mathtt{true}$ as
$\quad \mathit{assign}(e, \emptyset) = \mathit{resolve}(e, \{e\})$ as $\nexists q \in \emptyset.\ \mathit{compat}(-e, q)$
$\quad \mathit{resolve}(e, \{e\}) = \{e\}, \top$
$\quad$ where $e = -\ell[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]$

Assignment is trivial because the blame state is empty and therefore no compatible node can be found. Resolution is similarly trivial because the blame node in question

$$
\begin{array}{lll}
 & \textit{let } a = +\ell[\mathsf{nil}] & \\
 & \langle \emptyset, [\,], Id, ((\lambda y.y\,\mathtt{false})@^{a}(\mathtt{B} \to \mathtt{I}) \to \mathtt{I})\,\lambda x.\mathtt{true}\rangle & \\
 & \textit{let } b = -\ell[\mathsf{dom}_0/\mathsf{nil}]; \quad c = +\ell[\mathsf{cod}_0/\mathsf{nil}] & \\
\longrightarrow & \langle \emptyset, [a \mapsto 1], Id, ((\lambda y.y\,\mathtt{false})\,((\lambda x.\mathtt{true})@^{b}\mathtt{B} \to \mathtt{I}))@^{c}\mathtt{I}\rangle & (3.2a)\\
\longrightarrow & \langle \emptyset, [a \mapsto 1], Id \circ \Box\,@^{c}\mathtt{I}, (\lambda y.y\,\mathtt{false})\,((\lambda x.\mathtt{true})@^{b}\mathtt{B} \to \mathtt{I})\rangle & (3.2b)\\
\longrightarrow & \langle \emptyset, [a \mapsto 1], Id \circ \Box\,@^{c}\mathtt{I}, ((\lambda x.\mathtt{true})@^{b}\mathtt{B} \to \mathtt{I})\,\mathtt{false}\rangle & (3.2c)\\
 & \textit{let } d = +\ell[\mathsf{dom}_0/\mathsf{dom}_0/\mathsf{nil}]; \quad e = -\ell[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}] & \\
\longrightarrow & \langle \emptyset, [a \mapsto 1; b \mapsto 1], Id \circ \Box\,@^{c}\mathtt{I}, ((\lambda x.\mathtt{true})\,(\mathtt{false}@^{d}\mathtt{B}))@^{e}\mathtt{I}\rangle & (3.2d)\\
\longrightarrow & \langle \emptyset, [a \mapsto 1; b \mapsto 1], Id \circ \Box\,@^{c}\mathtt{I} \circ \Box\,@^{e}\mathtt{I}, (\lambda x.\mathtt{true})\,(\mathtt{false}@^{d}\mathtt{B})\rangle & (3.2e)\\
\longrightarrow & \langle \emptyset, [a \mapsto 1; b \mapsto 1], Id \circ \Box\,@^{c}\mathtt{I} \circ \Box\,@^{e}\mathtt{I} \circ \Box\,(\mathtt{false}@^{d}\mathtt{B}), \lambda x.\mathtt{true}\rangle & (3.2f)\\
\longrightarrow & \langle \emptyset, [a \mapsto 1; b \mapsto 1], Id \circ \Box\,@^{c}\mathtt{I} \circ \Box\,@^{e}\mathtt{I} \circ (\lambda x.\mathtt{true})\,\Box, \mathtt{false}@^{d}\mathtt{B}\rangle & (3.2g)\\
\longrightarrow & \langle \emptyset, [a \mapsto 1; b \mapsto 1], Id \circ \Box\,@^{c}\mathtt{I} \circ \Box\,@^{e}\mathtt{I} \circ (\lambda x.\mathtt{true})\,\Box, \mathtt{false}\rangle & (3.2h)\\
\longrightarrow & \langle \emptyset, [a \mapsto 1; b \mapsto 1], Id \circ \Box\,@^{c}\mathtt{I} \circ \Box\,@^{e}\mathtt{I}, \mathtt{true}\rangle & (3.2i)\\
\longrightarrow & \langle \emptyset, [a \mapsto 1; b \mapsto 1], Id \circ \Box\,@^{c}\mathtt{I}, \mathtt{true}@^{e}\mathtt{I}\rangle & (3.2j)\\
 & \textbf{\textit{Violation!}}\ \mathit{blame}_L(e, \emptyset, \mathtt{true}) = \{e\}, \mathtt{true} & \\
\longrightarrow & \langle \{e\}, [a \mapsto 1; b \mapsto 1], Id \circ \Box\,@^{c}\mathtt{I}, \mathtt{true}\rangle & (3.2k)\\
\longrightarrow & \langle \{e\}, [a \mapsto 1; b \mapsto 1], Id, \mathtt{true}@^{c}\mathtt{I}\rangle & (3.2l)\\
 & \textbf{\textit{Violation!}}\ \mathit{blame}_L(c, \{e\}, \mathtt{true}) = \{e\}, \mathtt{true} & \\
\longrightarrow & \langle \{e\}, [a \mapsto 1; b \mapsto 1], Id, \mathtt{true}\rangle & (3.2m)
\end{array}
$$

Figure 3.7: Blame Assignment with Logging Semantics

is a root. The modified operation $blame_L$ ignores the truth value and returns the value `true` in a modified state that assigns blame to node e, or $-\ell[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]$. Reduction (3.2l) pops the second integer contract from the frame stack, applying the contract to value `true`. Reduction (3.2m) evaluates the integer contract annotated with blame node c, or $+\ell[\mathsf{cod}_0/\mathsf{nil}]$. The value `true` does not conform to the contract `I` therefore a violation is triggered and the $blame_L$ operation is invoked.

Violation! $blame_L(c, \{e\}, \mathtt{true}) = \{e\}, \mathtt{true}$ as
$assign(c, \{e\}) = \{e\}, \bot$ as $\exists q \in \{e\}.\ compat(-c, q)$
where $c = +\ell[\mathsf{cod}_0/\mathsf{nil}]$
$e = -\ell[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]$

Attempting to assign blame to node c involves querying the blame state for existing violations. We observe that the negation of blame node c, defined as $-(+\ell[\mathsf{cod}_0/\mathsf{nil}]) = -\ell[\mathsf{cod}_0/\mathsf{nil}]$, is compatible with node e that has previously been assigned blame.

$$\dfrac{\dfrac{\mathsf{cod} \neq \mathsf{dom}}{compat(\mathsf{cod}_0/\mathsf{nil},\ \mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil})}}{compat(-\ell[\mathsf{cod}_0/\mathsf{nil}],\ -\ell[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}])}$$

The context has violated the contract by supplying an argument that does not conform to the domain contract $\mathtt{B} \to \mathtt{I}$, as evidenced by the blame assigned to the blame node $-\ell[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]$.

A consequence of assigning blame to the context is that the subject is no longer obligated to follow the contract; no blame is assigned when the subject produces a value that violates the codomain contract `I`.

The existence of a compatible blame node means that blame is *not* assigned to node c; the existing blame state is returned without modification. Blame does not need to be resolved because blame was never assigned, therefore the operation *assign* returns $\bot$, and the operation $blame_L$ returns the same blame state and value `true`.

After reduction (3.2m) the program configuration reaches an irreducible form. The final blame state correctly logs the contract violations: the blame node e informs us that the context has failed to conform to the contract $(\mathtt{B} \to \mathtt{I}) \to \mathtt{I}$. Furthermore, blame was correctly not assigned to the subject of the contract, even though the codomain contract was violated.

Example ▷ Presumptuous Functions Blame assigned to the context of a function does not always absolve the function of blame. In particular, a function that presumes

$$
\begin{array}{lll}
 & \textit{let } a = +\ell[\mathsf{nil}] & \\
 & \langle \emptyset, [\,], \mathit{Id}, ((\lambda y.(y\,\mathtt{false}, y\,1))@^{a}(\mathsf{B} \to \mathtt{I}) \to \mathsf{any})\,\lambda x.\mathtt{true}\rangle & \\
 & \textit{let } b = -\ell[\mathsf{dom}_0/\mathsf{nil}]; \quad c = +\ell[\mathsf{cod}_0/\mathsf{nil}] & \\
\longrightarrow & \langle \emptyset, [a \mapsto 1], \mathit{Id}, ((\lambda y.(y\,\mathtt{false}, y\,1))\,((\lambda x.\mathtt{true})@^{b}\mathsf{B} \to \mathtt{I}))@^{c}\mathsf{any}\rangle & (3.3a) \\
\longrightarrow & \langle \emptyset, [a \mapsto 1], \mathit{Id} \circ \square\,@^{c}\mathsf{any}, (\lambda y.(y\,\mathtt{false}, y\,1))\,((\lambda x.\mathtt{true})@^{b}\mathsf{B} \to \mathtt{I})\rangle & (3.3b) \\
 & \textit{let } V = ((\lambda x.\mathtt{true})@^{b}\mathsf{B} \to \mathtt{I}) & \\
\longrightarrow & \langle \emptyset, [a \mapsto 1], \mathit{Id} \circ \square\,@^{c}\mathsf{any}, (V\,\mathtt{false}, V\,1)\rangle & (3.3c) \\
 & \textit{let } K = \mathit{Id} \circ \square\,@^{c}\mathsf{any} \circ (\square, V\,1) & \\
\longrightarrow & \langle \emptyset, [a \mapsto 1], K, V\,\mathtt{false}\rangle & (3.3d) \\
 & \textit{let } d = +\ell[\mathsf{dom}_0/\mathsf{dom}_0/\mathsf{nil}]; \quad e = -\ell[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}] & \\
\longrightarrow & \langle \emptyset, [a \mapsto 1; b \mapsto 1], K, ((\lambda x.\mathtt{true})\,(\mathtt{false}@^{d}\mathsf{B}))@^{e}\mathtt{I}\rangle & (3.3e) \\
\longrightarrow & \langle \emptyset, [a \mapsto 1; b \mapsto 1], K \circ \square\,@^{e}\mathtt{I}, (\lambda x.\mathtt{true})\,(\mathtt{false}@^{d}\mathsf{B})\rangle & (3.3f) \\
\longrightarrow & \langle \emptyset, [a \mapsto 1; b \mapsto 1], K \circ \square\,@^{e}\mathtt{I} \circ \square(\mathtt{false}@^{d}\mathsf{B}), \lambda x.\mathtt{true}\rangle & (3.3g) \\
\longrightarrow & \langle \emptyset, [a \mapsto 1; b \mapsto 1], K \circ \square\,@^{e}\mathtt{I} \circ (\lambda x.\mathtt{true})\,\square, \mathtt{false}@^{d}\mathsf{B}\rangle & (3.3h) \\
\longrightarrow & \langle \emptyset, [a \mapsto 1; b \mapsto 1], K \circ \square\,@^{e}\mathtt{I} \circ (\lambda x.\mathtt{true})\,\square, \mathtt{false}\rangle & (3.3i) \\
\longrightarrow & \langle \emptyset, [a \mapsto 1; b \mapsto 1], K \circ \square\,@^{e}\mathtt{I}, \mathtt{true}\rangle & (3.3j) \\
\longrightarrow & \langle \emptyset, [a \mapsto 1; b \mapsto 1], K, \mathtt{true}@^{e}\mathtt{I}\rangle & (3.3k) \\
 & \textbf{\textit{Violation!}}\ \mathit{blame}_L(e, \emptyset, \mathtt{true}) = \{e\}, \mathtt{true} & \\
\longrightarrow & \langle \{e\}, [a \mapsto 1; b \mapsto 1], K, \mathtt{true}\rangle & (3.3l)
\end{array}
$$

Figure 3.8: Presumptuous Functions: Part One (with Logging Semantics)

ill-behaviour of the context is not allowed: a function must assume that the context has honest intentions.

Compatibility is defined to only consider the longest prefix of two blame paths, or the most specific application context. The concrete intuition for this behaviour is to prevent different applications of the same function from interfering. Each application must be considered fresh.

We illustrate this behaviour with an example presented in Figure 3.8 and Figure 3.9. As was the case before, we use the $blame_L$ definition that only logs assigned blame. Before explaining the reduction sequence we highlight some key differences with previous examples. First, the body of the wrapped function returns a tuple with each element applying argument y. This choice is made to demonstrate the presumptuous behaviour when evaluating the right element. Second, the codomain of the function contract is `any` rather than `I`. This choice is made to avoid extending value conformance to include tuples; we can trivially evaluate the contract `any` without conformance checking. For this example we assume that the operational semantics are extended with the standard rules for tuples; we do not require tuple contracts or associated operations.

We start with explanation of Figure 3.8 that contains the reduction sequence up to the evaluation of the first tuple element in the body of the wrapped function. Reduction (3.3a) to reduction (3.3c) proceed as before. Reduction (3.3d) pushes a tuple frame that accepts a value to be inserted into the left element, then the configuration proceeds to evaluate the term from the left element. Reduction (3.3e) performs the first wrapping of V, or the wrapped argument function. As was the case before, new blame nodes with extended paths are created for the domain and codomain contracts. Reduction (3.3e) to reduction (3.3l) evaluate similarly to the previous example, with the only difference being that continuation K holds the other element of the tuple. Reduction (3.3l) ends having evaluated the first element of the tuple to `true`. The blame state currently implicates the context for providing an argument that does not conform to the contract $\mathsf{B} \to \mathtt{I}$. Evaluation resumes in Figure 3.9.

Reduction (3.3m) exchanges the tuple elements in the configuration; the evaluated term is placed in the frame stack and the term in the right element is selected for evaluation. Reduction (3.3n) applies the wrapped argument function again. The context tracker is updated to indicate that this is the *second* application of the function, and the new blame nodes f and g for the domain and codomain contracts are correspondingly extended. Reduction (3.3o) to reduction (3.3q) evaluate the applied

$$
\begin{array}{lll}
 & \textit{let } K' = \mathit{Id} \circ \square\,@^{c}\mathsf{any} \circ (\mathsf{true}, \square) & \\
\longrightarrow & \langle \{e\}, [a \mapsto 1; b \mapsto 1], K', V\,1 \rangle & (3.3\text{m}) \\
 & \textit{let } f = +\ell[\mathsf{dom}_0/\mathsf{dom}_1/\mathsf{nil}]; \quad g = -\ell[\mathsf{dom}_0/\mathsf{cod}_1/\mathsf{nil}] & \\
\longrightarrow & \langle \{e\}, [a \mapsto 1; b \mapsto 2], K', ((\lambda x.\mathsf{true})(1@^{f}\mathsf{B}))@^{g}\mathtt{I} \rangle & (3.3\text{n}) \\
\longrightarrow & \langle \{e\}, [a \mapsto 1; b \mapsto 2], K' \circ \square\,@^{g}\mathtt{I}, (\lambda x.\mathsf{true})(1@^{f}\mathsf{B}) \rangle & (3.3\text{o}) \\
\longrightarrow & \langle \{e\}, [a \mapsto 1; b \mapsto 2], K' \circ \square\,@^{g}\mathtt{I} \circ \square(1@^{f}\mathsf{B}), \lambda x.\mathsf{true} \rangle & (3.3\text{p}) \\
\longrightarrow & \langle \{e\}, [a \mapsto 1; b \mapsto 2], K' \circ \square\,@^{g}\mathtt{I} \circ (\lambda x.\mathsf{true})\square, 1@^{f}\mathsf{B} \rangle & (3.3\text{q}) \\
 & \quad \textbf{\textit{Violation!}}\ \mathit{blame}_{L}(f, \{e\}, 1) = \{e, f\}, 1 & \\
\longrightarrow & \langle \{e, f\}, [a \mapsto 1; b \mapsto 2], K' \circ \square\,@^{g}\mathtt{I} \circ (\lambda x.\mathsf{true})\square, 1 \rangle & (3.3\text{r}) \\
\longrightarrow & \langle \{e, f\}, [a \mapsto 1; b \mapsto 2], K' \circ \square\,@^{g}\mathtt{I}, \mathsf{true} \rangle & (3.3\text{s}) \\
\longrightarrow & \langle \{e, f\}, [a \mapsto 1; b \mapsto 2], K', \mathsf{true}@^{g}\mathtt{I} \rangle & (3.3\text{t}) \\
 & \quad \textbf{\textit{Violation!}}\ \mathit{blame}_{L}(g, \{e, f\}, \mathsf{true}) = \{e, f\}, \mathsf{true} & \\
\longrightarrow & \langle \{e, f\}, [a \mapsto 1; b \mapsto 2], K', \mathsf{true} \rangle & (3.3\text{u}) \\
\longrightarrow & \langle \{e, f\}, [a \mapsto 1; b \mapsto 2], \mathit{Id} \circ \square\,@^{c}\mathsf{any}, (\mathsf{true}, \mathsf{true}) \rangle & (3.3\text{v}) \\
\longrightarrow & \langle \{e, f\}, [a \mapsto 1; b \mapsto 2], \mathit{Id}, (\mathsf{true}, \mathsf{true})@^{c}\mathsf{any} \rangle & (3.3\text{w}) \\
\longrightarrow & \langle \{e, f\}, [a \mapsto 1; b \mapsto 2], \mathit{Id}, (\mathsf{true}, \mathsf{true}) \rangle & (3.3\text{x})
\end{array}
$$

Figure 3.9: Presumptuous Functions: Part Two (with Logging Semantics)

function. Reduction (3.3r) evaluates the boolean contract which is applied to the integer 1, triggering a violation and raising blame on node f, or $+\ell[\mathsf{dom}_0/\mathsf{dom}_1/\mathsf{nil}]$.

Violation! $blame_L(f, \{e\}, 1) = \{e, f\}, 1$ as
$assign(f, \{e\}) = resolve(f, \{e, f\})$ as $\nexists q \in \{e\}.\ compat(-f, q)$
$resolve(f, \{e, f\}) = \{e, f\}, \top$
where $f = +\ell[\mathsf{dom}_0/\mathsf{dom}_1/\mathsf{nil}]$
$e = -\ell[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]$

We observe that the context has already violated the contract annotated with blame node e, however blame should still be assigned to f, the subject. Specifically, while e and f share the prefix dom_0 the two paths diverge at different application contexts: cod_0 and dom_1 respectively. The intuition is that the body of the function must *always* assume that the argument is conforming. A violation from a distinct and prior application of the argument function cannot be used as evidence to escape blame. If the context were to supply a conforming argument such as $\lambda x.42$, or the tuple elements were reversed, then there would be no blame assigned to e at the point f is blamed. In the current reduction there happens to be an unrelated violation from a different application, but this should not absolve the subject of blame. In general, the negative blame assigned to e cannot be relied upon by the subject of the contract.

Reduction (3.3s) to reduction (3.3u) proceed as before, with the only difference being that the continuation K' holds first element of the tuple. We omit discussion of the remaining reductions as they do not provide any additional insight regarding blame assignment.

Getting Away with It Due to the limitations of contract monitoring there are some cases where a presumptuous function can escape blame, or "get away with it". This is because contract monitoring is unable to prove conformance of a function in *all* cases; contract monitoring can only prove that a function does *not* conform. For example:

$$((\lambda f.\lambda x.f\,4)@^p(\mathsf{B} \to \mathsf{B}) \to (\mathsf{B} \to \mathsf{B}))\,(\lambda y.y)\,4$$

Evaluation of the term will assign blame to the context for applying the returned function to the integer 4, however, blame will not be assigned to the subject of the contract for applying f to the integer 4. This is undesirable because the function defined by the subject is presumptuous and relies on the violation from the context to escape blame. If the context were to supply `true` instead of 4 then the subject *would* be assigned blame.

Detecting this case is not possible in our system because we are unable to distinguish an integer value supplied by the context, and an integer value that originated in the body. For example:

$$((\lambda f.\lambda x.f\,x)@^{p}(\mathsf{B}\to\mathsf{B})\to(\mathsf{B}\to\mathsf{B}))\,(\lambda y.y)\,4$$

The function implemented here conforms to the function contract and assigning blame to the subject would be wrong. During evaluation we end up in the same situation as the previous example after substituting x with the integer 4. There is no way to distinguish 4 passed as argument, and 4 originating from inside the definition of the function.

A potential amelioration is to employ taint tracking. When the integer 4 provided by the context violates the first boolean contract the integer is tainted, distinguishing it from the integer inside the function definition. Another approach is to use a parametric contract $\forall X.(X\to X)\to(X\to X)$. The integer 4 provided by the context is sealed upon substitution for variable x, and unsealed upon substitution for variable y. The integer 4 in the body of the function is never sealed and a violation will be raised at any attempt to unseal the integer. This is dual to taint tracking. Instead of tainting the argument provided by the context, we vet the argument using a seal. However, this is not a general solution because the function may not be polymorphic in nature. A further solution would be to employ *where-provenance* (Buneman et al., 2001) to determine the location in the source program from where the integer originated.

Failing to detect certain presumptuous functions does not negatively impact our contract semantics. For example:

$$((\lambda f.\lambda x.f\,4)@^{p}(\mathsf{B}\to\mathsf{B})\to(\mathsf{B}\to\mathsf{B}))\,(\lambda y.y)\,4$$

Failing to assign blame to the subject is justified because a logical interpretation of the contract shows that the contract is satisfied. We interpret the function contract $(\mathsf{B}\to\mathsf{B})\to(\mathsf{B}\to\mathsf{B})$ as the proposition $(A\Rightarrow B)\Rightarrow(C\Rightarrow D)$, and interpret any contract violation as the falsification of the corresponding proposition. When proposition C is false the overall proposition is satisfied:

$$((A\Rightarrow B)\Rightarrow(\bot\Rightarrow D))\Leftrightarrow\top$$

Reinterpreting this in the setting of contracts: the contract $(\mathsf{B}_1\to\mathsf{B}_2)\to(\mathsf{B}_3\to\mathsf{B}_4)$ is satisfied when contract B_3 is violated, which is precisely the violation that occurs in our example program $((\lambda f.\lambda x.f\,4)@^{p}(\mathsf{B}\to\mathsf{B})\to(\mathsf{B}\to\mathsf{B}))\,(\lambda y.y)\,4$.

Furthermore, as we show in our contract semantics defined in Chapter 4, we do not consider this function to satisfy the contract *in general*. For the function to satisfy the contract in general the function must conform to the contract in *all* possible contexts. We can construct the following context for the function that will expose the lack of conformance, rendering the function unsatisfactory.

$$((\lambda f.\lambda x.f\ 4)@^{p}(\mathsf{B}\to\mathsf{B})\to(\mathsf{B}\to\mathsf{B}))\,(\lambda y.y)\,\mathtt{true}$$

3.3.2 Resolution

Blame that is assigned must be interpreted in the context of the current blame state, or *resolved*. There are two broad phases of resolution: root resolution where blame is assigned to a root node that annotated a source contract, and branch resolution where blame is assigned to a branch node and the parent intersection or union must interpret that blame. The definition of blame resolution is defined in Figure 3.4, with auxiliary definitions defined in Figure 3.5. For convenience we repeat the relevant cases inline. We discuss each phase of resolution in turn.

Root Resolution Resolving blame for a root node $\pm\ell[P]$ in state Φ requires no additional work and we indicate that a source contract has been violated by returning the boolean value $\top$. The blame state Φ is returned unmodified.

$$\mathit{resolve}(\pm\ell[P],\Phi) = \Phi,\top$$

Recall that blame assignment is responsible for adding $\pm\ell[P]$ to the state and at the point of resolution there is no further information to add.

Branch Resolution Resolving blame for a branch node $p \bullet d^{\pm}_{\diamond}[P]$ has four configurations: positive or negative, intersection or union. We discuss each starting with the simple cases for positive intersection and negative union.

Positive Intersection Resolving blame for a positive intersection branch $p \bullet d^{+}_{\cap}[P]$ immediately assigns blame to the parent node p.

$$\mathit{resolve}(p \bullet d^{+}_{\cap}[P],\Phi) = \mathit{assign}(\mathit{parent}(p \bullet d^{+}_{\cap}[P]),\Phi)$$

Recall that an intersection contract is assigned positive blame if *either* branch is assigned positive blame, and as a consequence there are no additional constraints required to assign blame to the parent. When propagating blame to the parent node

the operation *parent* is used to extract the parent from a branch. The purpose of the operation is to share information across nested intersection and union contracts; we defer detailed discussion of the operation until the end of this section.

Negative Union Resolving blame for a negative union branch $p \bullet d^{-}_{\cup}[P]$ immediately assigns blame to the parent node p.

$$\mathit{resolve}(p \bullet d^{-}_{\cup}[P], \Phi) = \mathit{assign}(\mathit{parent}(p \bullet d^{-}_{\cup}[P]), \Phi)$$

Recall that a union contract is assigned negative blame if *either* branch is assigned negative blame, and as a consequence there are no additional constraints required to assign blame to the parent.

Negative Intersection Resolving blame for a negative intersection branch $p \bullet d^{-}_{\cap}[P]$ depends upon the current blame state Φ.

$$\mathit{resolve}(p \bullet d^{-}_{\cap}[P], \Phi) = \begin{cases} \mathit{assign}(\mathit{parent}(p \bullet d^{-}_{\cap}[P]), \Phi) & \text{if } \exists P'.\ p \bullet \mathit{flip}(d)^{-}_{\cap}[P'] \in \Phi \\ & \quad \text{and } \mathit{elim}(P, P') \\ \Phi, \bot & \text{otherwise} \end{cases}$$

Recall that an intersection contract is assigned negative blame if *both* branches are assigned negative blame in the *same* elimination context. Consequently, the resolution process must examine the current blame state to determine if both branches have been assigned negative blame, and only then is blame assigned to the parent of the branch. The condition under which blame is assigned to the parent is concretely defined as:

$$\exists P'.\ p \bullet \mathit{flip}(d)^{-}_{\cap}[P'] \in \Phi \text{ and } \mathit{elim}(P, P')$$

The first half of the conjunction asserts that the other branch of the intersection has already been assigned negative blame with any path P', using operation *flip* to reverse the direction of a branch. This condition alone is insufficient because it does not assert that violations occur in the *same* elimination context. Revisiting the example from Section 2.4:

$$\begin{array}{l} \mathsf{let}\ f = (\lambda x.x)@^{p}(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B})\ \mathsf{in} \\ \mathsf{if}\ f\ \mathtt{true}\ \mathsf{then}\ f\ 1\ \mathsf{else}\ f\ 0 \end{array}$$

Evaluation of the condition will assign blame to the node $p \bullet \mathsf{left}^{-}_{\cap}[\mathsf{dom}_0/\mathsf{nil}]$: the domain contract (`I`) in the left branch of the intersection during the *first* application.

Evaluation of the "then" clause will assign blame to the node $p \bullet \mathsf{right}^-_\cap[\mathsf{dom}_1/\mathsf{nil}]$: the domain contract ($\mathtt{B}$) in the right branch of the intersection during the *second* application. At the point of the second violation the first conjunct will be satisfied as there exists a path ($\mathsf{dom}_0/\mathsf{nil}$) along which the inverse branch has been blamed. To ensure that no blame is assigned in this instance the second half of the conjunction asserts that the path must share the same elimination context, denoted $\mathit{elim}(P, P')$. The relation $\mathit{elim}(P, P')$ is defined to relate two paths if the wrap indices at the head of each path match. The blame contexts at the head of each path are not required to match; for instance the paths $\mathsf{dom}_0/\mathsf{nil}$ and $\mathsf{cod}_0/\mathsf{dom}_1/\mathsf{nil}$ are related. In this example the second conjunct is not satisfied because the two paths $\mathsf{dom}_0/\mathsf{nil}$ and $\mathsf{dom}_1/\mathsf{nil}$ are not related; the paths do not stem from the same elimination context, indicated by the differing wrap indices 0 and 1.

Example ▷ Negative Intersection We revisit another example from Section 2.4 that *should* be assigned blame:

$$((\lambda x.x)@^p(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B}))\,\text{"foo"}$$

Figure 3.10 presents the full reduction sequence. In the full sequence we replace the abstract blame node variable p with the concrete root node $+\ell[\mathsf{nil}]$. Furthermore, in the remaining examples we use *blame* instead of blame_L.

Reduction (3.4a) creates an argument frame and begins evaluating the intersection contract. Reduction (3.4b) decomposes the intersection contract, creating branch nodes a and b. Reduction (3.4c) creates a function continuation and begins evaluating the argument. Reduction (3.4d) wraps the application of the outer function contract $\mathtt{B} \to \mathtt{B}$. Reduction (3.4e) creates a contract continuation for the codomain contract $\mathtt{B}$. Reduction (3.4f) creates an argument continuation and begins evaluating the function wrapped in a single contract $\mathtt{I} \to \mathtt{I}$. Reduction (3.4g) creates a function continuation and begins evaluating the argument wrapped in the domain contract $\mathtt{B}$. Reduction (3.4h) evaluates the boolean contract applied to the string "foo", triggering a violation.

Violation! $\mathit{blame}(c, \emptyset, \text{"foo"}) = \{c\}, \text{"foo"}$ as
$\quad \mathit{assign}(c, \emptyset) = \mathit{resolve}(c, \{c\})$ as $\nexists q \in \emptyset.\ \mathit{compat}(-c, q)$
$\quad \mathit{resolve}(c, \{c\}) = \{c\}, \bot$ as $\nexists P'. -\ell[\mathsf{nil}] \bullet \mathsf{left}^-_\cap[P'] \in \{c\}$
$\quad$ where $c = -\ell[\mathsf{nil}] \bullet \mathsf{right}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]$

$$\langle \emptyset, [\,], Id, ((\lambda x.x)@^{+\ell[\mathsf{nil}]}(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B}))\text{“foo”}\rangle$$

$$\longrightarrow \quad \langle \emptyset, [\,], Id \circ \Box\text{“foo”}, (\lambda x.x)@^{+\ell[\mathsf{nil}]}(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B})\rangle \tag{3.4a}$$

$$\textit{let } a = +\ell[\mathsf{nil}] \bullet \mathsf{left}^{+}_{\cap}[\mathsf{nil}]; \quad b = +\ell[\mathsf{nil}] \bullet \mathsf{right}^{+}_{\cap}[\mathsf{nil}]$$

$$\longrightarrow \quad \langle \emptyset, [\,], Id \circ \Box\text{“foo”}, ((\lambda x.x)@^{a}\mathtt{I} \to \mathtt{I})@^{b}\mathtt{B} \to \mathtt{B}\rangle \tag{3.4b}$$

$$\longrightarrow \quad \langle \emptyset, [\,], Id \circ (((\lambda x.x)@^{a}\mathtt{I} \to \mathtt{I})@^{b}\mathtt{B} \to \mathtt{B})\,\Box, \text{“foo”}\rangle \tag{3.4c}$$

$$\textit{let } c = -\ell[\mathsf{nil}] \bullet \mathsf{right}^{-}_{\cap}[\mathsf{dom}_0/\mathsf{nil}]; \quad d = +\ell[\mathsf{nil}] \bullet \mathsf{right}^{+}_{\cap}[\mathsf{cod}_0/\mathsf{nil}]$$

$$\longrightarrow \quad \langle \emptyset, [b \mapsto 1], Id, (((\lambda x.x)@^{a}\mathtt{I} \to \mathtt{I})(\text{“foo”}@^{c}\mathtt{B}))@^{d}\mathtt{B}\rangle \tag{3.4d}$$

$$\longrightarrow \quad \langle \emptyset, [b \mapsto 1], Id \circ \Box\,@^{d}\mathtt{B}, ((\lambda x.x)@^{a}\mathtt{I} \to \mathtt{I})(\text{“foo”}@^{c}\mathtt{B})\rangle \tag{3.4e}$$

$$\longrightarrow \quad \langle \emptyset, [b \mapsto 1], Id \circ \Box\,@^{d}\mathtt{B} \circ \Box(\text{“foo”}@^{c}\mathtt{B}), (\lambda x.x)@^{a}\mathtt{I} \to \mathtt{I}\rangle \tag{3.4f}$$

$$\longrightarrow \quad \langle \emptyset, [b \mapsto 1], Id \circ \Box\,@^{d}\mathtt{B} \circ (\lambda x.x)@^{a}\mathtt{I} \to \mathtt{I}\,\Box, \text{“foo”}@^{c}\mathtt{B}\rangle \tag{3.4g}$$

Violation! $blame(c, \emptyset, \text{“foo”}) = \{c\}, \text{“foo”}$

$$\longrightarrow \quad \langle \{c\}, [b \mapsto 1], Id \circ \Box\,@^{d}\mathtt{B} \circ (\lambda x.x)@^{a}\mathtt{I} \to \mathtt{I}\,\Box, \text{“foo”}\rangle \tag{3.4h}$$

$$\textit{let } e = -\ell[\mathsf{nil}] \bullet \mathsf{left}^{-}_{\cap}[\mathsf{dom}_0/\mathsf{nil}]; \quad f = +\ell[\mathsf{nil}] \bullet \mathsf{left}^{+}_{\cap}[\mathsf{cod}_0/\mathsf{nil}]$$

$$\longrightarrow \quad \langle \{c\}, [b \mapsto 1; a \mapsto 1], Id \circ \Box\,@^{d}\mathtt{B}, ((\lambda x.x)(\text{“foo”}@^{e}\mathtt{I}))@^{f}\mathtt{I}\rangle \tag{3.4i}$$

$$\longrightarrow \quad \langle \{c\}, [b \mapsto 1; a \mapsto 1], Id \circ \Box\,@^{d}\mathtt{B} \circ \Box\,@^{f}\mathtt{I}, (\lambda x.x)(\text{“foo”}@^{e}\mathtt{I})\rangle \tag{3.4j}$$

$$\longrightarrow \quad \langle \{c\}, [b \mapsto 1; a \mapsto 1], Id \circ \Box\,@^{d}\mathtt{B} \circ \Box\,@^{f}\mathtt{I} \circ \Box\,\text{“foo”}@^{e}\mathtt{I}, \lambda x.x\rangle \tag{3.4k}$$

$$\textit{let } K = Id \circ \Box\,@^{d}\mathtt{B} \circ \Box\,@^{f}\mathtt{I} \circ (\lambda x.x)\,\Box$$

$$\longrightarrow \quad \langle \{c\}, [b \mapsto 1; a \mapsto 1], K, \text{“foo”}@^{e}\mathtt{I}\rangle \tag{3.4l}$$

Violation! $blame(e, \{c\}, \text{“foo”}) = \{c, e, -\ell[\mathsf{dom}_0/\mathsf{nil}]\}, \mathtt{blame}\ -\ell$

$$\longrightarrow \quad \langle \{c, e, -\ell[\mathsf{dom}_0/\mathsf{nil}]\}, [b \mapsto 1; a \mapsto 1], K, \mathtt{blame}\ -\ell\rangle \tag{3.4m}$$

$$\longrightarrow \quad \langle \{c, e, -\ell[\mathsf{dom}_0/\mathsf{nil}]\}, [b \mapsto 1; a \mapsto 1], Id, \mathtt{blame}\ -\ell\rangle \tag{3.4n}$$

Figure 3.10: Negative Intersection Blame (with Exception Semantics)

Blame is immediately assigned to node c, or $-\ell[\mathsf{nil}] \bullet \mathsf{right}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]$, as the blame state is empty. When considering how blame is resolved for c we observe that no left branch has been assigned negative blame, and therefore we cannot assign blame to the intersection. Blame resolution returns $\bot$ and no blame error is created. The contract violation is not without effect; the violation has been recorded as evidenced by the addition of c to the resulting blame state. Reduction (3.4i) wraps the application of the remaining function contract $\mathtt{I} \to \mathtt{I}$. Reduction (3.4j) creates a contract continuation for the codomain contract `I`. Reduction (3.4k) creates an argument continuation and begins evaluating the identity function. Reduction (3.4l) creates a function continuation and begins evaluating the argument wrapped in the domain contract `I`. Reduction (3.4m) evaluates the integer contract applied to the string "foo", triggering a violation.

Violation! $\mathit{blame}(e, \{c\}, \text{“foo”}) = \{c, e, -\ell[\mathsf{dom}_0/\mathsf{nil}]\}, \mathtt{blame}\ -\ell$ as
$\quad \mathit{assign}(e, \{c\}) = \mathit{resolve}(e, \{c, e\})$ as $\nexists q \in \{c\}.\ \mathit{compat}(-e, q)$
$\quad \mathit{resolve}(e, \{c, e\}) = \mathit{assign}(-\ell[\mathsf{dom}_0/\mathsf{nil}], \{c\})$ as $-\ell[\mathsf{nil}] \bullet \mathsf{right}^-_\cap[\mathsf{dom}_0/\mathsf{nil}] \in \{c, e\}$
$\quad \mathit{assign}(-\ell[\mathsf{dom}_0/\mathsf{nil}], \{c, e\}) = \mathit{resolve}(-\ell[\mathsf{dom}_0/\mathsf{nil}], \{c, e, -\ell[\mathsf{dom}_0/\mathsf{nil}]\})$
$\quad \mathit{resolve}(-\ell[\mathsf{dom}_0/\mathsf{nil}], \{c, e, -\ell[\mathsf{dom}_0/\mathsf{nil}]\}) = \{c, e, -\ell[\mathsf{dom}_0/\mathsf{nil}]\}, \top$
$\quad$ where $c = -\ell[\mathsf{nil}] \bullet \mathsf{right}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]$
$\qquad\quad e = -\ell[\mathsf{nil}] \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]$

Blame is immediately assigned to node e, or $-\ell[\mathsf{nil}] \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]$, as the blame state has no positive blame nodes to match the negation of e. When considering how blame is resolved for e we observe that a right branch has already been negatively blamed, specifically node c. Furthermore, the paths of both e and c match. This combined information indicates that the intersection should be blamed, and consequently blame is assigned to the parent of e. When blaming the parent we hoist the blame path up to the parent, the details of which we explain at the end of the section. Blame is assigned to the root node $-\ell[\mathsf{dom}_0/\mathsf{nil}]$ because there are no other root nodes in the blame state and resolution indicates that a source contract has been violated. The result of the blame operation is the new blame state $\{c, e, -\ell[\mathsf{dom}_0/\mathsf{nil}]\}$, and a blame error $\mathtt{blame}\ -\ell$. Reduction (3.4n) discards the remaining continuation, causing the blame error to propagate to the top.

$$
\begin{aligned}
& \langle \emptyset, [\,], \mathit{Id}, \text{“foo”}@^{+\ell[\mathsf{nil}]}\mathtt{I} \cup \mathtt{B} \rangle \\
& \quad \mathit{let}\ a = +\ell[\mathsf{nil}] \bullet \mathsf{left}^{+}_{\cup}[\mathsf{nil}]; \quad b = +\ell[\mathsf{nil}] \bullet \mathsf{right}^{+}_{\cup}[\mathsf{nil}] \\
\longrightarrow\ & \langle \emptyset, [\,], \mathit{Id}, (\text{“foo”}@^{a}\mathtt{I})@^{b}\mathtt{B} \rangle && \text{(3.5a)} \\
\longrightarrow\ & \langle \emptyset, [\,], \mathit{Id} \circ \Box\,@^{b}\mathtt{B}, \text{“foo”}@^{a}\mathtt{I} \rangle && \text{(3.5b)} \\
& \quad \textbf{\textit{Violation!}}\ \mathit{blame}(a, \emptyset, \text{“foo”}) = \{a\}, \text{“foo”} \\
\longrightarrow\ & \langle \{a\}, [\,], \mathit{Id} \circ \Box\,@^{b}\mathtt{B}, \text{“foo”} \rangle && \text{(3.5c)} \\
\longrightarrow\ & \langle \{a\}, [\,], \mathit{Id}, \text{“foo”}@^{b}\mathtt{B} \rangle && \text{(3.5d)} \\
& \quad \textbf{\textit{Violation!}}\ \mathit{blame}(b, \{a\}, \text{“foo”}) = \{a, b, +\ell[\mathsf{nil}]\}, \mathtt{blame}\ +\ell \\
\longrightarrow\ & \langle \{a, b, +\ell[\mathsf{nil}]\}, [\,], \mathit{Id}, \mathtt{blame}\ +\ell \rangle && \text{(3.5e)}
\end{aligned}
$$

Figure 3.11: Positive Union Blame (with Exception Semantics)

Positive Union Resolving blame for a positive union branch $p \bullet d^{+}_{\cup}[P]$ depends upon the current blame state Φ.

$$
\mathit{resolve}(p \bullet d^{+}_{\cup}[P], \Phi) = \begin{cases} \mathit{assign}(\mathit{parent}(p \bullet d^{+}_{\cup}[P]), \Phi) & \text{if } \exists P'.\ p \bullet \mathit{flip}(d)^{+}_{\cup}[P'] \in \Phi \\ \Phi, \bot & \text{otherwise} \end{cases}
$$

Recall that a union contract is assigned positive blame if *both* branches are assigned positive blame. Consequently, the resolution process must examine the current blame state to determine if both branches have been assigned positive blame, and only then is blame assigned to the parent of the branch. The condition under which blame is assigned to the parent is concretely defined as:

$$
\exists P'.\ p \bullet \mathit{flip}(d)^{+}_{\cup}[P'] \in \Phi
$$

The condition asserts that the other branch of the union has already been assigned positive blame with any path P', using operation *flip* to reverse the direction of a branch. Unlike intersection, there is no constraint on the matching path P' because the choice of branch is made once at the introduction of the value, not at each elimination. Blame is aggregated across all uses of the contract, or across all paths. Figure 3.11 presents the full reduction sequence of a union contract that is assigned positive blame.

Reduction (3.5a) decomposes the union contract, creating branch nodes a and b. Reduction (3.5b) creates a contract continuation for the right branch contract `B`. Re-

duction (3.5c) evaluates the integer contract applied to the string "foo", triggering a violation.

Violation! $\mathit{blame}(a, \emptyset, \text{“foo”}) = \{a\}, \text{“foo”}$ as
$\mathit{assign}(a, \emptyset) = \mathit{resolve}(a, \{a\})$ as $\nexists q \in \emptyset.\ \mathit{compat}(-a, q)$
$\mathit{resolve}(a, \{a\}) = \{a\}, \bot$ as $\nexists P'. + \ell[\mathsf{nil}] \bullet \mathsf{right}^+_\cup[P'] \in \{a\}$
where $a = +\ell[\mathsf{nil}] \bullet \mathsf{left}^+_\cup[\mathsf{nil}]$

Blame is immediately assigned to node a, or $+\ell[\mathsf{nil}] \bullet \mathsf{left}^+_\cup[\mathsf{nil}]$, as the blame state is empty. When considering how blame is resolved for a we observe that no right branch has been assigned positive blame, and therefore we cannot assign blame to the union. Blame resolution returns $\bot$ and no blame error is created. The contract violation is not without effect; the violation has been recorded as evidenced by the addition of a to the resulting blame state. Reduction (3.5d) pops the contract continuation from the frame stack. Reduction (3.5e) evaluates the boolean contract applied to the string "foo", triggering a violation.

Violation! $\mathit{blame}(b, \{a\}, \text{“foo”}) = \{a, b, +\ell[\mathsf{nil}]\}, \mathtt{blame}\ +\ell$ as
$\mathit{assign}(b, \{a\}) = \mathit{resolve}(b, \{a, b\})$ as $\nexists q \in \{a\}.\ \mathit{compat}(-b, q)$
$\mathit{resolve}(b, \{a, b\}) = \mathit{assign}(+\ell[\mathsf{nil}], \{a, b\})$ as $\exists P'. + \ell[\mathsf{nil}] \bullet \mathsf{left}^+_\cup[P'] \in \{a, b\}$
$\mathit{assign}(+\ell[\mathsf{nil}], \{a, b\}) = \mathit{resolve}(+\ell[\mathsf{nil}], \{a, b, +\ell[\mathsf{nil}]\})$
$\mathit{resolve}(+\ell[\mathsf{nil}], \{a, b, +\ell[\mathsf{nil}]\}) = \{a, b, +\ell[\mathsf{nil}]\}, \top$
where $a = +\ell[\mathsf{nil}] \bullet \mathsf{left}^+_\cup[\mathsf{nil}]$
$b = +\ell[\mathsf{nil}] \bullet \mathsf{right}^+_\cup[\mathsf{nil}]$

Blame is immediately assigned to node b, or $+\ell[\mathsf{nil}] \bullet \mathsf{right}^+_\cup[\mathsf{nil}]$, as the blame state has no negative blame nodes to match the negation of b. When considering how blame is resolved for b we observe that a left union branch has already been positively blamed, specifically node a. This indicates that the union should be blamed, and consequently blame is assigned to the parent of b. Blame is assigned to the root node $+\ell[\mathsf{nil}]$ because there are no other root nodes in the blame state and resolution indicates that a source contract has been violated. The result of the blame operation is the new blame state $\{a, b, +\ell[\mathsf{nil}]\}$, and a blame error $\mathtt{blame}\ +\ell$.

Blame Path Hoisting When blame propagates from branch node to parent node the operation $\mathit{parent}(p)$, defined in Figure 3.5, is used to select the parent node from p. The operation is a partial function on blame nodes, omitting the case for root nodes,

though our system only applies the operation to branch nodes. The motivation for the operation begins with a nested intersection contract.

$$((\mathsf{B} \to \mathsf{B}) \cap (\mathtt{I} \to \mathtt{I})) \cap (\mathsf{S} \to \mathsf{S})$$

Semantically all three function contracts exist at the same "level"; all three function contracts are eliminated at the same application context. When concretely monitoring these contracts the context information—*the blame paths*—will only be added to the deepest branch nodes. The consequence of this is that intermediate intersection or union contracts can miss out on information if blame is not correctly propagated. Consider the following example:

$$\langle \emptyset, [\,], Id, (V @^{+\ell[\mathsf{nil}]}(((\mathsf{B} \to \mathsf{B}) \cap (\mathtt{I} \to \mathtt{I})) \cap (\mathtt{I} \to \mathsf{any}))) \text{“foo”} \rangle$$

The domain contract for each function will be violated with the following blame nodes, reading left-to-right.

(a)	$-\ell[\mathsf{nil}] \bullet \mathsf{left}^-_\cap[\mathsf{nil}] \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]$	for contract $\mathsf{B} \to \mathsf{B}$
(b)	$-\ell[\mathsf{nil}] \bullet \mathsf{left}^-_\cap[\mathsf{nil}] \bullet \mathsf{right}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]$	for contract $\mathtt{I} \to \mathtt{I}$
(c)	$-\ell[\mathsf{nil}] \bullet \mathsf{right}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]$	for contract $\mathtt{I} \to \mathsf{any}$

Negative blame will be assigned to the intersection $(\mathsf{B} \to \mathsf{B}) \cap (\mathtt{I} \to \mathtt{I})$ contract in the left branch of the outermost intersection as blame is assigned to branch nodes *a* and *b*. If we were to naively blame the parent of these branches directly then blame would be assigned to the parent of *a* and *b*, node $-\ell[\mathsf{nil}] \bullet \mathsf{left}^-_\cap[\mathsf{nil}]$. However, the path of this parent node is empty and will never match the path for blame node *c*. The result is that negative blame will *never* be assigned to the top level intersection contract. By naively blaming the parent we lose valuable path information that has been collected by the children.

The solution we present is to *hoist* the blame path, but only under the condition that the parent path is empty. A parent node with an empty path must denote a sequence of nested intersection or union contracts. If the parent node had a non-empty blame path then there must be some intermediate function contracts between the current intersection or union contract, and the parent intersection or union contract. An alternate phrasing is to say that we compare the elimination contexts of a branch blame node and its parent. If the path of the parent is empty then there were no intermediate evaluation contexts between parent and child: they were eliminated in the same context. Consequently, blame path information from the child should be shared with the parent.

The special form of a parent with an empty path always denotes a blame node where that parent and the child are from the same context, consequently it is never the case that the polarity of the parent and child differ.

Proposition 3.3.6. *If M is a source program then $\langle \emptyset, [\,], Id, M\rangle \rightarrowtail^* \langle \Phi, \Delta, K, N\rangle$ where $p \bullet d^{\pm}_{\diamond}[\mathsf{nil}] \bullet d'^{\mp}_{\diamond'}[P] \in \langle \Phi, \Delta, K, N\rangle$.*

Furthermore, given that a blame node with a parent that has an empty path denotes a contract nested within an intersection or union, and all nested contracts share the same evaluation context, we expect blame node compatibility to be preserved by blame path hoisting.

Proposition 3.3.7. *If $compat(p, q)$ then $compat(parent(p), parent(q))$*
when $p = p_1 \bullet d^{\pm}_{\diamond}[\mathsf{nil}] \bullet d'^{\pm}_{\diamond'}[P]$ and $q = p_1 \bullet d^{\pm}_{\diamond}[\mathsf{nil}] \bullet d'^{\pm}_{\diamond'}[P']$

In the example, when blame propagates from the inner intersection the path is hoisted and the modified parent $-\ell[\mathsf{nil}] \bullet \mathsf{left}^{-}_{\cap}[\mathsf{dom}_0/\mathsf{nil}]$ is blamed. The modified parent of nodes a and b will now have a matching path with node c. The result is that negative blame will be correctly assigned to the outermost intersection.

The definition of *parent* has the blemish of performing case analysis on the parent path. The is a consequence of our heavily syntactic approach to representing blame nodes using lists. Future work would be to remedy this and find a representation closer to the underlying semantics, where the evaluation context spanning multiple intersection or union contracts is represented in blame tracking.

3.4 Related Work

3.4.1 Gradual Typing with Intersection and Union

The only existing work that presents a design of higher-order intersection and union contracts is that of Keil and Thiemann (2015a), from which we take much inspiration. There are two key differences between their system and ours.

First is our use of the context tracker Δ to record function contract eliminations, or applications. The use of the contract tracker is critical to the implementation of uniform contract monitoring for intersection and union, however the context tracker does incur an overhead. The size of the overhead is proportional to the size of the blame state. Keil and Thiemann (2015a) do not require a context tracker as their monitoring rules are not uniform, and consequently avoid the extra cost. Both systems

maintain a blame state and the overhead is the same in each system; the blame states are implemented as a forest of directed acyclic graphs of similar structure. In the presence of recursive contracts the blame state is unbounded in size (under a naive solution), however neither system implements recursive contracts. Finding efficient representations of the blame state remains an open problem.

The second key difference is that Keil and Thiemann (2015a) implement user-defined contracts, while we do not. In most cases our system can be immediately extended to support user-defined contracts without change, however there is one class of contracts that demand attention. Keil and Thiemann (2015a) identify that user-defined contracts that apply their argument, such as

$$\lambda f.f\,1 > 0$$

will violate the commuting property of intersection and union contracts. Contracts from one branch may flow into contracts from the other branch given a certain ordering of an intersection or union contract. For example, taking flat contract C to be the predicate $\lambda f.f\,1 > 0$, then the contract $(\mathsf{B} \to \mathsf{B}) \cap C$ will evaluate C with an argument wrapped in contract $\mathsf{B} \to \mathsf{B}$, while the contract $C \cap (\mathsf{B} \to \mathsf{B})$ will not. Keil and Thiemann (2015a) solve this problem by dropping the function contract $\mathsf{B} \to \mathsf{B}$ when it flows into the contract C. We expect that their solution can be adapted to our system in a straightforward manner. This technique is similar to the *lax* contract monitoring strategy and is therefore not a *complete monitor* (Dimoulas et al., 2012). Implementing intersection and union contracts with user-defined contracts that form a complete monitor is interesting and challenging future work.

The Racket (Flatt and PLT, 2010) language provides the most extensive implementation of contracts and combinators, however there is no higher-order intersection or union combinator. The closest counterparts are the combinators `and/c` and `or/c`. Both operators treat positive and negative blame in a covariant way. The combinator `and/c` is assigned blame if any of the constituents are assigned positive or negative blame. The combinator `or/c` is assigned blame if all of the constituents are assigned positive or negative blame. Another phrasing is that `and/c` handles both positive and negative obligations using conjunction, while `or/c` handles both positive and negative obligations using disjunction.

The `or/c` combinator also has the additional constraint where only one higher-order type is allowed. For example:

```
(or/c number? (-> string? string? string?))
```

is a legal use of the combinator, while

```
(or/c (-> number? number?) (-> string? string? string?))
```

is not a legal use of the combinator. The implementation of `or/c` depends upon being able to detect a violation using only first-order checks, or having an unambiguous candidate after first-order checks are applied. There is no first-order check capable of distinguishing the contracts `(-> number? number?)` and `(-> string? string? string?)`, therefore the combination is outlawed.

Castagna and Lanvin (2017) are the first to combine gradual types with set-theoretic types including intersection, union, and negation types. The premise of their work is that combining intersection and union types with the unknown (or any) type will provide greater control over the spectrum between static and dynamic. Consider an example that they present:

```
1 function f(condition: B, x: /* ???? */) {
2   if (condition) {
3     return succ(x);
4   } else {
5     return not(x);
6   }
7 }
```

When `condition` is true the successor function is applied to `x`; when `condition` is false the negation function is applied to `x`. Ascribing the `any` type to `x` allows the programmer to write the body of `f` without explicit type casts, but allows a caller to provide any arguments, including those outside the domain of `succ` and `not`. Ascribing the type `I ∪ B` to `x` will statically reject any argument outside the domain of `succ` and `not`, but requires the programmer to write explicit type casts in the body of `f`. Castagna and Lanvin (2017) propose ascribing the type `(I ∪ B) ∩ any` to `x`, yielding the best of both approaches. Static type checking is enforced at the call-site of the function: the argument must be an integer or boolean. Automatic cast insertion is provided in the body of the function: the `any` type annotation omits the need for the programmer to write casts by hand.

The semantics of gradual types are given using abstract interpretation in the style of Garcia et al. (2016), where gradual types are interpreted as sets of static types. Their work does not consider blame or the gradual guarantee (Siek et al., 2015b). A blame

theorem is sacrificed in favour of a simpler, but sound, cast calculus. Multiple function casts are collapsed, and only the outer cast is considered.

Castagna et al. (2019) build on the ideas of Castagna and Lanvin (2017) and seek to combine semantic subtyping, gradual typing, and polymorphism. They are inspired by the work of Garcia et al. (2016) on abstract interpretation but observe that the approach is still heavily syntactic, rather than semantic. Castagna et al. (2019) take a new perspective that interprets the unknown type as a particular form of type variable, where each occurrence of the unknown type is possibly a distinct type variable.

Their idea leads to the notion of *discrimination* which gives a semantics to gradual types by replacing the unknown type with a type variable. Under discrimination, each polymorphic gradual type is mapped to a set of polymorphic static types that are interpreted using semantic subtyping. Castagna et al. (2019) also give new insights about the fundamental relations between gradual types. Their work replaces consistency (Siek and Taha, 2006) and consistent subtyping (Garcia et al., 2016) with subtyping and materialization. Both subtyping and materialization are preorders and therefore transitive, while consistency and consistency subtyping are not.

Castagna et al. (2019) provide a blame theorem for their calculus, but they do not prove the theorem in the style of Wadler and Findler (2009) that use positive and negative subtyping. Instead, the polarity of a blame label is statically typed according to the variance of the cast. Blame soundness therefore follows directly from type soundness. Castagna et al. (2019) also prove the static component of the gradual guarantee (Siek et al., 2015b).

3.4.2 Provenance

The formal study of provenance in programming languages covers many aspects regarding origin, history, and usage. Buneman et al. (2001) develop *where* and *why* provenance. The former distinguishes an unambiguous location from which an object in the output was sourced. There are both similarities and differences with the blame tracking we present and where provenance. We use blame nodes to identify unique first-order contracts in a manner reminiscent of where provenance, with both systems employing a notion of a "path". A key difference is that blame nodes identify contracts, not the data to which the contract is applied; in contrast, where provenance identifies data. This difference gives rise to a limitation in contract monitoring, as we describe in Section 3.3.1 with our example *Getting Away with It*. We can uniquely

identify a contract but we cannot uniquely identify the value that violated the contract, and consequently there are pathological examples where a function can avoid blame. A future extension to contract monitoring might use blame nodes for contracts and labels for data, combining both to detect more violations.

Related variants *why* (Buneman et al., 2001) provenance and *dependency* provenance (Cheney et al., 2007) are concerned with identifying the data that contributed to a particular output. We draw two parallels between why provenance and gradual typing. The first is with blame assignment. Blame assignment is a function of blame state, and given a root node that is assigned blame we might consider the set of nodes that "proved" blame as a form of why provenance. Our formalism does not report this provenance, but in practice this provenance is a useful debugging aid. For example, a union contract is violated when both branches are violated; knowing the root node tells us the location of the union contract, but it does not tell us the location of each branch violation. Ideally, we would like to know *why* the contract was violated, which corresponds the to set of branch nodes that proved blame. The second parallel is with the work of Vitousek et al. (2017). They use a blame map and *collaborative* blame. Given a contract violation the blame map is consulted to learn *why* the contract was violated. The difference with this work and ours is that the blame map is not used to assign blame, only to reconstruct the provenance.

Acar et al. (2012) develop a core calculus for provenance that exploits traced execution and provides generalised provenance extraction; many existing forms of provenance can be seen as instances of the generalised extraction mechanism. Blame nodes provide a limited form of tracing that is specialised to contract elimination, while tracing in the core calculus considers all expression forms. An interesting line of work would be to instantiate contract checking in the core calculus, evoking a direct relationship between blame tracking and tracing, and blame assignment and provenance extraction. A further line of work would be to relate the various contract monitoring strategies presented by Swords et al. (2018) as various forms of provenance extraction. Dimoulas et al. (2011) develop *complete monitors* that use ownership labels to enforce correctness properties of contracts; only programs that satisfy certain ownership constraints can reduce. An interesting question is whether *complete monitoring* can be restated as a property of traces and provenance extraction. For example, given a trace from a contract violation and an appropriate provenance extraction function, there should be a single owner associated with the provenance of the violation.

Chapter 4

Blame-oriented Contract Semantics

In Chapter 3 a concrete operational semantics for contracts was presented, focusing on the detection of contract violations and blame assignment. An alternate view might focus on the dual perspective: the absence of contract violations and blame. What does it mean for a program to satisfy a contract, and how might we expect a satisfying program to behave?

In this chapter we investigate contract semantics, focusing on notions of contract satisfaction and properties of programs that satisfy contracts. Contract monitoring distinguishes two parties: subject and context, and distinguishes blame accordingly: positive and negative. We continue this dichotomy when discussing the semantics of contract satisfaction. Values positively satisfy a contract and continuations, or contexts, negatively satisfy a contract. From a definition of contract satisfaction we derive a series of properties that relate contract types and contract satisfaction for both values and continuations.

We begin by discussing two broad approaches to contract satisfaction and then present a new refinement on contract satisfaction that we refer to as *blame-oriented* contract satisfaction. Next, we relate existing notions of contract soundness to our presented definition of contract satisfaction, and consider new implications that arise from our definition. Equipped with a definition of contract satisfaction we present a series of *sound monitoring properties* that relate contract types and contract satisfaction, in essence, providing a set of correctness checks for a contract implementation. We conclude with a discussion of similar approaches and related work.

4.1 Approaches to Contract Satisfaction

Contract satisfaction is the notion that a program fulfils the invariants of a contract. We following existing work by Dimoulas and Felleisen (2011) and Keil and Thiemann (2015a) that distinguishes positive and negative contract satisfaction, separating the obligations associated with a contract. A value may positively satisfy a contract while a continuation or evaluation context may negatively satisfy a contract. Dimoulas and Felleisen (2011) and Keil and Thiemann (2015a) both use a positive and negative duality in their definition of contract satisfaction, however the fundamental semantics are significantly different. We characterise the two approaches as *monitoring oriented* contract satisfaction and *denotational* contract satisfaction.

4.1.1 Monitoring Oriented Contract Satisfaction

A monitoring oriented definition of contract satisfaction is one in which contracts and contract monitoring play a central role in defining the semantics. In this setting contract satisfaction is defined by the observed behaviour of a program when a contract is applied.

Dimoulas and Felleisen (2011) define contract satisfaction using observational equivalence between programs monitored with contracts. A contract can be split into positive and negative components that only monitor the positive and negative obligations of the contract respectively. For example, define operations A^+ and B^- on simple types as:

Definition 4.1.1 (Positive and Negative Contract Obligations)**.**

$$
\begin{array}{lll}
(A \to B)^+ = A^- \to B^+ & \iota^+ = \iota & \mathsf{any}^+ = \mathsf{any} \\
(A \to B)^- = A^+ \to B^- & \iota^- = \mathsf{any} & \mathsf{any}^- = \mathsf{any}
\end{array}
$$

The operation A^+ returns the positive obligations of type A and the operation B^- returns the negative obligations of type B. Obligations for function types are contravariant in the domain and covariant in the codomain, like subtyping and blame. The positive obligation of a base type ι is the type itself, while the negative obligation of a base type ι is the type any. There are no requirements placed on a base type by the context so any is used instead. There are no obligations associated with the type any therefore the operations any^+ and any^- are the identity.

Dimoulas and Felleisen (2011) define a value as positively satisfying contract A if monitoring the value with contract A is observationally equivalent to monitoring

the value with contract A^- in *all* contexts. Similarly, a context is defined as negatively satisfying contract B if monitoring the context with contract B is observationally equivalent to monitoring the context with contract B^+, for *all* values passed to the context. An alternate phrasing is that a value satisfies contract A when the positive (or subject) obligations of contract A are not observable when monitoring the value, precisely because the value never violates the positive obligations.

A monitoring oriented definition of contract satisfaction such as the one presented by Dimoulas and Felleisen (2011) does not capture any external intuition associated with a particular contract type. The definition does little to constrain which values should satisfy a type, and which values should not. For example, a nonsensical contract implementation may choose to trigger a violation at every use of an integer contract, independent of the value under contract. Clearly such a design contradicts our intuition that only non-integers should violate the integer contract; in general we would like contract violations to indicate meaningful errors. In Section 4.4 we supplement contract satisfaction with a series of monitoring properties to guide the design of contracts, and these properties should be derivable in a system. The monitoring properties act as our frame of reference, articulating our intuition about types that contract satisfaction should entail. Dimoulas and Felleisen (2011) do not derive monitoring properties for types, however a set of properties could be similarly obtained for their system.

4.1.2 Denotational Contract Satisfaction

A denotational definition of contract satisfaction is one in which contracts and contract monitoring do not define the semantics. Contract satisfaction is defined by the structure of a given program, distinct from any notion of contract monitoring.

Keil and Thiemann (2015a) present a novel denotational approach to contract satisfaction that describes the intrinsic properties of satisfying programs. Their semantics defines sets of terms that positively satisfy a contract and sets of contexts that negatively satisfy a contract. Each set is constructed co-inductively using rules that describe the shape of satisfying programs. For example—and as no surprise—the set of terms that positively satisfy the integer contract is the set of terms that reduce to an integer. Keil and Thiemann (2015a) support user-defined contracts therefore the set is precisely defined as the set of terms that do not reduce to *false* when the integer predicate is applied. This denotational set-based semantics pleasingly extends

to intersection and union in the expected way. For example, the set of terms that positively satisfy the contract $A \cap B$ is the intersection of terms that positively satisfy A *and* positively satisfy B. In contrast, the set of terms that positively satisfy the contract $A \cup B$ is the union of terms that positively satisfy A *or* positively satisfy B.

The definition of negative contract satisfaction is more involved, for example, consider the rule for evaluation contexts where the hole is in an argument position.

$$\frac{\forall \mathcal{M}, V.\ \lambda x.M = \lambda x.\mathcal{M}[x] \Rightarrow E[\mathcal{M}\{x := V\}[\Box]] \in [\![C]\!]^-}{E[(\lambda x.M)\,\Box] \in [\![C]\!]^-}$$

While complex, this rule does capture the intuitive behaviour. We do not describe the rule in detail but present a high-level reading. A context that applies $\lambda x.M$ to the hole satisfies C if the context that arises by replacing all-but-one bound occurrences of x in M with V, also satisfies C, where the one remaining occurrence of x is replaced by the hole. Importantly, the rule ranges over all possible values and all possible choices of the remaining hole in the body of the function.

Blume and Mcallester (2006) also present a denotation style model of contract satisfaction that is sound and complete. They define sets of terms that satisfy a contract, but unlike Keil and Thiemann (2015a); they do not have a context counterpart to contract satisfaction.

4.2 Blame-oriented Contract Satisfaction

We present blame-oriented contract satisfaction: defining contract satisfaction using *blame assignment*. We take inspiration from both the monitoring and denotational approaches, defining contract satisfaction in terms of the former, and then deriving properties capturing the essence of the latter. Additionally, we adopt the presentation style of Keil and Thiemann (2015a) that uses sets of values to denote positive satisfaction and sets of continuations to denote negative satisfaction.

First, we start with a high level description of blame-oriented contract satisfaction before revealing the details in the definition. In Section 4.4 we then present a series of sound monitoring properties relating contract types and contract satisfaction.

We write $[\![A]\!]^+$ to denote the set of closed values that positively satisfy the type A, and $[\![B]\!]^-$ to denote the set of closed continuations that negatively satisfy the type B. A value V satisfies type A if applying a contract of that type can never elicit *positive* blame in any context. A continuation K satisfies a type B if applying a contract of

that type can never elicit *negative* blame for any value.

Figure 4.1 presents the definition of contract satisfaction, with auxiliary definitions presented in Figure 4.2. We proceed by discussing a particular instance of contract satisfaction that we call *witness satisfaction*.

Witness Satisfaction If contract monitoring is the process of obtaining counter-examples to the claim that a value or continuation satisfies a contract, then the blame node annotating the contract acts as witness to the counter-example. This idea is captured by witness satisfaction, whereby a value or continuation satisfies a contract observed by a particular witness, or blame node.

We write $[\![A]\!]_p^+$ to denote the set of closed values that positively satisfy the type A for witness p; the definition is given Figure 4.1 and we repeat the definition inline.

$$V \in [\![A]\!]_p^+ \stackrel{\text{def}}{=} \forall \Phi, \Delta, K. \langle \Phi, \Delta, K, V@^p A\rangle \longmapsto^* \langle \lightning\, p\rangle \qquad \text{when } p \# \langle \Phi, \Delta, K, V\rangle$$

A value V positively satisfies type A for witness p if $\langle \Phi, \Delta, K, V@^p A\rangle$ is *safe for* p, written $\langle \Phi, \Delta, K, V@^p A\rangle \longmapsto^* \langle \lightning\, p\rangle$. The statement ranges over all blame states, context trackers, and continuations, as satisfaction should not be conditioned on a particular context. Furthermore, we require that the witness node p is fresh in the configuration, written $p \# \langle \Phi, \Delta, K, V\rangle$. The semantics of blame safety and freshness are defined in Figure 4.1 and Figure 4.2 respectively, and described in the following section.

We write $[\![B]\!]_p^-$ to denote the set of closed continuations that negatively satisfy the type B for witness p; the definition is given Figure 4.1 and we repeat the definition inline.

$$K \in [\![B]\!]_p^- \stackrel{\text{def}}{=} \forall \Phi, \Delta, V. \langle \Phi, \Delta, K, V@^p B\rangle \longmapsto^* \langle \lightning\, -p\rangle \quad \text{when } p \# \langle \Phi, \Delta, K, V\rangle$$

A continuation K negatively satisfies type B for witness p if $\langle \Phi, \Delta, K, V@^p A\rangle$ is *safe for* $-p$, written $\langle \Phi, \Delta, K, V@^p A\rangle \longmapsto^* \langle \lightning\, -p\rangle$. The statement ranges over all blame states, context trackers, and values, and also requires freshness of p.

The choice to be explicit about the witness to the contract may seem pedantic; contract monitoring is typically concerned with the presence or absence of violations, rather than the particular witness involved. The strength of witness satisfaction is that it enables us to relate satisfaction of multiple program components that share a common witness; certain desirable properties only hold for a particular combination of witnesses. In Section 4.3 we show how witness satisfaction allows us to reason about contract soundness for function contracts.

Blame Implication $\boxed{\Phi \models p}$

$$\Phi \models p \stackrel{\text{def}}{=} (\exists P'.\, \textit{replace-path}(p, P') \in \Phi \wedge \textit{prefix}(\textit{path}(p), P')) \text{ and } (\nexists P'.\, \textit{replace-path}(p, P') \in \Phi \wedge \textit{compat}(\textit{path}(p), P'))$$

Blame Safety $\boxed{\langle \Phi, \Delta, K, M\rangle \not\longrightarrow^* \langle \lightning\, p\rangle}$

$$\langle \Phi, \Delta, K, M\rangle \not\longrightarrow^* \langle \lightning\, p\rangle \stackrel{\text{def}}{=} \nexists \Phi', \Delta', K', N.\, \langle \Phi, \Delta, K, M\rangle \longrightarrow^* \langle \Phi', \Delta', K', N\rangle \wedge \Phi' \models p$$

Witness Satisfaction $\boxed{V \in [\![A]\!]^+_p \text{ and } K \in [\![B]\!]^-_p}$

$$V \in [\![A]\!]^+_p \stackrel{\text{def}}{=} \forall \Phi, \Delta, K.\langle \Phi, \Delta, K, V@^p A\rangle \not\longrightarrow^* \langle \lightning\, p\rangle \quad \text{when } p \# \langle \Phi, \Delta, K, V\rangle$$
$$K \in [\![B]\!]^-_p \stackrel{\text{def}}{=} \forall \Phi, \Delta, V.\langle \Phi, \Delta, K, V@^p B\rangle \not\longrightarrow^* \langle \lightning\, -p\rangle \quad \text{when } p \# \langle \Phi, \Delta, K, V\rangle$$

Contract Satisfaction $\boxed{V \in [\![A]\!]^+ \text{ and } K \in [\![B]\!]^-}$

$$V \in [\![A]\!]^+ \stackrel{\text{def}}{=} \forall p.\, V \in [\![A]\!]^+_p$$
$$K \in [\![B]\!]^- \stackrel{\text{def}}{=} \forall p.\, K \in [\![B]\!]^-_p$$

Figure 4.1: Contract Satisfaction (Auxiliary Definitions in Figure 4.2)

Blame Safety and Freshness The predicate on program configurations:

$$\langle \Phi, \Delta, K, M\rangle \not\longrightarrow^* \langle \lightning\, p\rangle$$

is read *safe for p*. Informally, a configuration is safe for blame node p if the configuration never reduces to a configuration that assigns blame to p—where p has *witnessed* a contract violation. Formally, a configuration is safe for p if the configuration never reduces to a configuration $\langle \Phi', \Delta', K', N\rangle$ where Φ' *implicates* p. We write $\langle \Phi, \Delta, K, M\rangle \longrightarrow^* \langle \lightning\, p\rangle$ for the negation of the predicate.

Blame implication, written $\Phi \models p$ and defined in Figure 4.1, states that p has been assigned blame in Φ for some path extension. We write this condition as:

$$\exists P'.\, \textit{replace-path}(p, P') \in \Phi \wedge \textit{prefix}(\textit{path}(p), P')$$

which states that there is some path replacement of blame node p in Φ, such that the existing path of p is a *prefix* of the new path P'. The operations *replace-path*, *prefix*,

Path Prefix $\boxed{\mathit{prefix}(P, P')}$

$$\frac{}{\mathit{prefix}(P, P)} \qquad \frac{\mathit{prefix}(P, P')}{\mathit{prefix}(P, P' \gg c_n)}$$

Path Extraction $\boxed{\mathit{path}(p)}$ **Path Replacement** $\boxed{\mathit{replace\text{-}path}(p, P)}$

$$\begin{array}{lcl} \mathit{path}(\pm\ell[P]) & = & P \\ \mathit{path}(p \bullet d^{\pm}_{\circ}[P]) & = & P \end{array} \qquad \begin{array}{lcl} \mathit{replace\text{-}path}(\pm\ell[P], P') & = & \pm\ell[P'] \\ \mathit{replace\text{-}path}(p \bullet d^{\pm}_{\circ}[P], P') & = & p \bullet d^{\pm}_{\circ}[P'] \end{array}$$

Blame Ordering $\boxed{p \leq q}$

$$\frac{}{p \leq p} \qquad \frac{p \leq q}{p \leq (q \gg c_n)} \qquad \frac{p \leq q}{p \leq q \bullet d^{\pm}_{\circ}[\mathsf{nil}]} \qquad \frac{p \leq q \bullet d^{\pm}_{\circ}[P]}{p \leq q \bullet d^{\pm}_{\circ}[\mathsf{nil}] \bullet d'^{\pm}_{\circ'}[P]}$$

Context Ordering $\boxed{p \leq_{\Delta} q}$

$$\frac{}{p \leq_{\Delta} p} \qquad \frac{p \leq_{\Delta} q \qquad \delta(\Delta, q) = (\Delta', m) \qquad m \leq n}{p \leq_{\Delta} (q \gg c_n)} \qquad \frac{p \leq_{\Delta} q}{p \leq_{\Delta} q \bullet d^{\pm}_{\circ}[\mathsf{nil}]} \qquad \frac{p \leq_{\Delta} q \bullet d^{\pm}_{\circ}[P]}{p \leq_{\Delta} q \bullet d^{\pm}_{\circ}[\mathsf{nil}] \bullet d'^{\pm}_{\circ'}[P]}$$

Freshness $\boxed{p \# \langle \Phi, \Delta, K, V \rangle}$

$$p \# \langle \Phi, \Delta, K, V \rangle \stackrel{\text{def}}{=} \nexists q. \in \{\Delta, K, V\}.\ (p \leq \pm q) \vee (\pm q \leq_{\Delta} p) \text{ and } \nexists q. \in \Phi.\ (p \leq \pm q)$$

Figure 4.2: Auxiliary Definitions for Figure 4.1

and *path* are defined in Figure 4.2. For example, if $\Phi = \{+\ell[\mathsf{cod}_0/\mathsf{nil}]\}$ then $\Phi \models +\ell[\mathsf{nil}]$ by the path replacement of $+\ell[\mathsf{nil}]$ with path $\mathsf{cod}_0/\mathsf{nil}$.

Blame implication tolerates path contraction, that is, replacing the path of a blame node with a prefix. Violating a function sub-contract should also implicate the containing function contract. For example, if the integer contract in the type $\mathsf{B} \longrightarrow \mathtt{I}$ has been violated, then the function contract has also been violated. Blame implication does not include branch extension because assigning blame to a branch node does not imply that the parent will be assigned blame. We include an additional constraint on blame implication stating that there is no node in the blame state that is compatible

with the implicated node. We write this condition as:

$$\nexists P'.\ \mathit{replace\text{-}path}(p, P') \in \Phi \wedge \mathit{compat}(\mathit{path}(p), P')$$

which states that there is no path replacement of blame node p in Φ, such that the existing path of p is compatible with the new path P'. This condition in primarily technical and serves to ensure that the implicated blame node p is unambiguously the first violation associated with a contract.

We write $p \# \langle \Phi, \Delta, K, V \rangle$ to indicate that p is *fresh* in the configuration. Freshness is defined in Figure 4.2 and requires orderings $p \leq q$ and $p \leq_\Delta q$, also defined in Figure 4.2. We refer to the former as blame ordering, and the latter as context ordering. Defining freshness for p only as the absence of p in the configuration is insufficient. A blame node p that was initially absent may become present as blame propagates from child to parent. For example, take the following blame state:

$$\{+\ell[\mathsf{nil}] \bullet \mathsf{right}^+_\cap[\mathsf{nil}]\}$$

The blame node $+\ell[\mathsf{nil}]$ does not directly appear in the blame state, but will be synthesised when blame is assigned to branch $+\ell[\mathsf{nil}] \bullet \mathsf{right}^+_\cap[\mathsf{nil}]$ and propagates to parent $+\ell[\mathsf{nil}]$. We say that $+\ell[\mathsf{nil}]$ *is not* fresh in blame state $\{+\ell[\mathsf{nil}] \bullet \mathsf{right}^+_\cap[\mathsf{nil}]\}$. Furthermore, a blame node p that was initially absent may become present as a contract decomposes. For example, take the following program configuration:

$$\langle \emptyset, [\,], \mathit{Id}, (V @^{+\ell[\mathsf{nil}]} \mathsf{B} \to \mathtt{I})\, 4 \rangle$$

The blame node $+\ell[\mathsf{cod}_0/\mathsf{nil}]$ does not directly appear in the configuration, but will be synthesised when the application is wrapped. We say that $+\ell[\mathsf{cod}_0/\mathsf{nil}]$ *is not* fresh in configuration $\langle \emptyset, [\,], \mathit{Id}, (V @^{+\ell[\mathsf{nil}]} \mathsf{B} \to \mathtt{I})\, 4 \rangle$.

Blame ordering describes the case where nodes are synthesised through propagating blame, and context ordering describes the case where nodes are synthesised through contract decomposition. The definition of each ordering replays the way a blame node may be synthesised. Context ordering is parameterised by a context tracker Δ and uses operation $\delta(\Delta, q)$ defined in Figure 3.3, this provides a greater degree of tolerance. For example, take the following program configuration:

$$\langle \emptyset, [+\ell[\mathsf{nil}] \mapsto 5], \mathit{Id}, (V @^{+\ell[\mathsf{nil}]} \mathsf{B} \to \mathtt{I})\, 4 \rangle$$

The blame node $+\ell[\mathsf{cod}_0/\mathsf{nil}]$ is currently fresh in the configuration and will remain fresh because any wrapping of the function contract will only extend paths using

indices of 5 or greater. When the context tracker parameterising context ordering is the empty map [ ], then blame and context ordering coincide.

Proposition 4.2.1. $p \leq q \Leftrightarrow p \leq_{[]} q$

The last rules for blame ordering and context ordering that inject an empty branch node are motivated by the operational semantics and the notion of blame path hoisting. The orderings are intended to approximate the blame nodes that can be synthesised and assigned blame during evaluation, and consequently we require the following proposition to be valid.

Proposition 4.2.2. *If* $p = \mathit{parent}(q)$ *then* $p \leq q$ *and* $p \leq_{\Delta} q$.

This proposition is not valid without the last rules in blame ordering and context ordering. The unintuitive origin of these rules is a direct consequence of the unintuitive definition of $\mathit{parent}(q)$ required to resolve nested intersection and union contracts. As discussed in Section 3.3.2, we consider it future work to find a representation of blame nodes that is less dependent on syntactic manipulation; a more natural definition of blame node ordering would also follow from this work.

The definition of freshness delivers the expected property that fresh blame nodes are never assigned blame.

Lemma 4.2.3 (Safety by Freshness)**.** *If* $p \# \langle \Phi, \Delta, K, M \rangle$ *then* $\langle \Phi, \Delta, K, M \rangle \not\longrightarrow^* \langle \lightning \pm p \rangle$.

Proof. See Appendix A. □

Contract Satisfaction We may now give a concrete definition of contract satisfaction, having described the foundations. We write $[\![A]\!]^+$ to denote the set of closed values that positively satisfy the type A; the definition is given Figure 4.1 and we repeat the definition inline.

$$V \in [\![A]\!]^+ \stackrel{\text{def}}{=} \forall p.\ V \in [\![A]\!]^+_p$$

A value positively satisfies the type A if the value is in the positive witness satisfaction set $[\![A]\!]^+_p$ for *all* witnesses p.

We write $[\![B]\!]^-$ to denote the set of closed continuations that negatively satisfy the type B; the definition is given Figure 4.1 and we repeat the definition inline.

$$K \in [\![B]\!]^- \stackrel{\text{def}}{=} \forall p.\ K \in [\![B]\!]^-_p$$

A continuation negatively satisfies the type B if the continuation is in the negative witness satisfaction set $[\![B]\!]^-_p$ for *all* witnesses p.

A value or continuation can only satisfy a contract if it satisfies the contract for any witnesses, without relying on leniency or special treatment from a particular witness. Interactions specific to a particular witness can arise from blame nodes created through function wrapping, for example $-p \gg \mathsf{dom}_n$ and $p \gg \mathsf{cod}_n$. If our current witness is $p \gg \mathsf{cod}_n$ then assigning blame to $-p \gg \mathsf{dom}_n$ makes it impossible to ever assign blame to $p \gg \mathsf{cod}_n$, even if $p \gg \mathsf{cod}_n$ annotates a violated contract. If we were to change the witness to $q \gg \mathsf{cod}_n$, for some distinct q, then blame would be assigned to the witness $q \gg \mathsf{cod}_n$ because it is not compatible with $-p \gg \mathsf{dom}_n$. This interaction is not ruled out by freshness; $-p \gg \mathsf{dom}_n$ and $p \gg \mathsf{cod}_n$ are fresh with respect to each-other and can co-occur. Treating these two blame nodes as distinct is necessary because an application of the wrap rule for function contracts should synthesise two fresh nodes.

Freshness is not primarily concerned with preventing blame nodes interacting through compatibility; freshness is primarily concerned with ensuring that we can accurately attribute the source of a violation.

Satisfaction for Terms Positive witness satisfaction and contract satisfaction can be extended to include terms. Definition 4.2.4 defines witness satisfaction for terms.

Definition 4.2.4 (Term Witness Satisfaction)**.**

$$M \in [\![A]\!]^+_p \stackrel{def}{=} M \longrightarrow^* V \text{ and } V \in [\![A]\!]^+_p$$

We write $M \longrightarrow^* V$ as an abbreviation for $\langle \emptyset, [\,], Id, M\rangle \longrightarrow^* \langle \Phi, \Delta, Id, V\rangle$. A term M satisfies type A for witness p if M reduces to value V and V satisfies A for witness p.

Definition 4.2.5 defines contract satisfaction for terms.

Definition 4.2.5 (Term Contract Satisfaction)**.**

$$M \in [\![A]\!]^+ \stackrel{def}{=} \forall p.\ M \in [\![A]\!]^+_p$$

A term M satisfies type A if M satisfies type A for all witness p.

4.3 Contract Soundness

The property of contract soundness is a fundamental tenet of contract systems. The essential characteristic provided by contract soundness is that applying a contract to any program produces a program that satisfies the contract. We crystallise this property for our system in Theorem 4.3.1, adopting a similar presentation to Keil and Thiemann (2015a).

Theorem 4.3.1 (Contract Soundness)**.**

(a) $M@^{\pm\ell[P]}A \in [\![A]\!]^{+}$

(b) $K \circ \square\, @^{\pm\ell[P]}B \in [\![B]\!]^{-}$

Proof. See Appendix B. □

Contract soundness consists of positive and negative components. For any term M, applying a contract of type A annotated with a root blame node $\pm\ell[P]$ yields a term that positively satisfies type A. For any continuation K, applying a contract of type B annotated with a root blame node $\pm\ell[P]$ yields a continuation that negatively satisfies type B. The use of a root blame node to annotate the guarding contract is fundamental to the validity of contract soundness. A root node is required to ensure blame assigned to the inner contract generates a blame error that terminates the program before the outer contract observes ill-behaviour. If the blame node annotating the inner contract is a branch blame node there is no guarantee that blame will resolve to the parent root node, and consequently, the outer contract *may* get evaluated.

To explain the intuition behind contract soundness we expand the definition for the positive case.

$$M@^{\pm\ell[P]}A \in [\![A]\!]^{+} \equiv (M@^{\pm\ell[P]}A)@^{p}A \longrightarrow\!\!\!\!\!\!\rightarrow^{*} \langle \lightning\, p \rangle$$

Satisfaction of type A is equivalent to guaranteeing that the outer contract is never assigned positive blame. The guarantee is fulfilled in two ways. If term M already satisfies type A then both contracts act as the identity. If term M does not satisfy type A then any illicit behaviour will first assign blame to the inner contract, generating a blame error and terminating the program before the outer contract can be violated. The case for negative satisfaction is interpreted in a similar manner, except it is the outer contract that prevents the inner contract from detecting a negative contract violation.

Contract soundness is useful for reasoning about function contracts. For example:

$$(\lambda x.x)@^{\pm\ell[\mathsf{nil}]}(\mathtt{I} \to \mathtt{I}) \to (\mathtt{I} \to \mathtt{I})$$

The above function contract will never be assigned positive blame because any argument will always be wrapped in a guarding contract for the type $\mathtt{I} \to \mathtt{I}$. Specifically, after β-reduction we have a value of the form:

$$(W@^{-\ell[\mathsf{dom}_n/\mathsf{nil}]}\mathtt{I} \to \mathtt{I})@^{+\ell[\mathsf{cod}_n/\mathsf{nil}]}\mathtt{I} \to \mathtt{I}$$

for some argument W. Using contract satisfaction we derive two observations:

- $W@^{-\ell[\mathsf{dom}_n/\mathsf{nil}]}\mathtt{I} \to \mathtt{I} \in [\![\mathtt{I} \to \mathtt{I}]\!]^{+}$
- $Id \circ \square\, @^{+\ell[\mathsf{cod}_n/\mathsf{nil}]}\mathtt{I} \to \mathtt{I} \in [\![\mathtt{I} \to \mathtt{I}]\!]^{-}$

The statements tell us that blame nodes $+\ell[\mathsf{cod}_n/\mathsf{nil}]$ and $+\ell[\mathsf{dom}_n/\mathsf{nil}]$ will never be assigned blame therefore the function contract will never be assigned positive blame.

The property of contract soundness is not compositional. Consider a similar example using an intersection contract.

$$(\lambda x.x)@^{\pm\ell[\mathsf{nil}]}((\mathtt{I} \to \mathtt{I}) \to (\mathtt{I} \to \mathtt{I})) \cap ((\mathsf{B} \to \mathsf{B}) \to (\mathsf{B} \to \mathsf{B}))$$

This function will never be assigned positive blame for the same fundamental reason as before, however contract soundness cannot be applied here. After splitting the intersection contract the two function contracts are annotated with branch nodes, not root nodes, and therefore contract soundness does not apply. The limitation is that contract soundness depends upon the guarding contract causing divergence before the outer contract can observe ill-behaviour, but this property is not compositional in the presence of intersection and union.

We would like a supplementary property that is not reliant on contract violations causing divergence, applying uniformly to constituent branches of an intersection or union contract. We introduce the notion of *witness soundness*.

Theorem 4.3.2 (Witness Soundness)**.**

(a) $M@^{-p \gg \mathsf{dom}_n} A \in [\![A]\!]^{+}_{p \gg \mathsf{cod}_n}$

(b) $K \circ \square\, @^{p \gg \mathsf{cod}_n} B \in [\![B]\!]^{-}_{-p \gg \mathsf{dom}_n}$

Proof. See Appendix B. □

For any term M, applying a contract of type A annotated with blame node $-p \gg \mathsf{dom}_n$ yields a term that positively satisfies type A for witness $p \gg \mathsf{cod}_n$. For any continuation K, applying a contract of type B annotated with blame node $p \gg \mathsf{cod}_n$ yields a continuation that negatively satisfies type B for witness $-p \gg \mathsf{dom}_n$.

Witness soundness captures the implication-like quality of function types. If a context assumes responsibility for providing a value of a type A, then a subject may freely assume that the value satisfies type A. If the context fails to provide a satisfying value then the subject is relieved of all obligations. As witness soundness is explicit about the blame nodes, or witnesses, this property can be satisfied by correct blame assignment. The property that all contract violations guarantee divergence does not hold as intersection or union contracts decompose, but blame assignment is unaffected. For example, consider the term:

$$(W @^{-\ell[\mathsf{nil}]\bullet\mathsf{right}^{-}_{\cup}[\mathsf{dom}_n/\mathsf{nil}]}\mathtt{I} \longrightarrow \mathtt{I}) @^{+\ell[\mathsf{nil}]\bullet\mathsf{right}^{+}_{\cup}[\mathsf{cod}_n/\mathsf{nil}]}\mathtt{I} \longrightarrow \mathtt{I}$$

Intuitively the inner contract should guard the outer contract, however we cannot reason using contract soundness because the blame nodes are branches rather than roots. However, we can express the blame nodes as $-p \gg \mathsf{dom}_n$ and $p \gg \mathsf{cod}_n$, and may therefore reason using witness soundness.

Contract soundness and witness soundness are complementary properties as one does not subsume the other. In some cases we would like the strong guarantee that no contract can observe a violation, which is where contract soundness shines. In other cases we only require that a function result is never violated under the condition that the function argument is never violated, which is where witness soundness shines.

4.4 Monitoring Properties of Contracts

In this section we present sound monitoring properties associated with each contract type and Figure 4.3 gives the complete definition. Each property captures the intuitive behaviour associated with a given contract type as well describing how contract satisfaction composes. Properties come in pairs: for each contract type there is a property for values and a property for continuations. The structure of each property is an implication under which satisfaction of a value or continuation is expected to hold, however in the setting of contract violations the interesting consequences are obtained through contraposition. In essence the rules say that if a value (or continuation) violates a contract then there is a meaningful reason for that violation.

$$
\begin{array}{lll}
V \in [\![\iota]\!]^{+} & \text{if} & V : \iota \\
K \in [\![\iota]\!]^{-} & \text{if} & \text{true} \\
V \in [\![\mathsf{any}]\!]^{+} & \text{if} & \text{true} \\
K \in [\![\mathsf{any}]\!]^{-} & \text{if} & \text{true} \\
V \in [\![A \to B]\!]^{+}_{p} & \text{if} & \forall N \in [\![A]\!]^{+}_{p \gg \mathsf{cod}_n}.\ V\,N \in [\![B]\!]^{+}_{p \gg \mathsf{cod}_n} \wedge \\
 & & \quad \forall K \in [\![B]\!]^{-}_{-p \gg \mathsf{dom}_n}.\ K \circ V\,\square \in [\![A]\!]^{-}_{-p \gg \mathsf{dom}_n} \\
K \in [\![A \to B]\!]^{-} & \text{if} & \forall K', N.\ K \longrightarrow^{*}_{\square} K' \circ \square\,N \Rightarrow N \in [\![A]\!]^{+} \wedge K' \in [\![B]\!]^{-} \\
V \in [\![A \cap B]\!]^{+} & \text{if} & V \in [\![A]\!]^{+} \wedge V \in [\![B]\!]^{+} \\
K \in [\![A \cap B]\!]^{-} & \text{if} & K \in [\![A]\!]^{-} \vee K \in [\![B]\!]^{-} \\
V \in [\![A \cup B]\!]^{+} & \text{if} & V \in [\![A]\!]^{+} \vee V \in [\![B]\!]^{+} \\
K \in [\![A \cup B]\!]^{-} & \text{if} & K \in [\![A]\!]^{-} \wedge K \in [\![B]\!]^{-}
\end{array}
$$

$$
\begin{array}{llll}
\text{Terms} & M, N & ::= & \cdots \mid V^{\square} \\
\text{Values} & V, W & ::= & \cdots \mid V^{\square}
\end{array}
\qquad
\frac{\text{for some } \Phi, \Delta, V \quad \langle \Phi, \Delta, K, V^{\square} \rangle \longrightarrow^{*} \langle \Phi', \Delta', K' \circ V^{\square}\,\square, N \rangle}{K \longrightarrow^{*}_{\square} K' \circ \square\,N}
$$

Figure 4.3: Sound Monitoring Properties

Base Types A value V positively satisfies base type ι if V conforms to base type ι. Using contraposition this property says that if V violates contract ι then V is not a constant that conforms to ι. To summarise: *all values that violate the integer contract are not integers.*

A continuation K negatively satisfies base type ι unconditionally. Using contraposition this property says that no context can be assigned negative blame for violating a base type contract.

The any **Type** A value V positively satisfies type any unconditionally. Similarly, a continuation K negatively satisfies type any unconditionally. There are no obligations associated with a contract of type any and therefore the contract should never assign positive or negative blame.

Function Types The property constraining contract satisfaction for values and function types makes use of witness satisfaction. A value V positively satisfies type $A \to B$ for witness p if the value respects a conjunction constraining the result when

V is applied, and how the argument is used when V is applied. The first conjunct states that for all arguments N that satisfy type A for witness $p \gg \mathrm{cod}_n$, applying V to N returns a result that satisfies B, also for witness $p \gg \mathrm{cod}_n$. To summarise: *applying V to satisfying arguments returns satisfying results.* The second conjunct states that for all continuations K that satisfy type B for witness $-p \gg \mathrm{dom}_n$, the composed continuation that first applies V and then passes the result to K satisfies type A, also for witness $-p \gg \mathrm{dom}_n$. To summarise: *applying V in a context that respects the result ensures that V respects the argument.*

Witness satisfaction is used to make the implication naturally associated with function types explicit in the semantics of blame. For example, the first conjunct states that if the witness $p \gg \mathrm{cod}_n$ is unable to provide a counter-example that the argument satisfies the domain type, then the witness should not be able to provide a counter-example that the result satisfies the codomain type.

A pertinent question is whether witness satisfaction is mandatory, or whether the same property can be stated and verified using contract satisfaction. We conjecture that the property can be stated and verified using contract soundness, but at the cost of additional reasoning. For example, when verifying the property and considering the behaviour of V wrapped in a function contract, such as $V@^{p}A \to B$, we must be able to synthesise an argument that satisfies the domain contract. Using witness satisfaction the satisfying argument is immediate; any argument is wrapped as $N@^{-p \gg \mathrm{dom}_n}A$, and $N@^{-p \gg \mathrm{dom}_n}A \in [\![A]\!]^{+}_{p \gg \mathrm{cod}_n}$ follows from Theorem 4.3.2. If we were to use contract satisfaction then $N@^{-p \gg \mathrm{dom}_n}A \in [\![A]\!]^{+}$ is not immediate, or necessarily valid. Instead we must consider three possible cases. First, N satisfies A. Second, N does not satisfy A and the application of V assigns blame to $-p \gg \mathrm{dom}_n$, rendering satisfaction of B trivial because the codomain contract can never be assigned blame. The final case is the most nuanced: N does not satisfy A however function V does not exercise N sufficiently to assign negative blame. In this instance we must be able to show that there is another argument N' that does satisfy A and is observationally equivalent to N in the body of V. For example, if we consider the contraposition of the monitoring property and assume $(\lambda f.42) \notin [\![(\mathtt{I} \to \mathtt{I}) \to \mathtt{B}]\!]^{+}$, we must consider the case that an argument is not satisfying, but does not evoke negative blame. For example:

$$\langle \emptyset, [\,], \mathit{Id}, ((\lambda f.42)@^{p}(\mathtt{I} \to \mathtt{I}) \to \mathtt{B})(\lambda x.\mathtt{true})\rangle \longrightarrow^{*} \langle \lightning\, p \gg \mathrm{cod}_n\rangle$$

then we must be able to show that:

$$\langle \emptyset, [\,], \mathit{Id}, ((\lambda f.42)\,N)@^{p \gg \mathrm{cod}_n}\mathtt{B}\rangle \longrightarrow^{*} \langle \lightning\, p \gg \mathrm{cod}_n\rangle$$

for some argument $N \in [\![\mathtt{I} \to \mathtt{I}]\!]^+$. We cannot use $\lambda x.\mathtt{true}$ or $(\lambda x.\mathtt{true})@^{-p \gg \mathrm{dom}_n \mathtt{I} \to \mathtt{I}}$ as they are not known to satisfy $\mathtt{I} \to \mathtt{I}$, and there is no negative blame violation that makes the case trivial as f is never applied. Instead, we must be able to construct an argument that behaves the same as $\lambda x.\mathtt{true}$ in the current context but also satisfies the domain type, for example $\lambda x.42$.

Using witness satisfaction avoids this complexity and the monitoring property still quantifies over all blame nodes annotating the function contract. The cost is that the premise of the monitoring property is less general as it refers to specific witness nodes, however the mentioned nodes are precisely those that arise naturally in reduction. We consider the validation of a more general rule as future work and the use of witness satisfaction to present a weaker, but useful property, as a contribution of this work.

The property constraining contract satisfaction for continuations and function types makes use of contract satisfaction, not witness satisfaction. The reason is that the property does not encode an implication between domain and codomain types. A continuation K negatively satisfies type $A \to B$ if for every argument continuation of the form $K' \circ \Box\, N$ that K reduces to, then N positively satisfies A and K' negatively satisfies B. The property makes use of context reduction, written $\longrightarrow^*_\Box$, and defined in Figure 4.3. Context reduction:

$$K \longrightarrow^*_\Box K' \circ \Box\, N$$

states that applying continuation K to some value V reduces to a configuration that applies V to N in continuation K'; we use universal quantification to range over all resulting configurations. Context reduction uses notation $V^\Box$ to distinguish the value passed to continuation K; operationally $V^\Box$ behaves identically to V.

Our monitoring properties for function types capture the essence of a function contract. As we do not define satisfaction structurally like Keil and Thiemann (2015a) we avoid having to present detailed rules involving function abstraction, as presented in Section 4.1.2.

Intersection Types A value V positively satisfies type $A \cap B$ if V positively satisfies type A and positively satisfies type B. If a value violates an intersection contract then it violates either the left branch or the right branch.

A continuation K negatively satisfies type $A \cap B$ if K negatively satisfies type A or negatively satisfies type B. If a continuation violates an intersection contract then it violates both the left branch and the right branch.

Union Types A value V positively satisfies type $A \cup B$ if V positively satisfies type A or positively satisfies type B. If a value violates a union contract then it violates both the left branch and the right branch.

A continuation K negatively satisfies type $A \cup B$ if K negatively satisfies type A and negatively satisfies type B. If a continuation violates a union contract then it violates either the left branch or the right branch.

Theorem 4.4.1 (Monitoring Properties)**.** $\lambda^{\cap\cup}$ *satisfies the properties in Figure 4.3.*

Proof. See Appendix C. □

4.5 Comparison

In this section we compare our work to the primary examples of monitoring oriented contract satisfaction and denotational contract satisfaction identified in Section 4.1.

4.5.1 On Contract Satisfaction in a Higher-order World

Dimoulas and Felleisen (2011) define a contract semantics for CPCF, a call-by-value variant of PCF with dependent function contracts. An initial observation is that our definition of contract satisfaction is closely related to their definition. In particular, their definition of contract satisfaction implies our definition. We restate Theorem 5.3 by Dimoulas and Felleisen (2011). We write $V \models A$ to denote that value V satisfies A according to the definition of contract satisfaction by Dimoulas and Felleisen (2011).

$$\text{If } V \models A \text{ then } C[V@^{+\ell}A] \not\Downarrow \texttt{blame } +\ell \text{ for all contexts } C.$$

where $C[V@^{+\ell}A] \not\Downarrow \texttt{blame } +\ell$ is equivalent to our definition $V \in [\![A]\!]^+$. Recall that Dimoulas and Felleisen (2011) define contract satisfaction using obligation splitting as defined in Definition 4.1.1; specifically, using observational equivalence between contracts with full and partial obligations. We restate Definition 5.1 by Dimoulas and Felleisen (2011).

Definition (Tight Contract Satisfaction)**.**

$$V \models A \; \textit{if} \; V@^{+\ell}A \simeq V@^{+\ell}A^-$$
$$C \models B \; \textit{if} \; C[V@^{+\ell}B] \simeq C[V@^{+\ell}B^+]$$

For simple types their definition of contact satisfaction implies our definition. If the language of contracts is extended to include intersection and union then the correspondence is less clear. There is no immediate way to extend the Dimoulas and Felleisen (2011) style of contract satisfaction to include intersection and union. A first attempt may extend Definition 4.1.1 with cases for intersection and union using congruence.

Definition 4.5.1 (Positive and Negative Contract Obligations with $\cap$ and $\cup$)**.**

$$(A \cap B)^+ = A^+ \cap B^+ \qquad (A \cup B)^+ = A^+ \cup B^+$$
$$(A \cap B)^- = A^- \cap B^- \qquad (A \cup B)^- = A^- \cup B^-$$

However consider the following example. The continuation $K \circ \square 4$ negatively satisfies the type $(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B})$, therefore using the Dimoulas and Felleisen (2011) definition of satisfaction suggests that monitoring with the complete intersection contract is observationally equivalent to monitoring with the positive obligations only. Specifically, the following two program configurations should be observationally equivalent.

$$\langle K \circ \square 4, V @^{+\ell} (\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B}) \rangle \simeq \langle K \circ \square 4, V @^{+\ell} (\mathsf{any} \to \mathtt{I}) \cap (\mathsf{any} \to \mathtt{B}) \rangle \quad \textit{for all } V$$

Suppose that we choose a value V that we expect to positively satisfy the contract, such as $\lambda x.x$. The program with the complete contract will evaluate to 4, while the program with the negative obligations erased will raise a blame error, assigning blame to $+\ell$. These programs are *not* observationally equivalent, despite the context *and* the value both intuitively satisfying the contract. Erasing obligations from a contract in a satisfying context should not add blame to a wrapped value, however this is what has occurred. The contract we obtain by erasing negative obligations is $(\mathsf{any} \to \mathtt{I}) \cap (\mathsf{any} \to \mathtt{B})$ and no total function can satisfy this type. Our naive attempt to extend Dimoulas and Felleisen (2011) style contract satisfaction to include intersection and union has failed. We argue that the source of the disparity between Dimoulas and Felleisen (2011) style satisfaction and ours is the particular effect each system observes when monitoring. Dimoulas and Felleisen (2011) observe the effect of raising a contract error, or diverging, and from that deduce blame assignment. Our approach is to observe the effect of assigning blame, and from that deduce contract errors. For simple types the two effects are equivalent: raising a contract error implies that blame has been assigned; assigning blame implies that a contract error is

raised. The introduction of intersection and union breaks this bi-implication: raising a contract error implies that blame has been assigned, however assigning blame does not necessarily imply that a contract error is raised.

Extending the Dimoulas and Felleisen (2011) approach to contract satisfaction may be possible, however the surgical process of separating obligations in a contract requires greater precision.

4.5.2 Higher-order Contracts with Intersection and Union

The system of contract semantics presented by Keil and Thiemann (2015a) supports all the sound monitoring properties we specify in Figure 4.3. The denotational approach of Keil and Thiemann (2015a) means that the properties are satisfied by construction, as all satisfying values and contexts are built using rules that effectively encode the monitoring properties. This should come as no surprise because the monitoring properties we provide are heavily influenced by the work of Keil and Thiemann (2015a) and their definition of contract satisfaction.

Keil and Thiemann (2015a) present a denotational definition of contract satisfaction and therefore there is no equivalent notion of witness satisfaction, which relies on an explicit reference to a blame node annotating a contract. Consequently, there is also no equivalent notion of witness soundness (Theorem 4.3.2). A similar property is likely provable in the Keil and Thiemann (2015a) system, but formulating the property in terms of contract satisfaction may be difficult.

Our implementation of function contracts follows the design of Keil and Thiemann (2015a) which does not conduct any first order checks and therefore a primitive value such an integer trivially satisfies any function contract. Keil and Thiemann (2015a) propose encoding a traditional function contract using an intersection that combines a first-order tag check and a higher-order function component. However, using the definition of contract satisfaction of Keil and Thiemann (2015a) or our sound monitoring property reveals that an intersection contract is not the right operator.

We write $A \mapsto B$ to denote a traditional function contract. Suppose we were to implement $A \mapsto B$ such that $A \mapsto B = (\rightarrow) \cap (A \rightarrow B)$, where $\rightarrow$ is the flat contract type that all abstractions conform to. Following our sound monitoring properties then any continuation that satisfies the type $(\rightarrow) \cap (A \rightarrow B)$ need only satisfy one branch, and furthermore, all continuations satisfy flat contract types such as $\rightarrow$. The consequence is that *all* continuations satisfy the type $A \mapsto B$.

An observation to draw from this behaviour is that intersection contracts should not be used combine types with disjoint eliminators. If the elimination contexts for the types are disjoint then there is no context where both branches of the intersection could be assigned negative blame, and it follows that there is no context where the intersection could be assigned negative blame.

We propose using the *and* contract, which we write as $\sqcap$, and is introduced in multiple contract systems (Findler et al., 2004; Flatt and PLT, 2010). Positive and negative blame for the operator $\sqcap$ is covariant, acting like intersection for positive blame and union for negative blame. The monitoring properties for $\sqcap$ are defined as:

Definition 4.5.2 (Monitoring Properties for $\sqcap$)**.**

$$V \in [\![A \sqcap B]\!]^+ \quad \textit{if} \quad V \in [\![A]\!]^+ \wedge V \in [\![B]\!]^+$$
$$K \in [\![A \sqcap B]\!]^- \quad \textit{if} \quad K \in [\![A]\!]^- \wedge K \in [\![B]\!]^-$$

If $A \mapsto B$ is defined such that $A \mapsto B = (\rightarrow) \sqcap (A \rightarrow B)$, then the monitoring properties tell us that a continuation satisfies $A \mapsto B$ if it satisfies $\rightarrow$ and $A \rightarrow B$. The former condition is trivially satisfied and acts as a neutral constraint, allowing us to restate the condition: a continuation satisfies $A \mapsto B$ if it satisfies $A \rightarrow B$.

4.6 Related Work

4.6.1 Correctness Criteria

Dimoulas et al. (2011) develop a framework for specifying and classifying correct blame assignment and use their framework to evaluate two contract monitoring strategies for dependent function contracts: *lax* and *picky*. The *lax* strategy does not wrap the variable bound in the codomain of the contract with the domain contract, while the *picky* strategy *will* wrap the variable. Two concepts are introduced: ownership and obligation. Ownership relates expressions and blame nodes, attributing the result of an expression to a blame node. Obligations relate contract types and blame nodes, attributing the positive and negative obligations of a contract to a set of blame nodes. Dimoulas et al. (2011) proceed to define *correct blame* using ownership and obligation. Blame correctness guarantees that at the evaluation of every flat contract the positive (server) blame node annotating the contract is the owner of the contracted value, and additionally, the positive blame node is contained in the set of positive obligations for the contract. The *lax* contract monitoring strategy is shown to be blame correct,

while the *picky* strategy is not blame correct. Under the *picky* strategy, the contract that wraps the variable bound in a dependent contract may blame the wrong party.

While the *picky* strategy is not blame correct, the strategy does detect more contract violations than the *lax* strategy. To get the best of both—detecting more violations with correct blame assignment—Dimoulas et al. (2011) introduce a third strategy called *indy*. Every contract monitor is equipped with an additional blame node that represents the contract itself. The *indy* strategy uses this new blame node to wrap the variable bound in a dependent function contract. By distinguishing a third party in contract monitoring, the contract itself, Dimoulas et al. (2011) are able to detect the same violations as *picky* whilst retaining the blame correctness of *lax*.

Dimoulas et al. (2012) observe that while the *indy* strategy is better than the *lax* strategy, the criterion of blame correctness does not distinguish the two. In order to elucidate the differences between *indy* and *lax* they enforce a *single owner* policy and develop the criterion of a *complete monitor*. The single owner policy lifts ownership (Dimoulas et al., 2011) to a fundamental component of term reduction. A term may only reduce if it is the *sole* owner of all arguments in the redex. The criterion of a complete monitor extends blame correctness. The criterion ensures that for any violation indicating a failed contract, the owner of the violating value matches the positive blame node decorating the violated contract, and additionally, the contract belongs to the positive obligations of that node. Furthermore, complete monitoring also requires progress: no term gets *stuck*. The *indy* strategy is shown to be a complete monitor, while the *lax* strategy is not a complete monitor. Specifically, the *lax* strategy can violate the single owner policy causing terms to get stuck. The *picky* strategy is also not a complete monitor because it does not ensure blame correctness.

We conjecture that our calculus is a complete monitor, but primarily for the uninteresting reason that is we omit user-defined and dependent contracts. In our setting it does not make sense to distinguish between *lax*, *pick*, or *indy*. Keil and Thiemann (2015a) support user-defined contracts but their calculus is not a complete monitor. They drop contracts that cross into other contracts from separate intersection or union branches. If we extend our calculus to support user-defined contracts in their style then we would similarly fail the complete monitoring property. Adding user-defined contracts and maintaining complete monitoring is future work.

4.6.2 Monitoring Semantics

In the monitoring semantics we present, all monitoring is synchronous and within the same execution context, however contract monitoring is not restricted to this design. Swords et al. (2018) study a variety of monitoring strategies such as strict, lazy, and concurrent, and provide a core theoretical framework capable of describing each. Their key insight is to describe contract monitoring as *communication* between a monitor and a contracted program. For example, application of an eager contract monitor can be thought of as spawning a new process to perform the contract check, sending the contracted value along a channel to the new process, and then awaiting the result which is either the initial value or a contract error. Alternatively, a promise-based contract monitor can be thought of as spawning a new process to perform the contract check, sending the contracted value along a channel to the new process, and then returning a promise that contains the result of the contract application. The communication oriented framework presented by Swords et al. (2018) is compositional, permitting contracts for data structures to combine multiple strategies. Extending their work to support intersection and union contracts would be an interesting line of work. Intersection and union operators naturally lend themselves to parallel monitoring; Keil and Thiemann (2015a) first explain their semantics for intersection and union using a non-deterministic parallel operator.

Findler et al. (2004) investigate the semantics of contracts as pairs of projections, building a denotational categorical model of contracts. Specifically, they describe errors as pairs of error projections: idempotent functions that may additionally return an error value. The denotational model they construct is fully abstract with respect to the calculus SPCF, and all semantic contracts may be written in SPCF. The question that they pose is whether the contract operators of Findler and Felleisen (2002) are sufficient to represent all semantic contracts, and the answer they find is *"no"*. In particular, the existing operators are unable to compose first-order and higher-order contracts, leading them to introduce the *and* operator we discuss in Section 4.5.2.

Findler and Blume (2006) also investigate contracts as pairs of projections. They begin by discussing a contract implementation based on the composition of projection functions. Their analysis finds that a projection-based implementation is more efficient than the traditional pattern-matching approach, where contract types are reified as data-structures. Their work also considers the ordering that arises between contracts; an ordering between contracts A and B means that A raises a violation *at*

least as often as B. The ordering raises the question: what is the highest contract in the ordering, a contract we might call the any contract. Perhaps surprisingly, the semantics of the highest contract is not the semantics of any that we present in Section 4.4. Instead, the highest contract is one that *never* assigns positive blame, and *always* assigns negative blame.

Chapter 5

Contracts for Gradual Typing

In this chapter we make the transition from the design of contracts to their application. We present *The Prime Directive*: a tool that implements sound gradual typing in TypeScript and is based on the technical foundations from previous chapters and blame calculus. Our tool uses contracts to monitor JavaScript libraries and TypeScript clients for conformance to the corresponding TypeScript definition file. We show how JavaScript proxies can be used to implement higher-order contracts, including parametric polymorphic contracts.

Sound gradual typing should satisfy non-interference. Monitoring a program should never change its behaviour, except to raise additional type errors when a program does not conform to the expected type. This idea is the inspiration for the name *The Prime Directive*. In the *Star Trek* universe, members of Starfleet must follow the Prime Directive: personnel should not interfere with the development of monitored civilisations and planets. Viewers of the television series will know that the characters flagrantly ignore the directive; proxies in JavaScript are equally disobedient. Opaque proxies, as they are implemented in JavaScript, do not guarantee non-interference. We discuss how proxies that are used to implement contracts can violate non-interference.

5.1 Introduction

There is now a variety of widely used programming languages that exhibit gradual typing in some form such as C#, Clojure, Dart, Python, Racket, Flow (Chaudhuri et al., 2017), and TypeScript (Bierman et al., 2014). Flow is a static type checker for JavaScript that favours soundness when possible, but allows integration with dynam-

ically typed code through the any type. TypeScript is a superset of JavaScript that supports optional type annotations with the primary aim of enhancing developer tools such as code editors and documentation.

5.1.1 TypeScript

A Pragmatic Approach Type annotations are used within TypeScript for both type inference and type checking. Type inference improves the accuracy of auto-completion by suggesting properties compatible with the inferred type. Type checking prevents certain programming errors such as accessing unavailable properties. TypeScript favours pragmatism over rigorous correctness and is *unsound by design* (Bierman et al., 2014). The type checker supports unsound rules such as bivariant subtyping for functions and covariant subtyping for mutable objects, but it is the distinguished type any that best distils TypeScript's philosophy. A programmer may opt out of static type checking through use of the dynamic type any, and opt back in through the addition of an explicit type annotation. This provides the option to sacrifice static safety if the programmer is sure the program will not exhibit bad behaviour when executed, or the perceived development cost of maintaining safety is too high. TypeScript adopts the weakest form of gradual typing, the *erasure embedding* (Greenman and Felleisen, 2018), which retains no type information when compiling TypeScript to executable code.

```
1 const XZ: any = { x: 3, z: false };
2 const XY: { x: number; y: boolean } = XZ;
```

Here XZ has a field with label z while the declared type of XY has a field with label y. Annotating XZ with type any delegates all responsibility for the type of XZ from TypeScript to the programmer. There are no checks, static or dynamic, that ensure the untyped identifier XZ conforms to the static type declared at the assignment to identifier XY. Type inference will provide auto-complete suggestions for XY that are incompatible with the value at run-time. Type checking will miss errors that arise through accessing non-existent property y.

TypeScript with JavaScript A crucial aspect to TypeScript's popularity is supporting the use of existing JavaScript libraries within TypeScript applications. An untyped JavaScript library is paired with a type specification of its API, a *definition*

file, and is then imported into a TypeScript client. DefinitelyTyped[1] is the primary repository of definition files for JavaScript libraries, hosting over 4000 definitions. A definition file acts like a type annotation for a library, assisting auto-completion and providing documentation for clients. Libraries and definitions are disjoint entities, making it possible to apply definitions to legacy JavaScript libraries without modification of the library code.

Conformance A JavaScript library and TypeScript client should conform to the definition. When calling a library function a client should provide arguments of the correct type, while the library should return a result of the correct type. However, as was the case with type annotations, TypeScript takes a similarly pragmatic view when checking conformance to the definition. JavaScript is untyped so there is no checking for a library's conformance to the definition. TypeScript is checked but unsound, so there is no guarantee of a client's conformance to the definition. Each definition file is maintained separately to the library it describes and is often written by a different author than the library. Mistakes can be easily introduced and left unchecked, or the library can change causing the definition to become outdated. If the definition file and library do not conform then auto-completion suggestions can be misleading, producing unexpected results or introducing errors.

5.1.2 Contracts for Sound Gradual Typing

A Safer Approach Sound gradual typing permits typed and untyped code to coexist but does not follow the approach taken by TypeScript that chooses to forgo safety. Instead, dynamic checks are inserted at the typed-untyped boundary to ensure that dynamically typed code conforms to the expected static type an run-time. We apply this approach to JavaScript libraries with TypeScript definitions. Dynamic checks are inserted at the library-client boundary to ensure that library and client code conforms to the definition file. Contracts are an apt mechanism for implementing these dynamic checks because they monitor both parties and can be applied without modifying library code. Instead, contracts are applied when a library is imported, wrapping the library.

Implementing Contracts Proxies are presented as a suitable mechanism for implementing contracts in JavaScript because they allow new behaviours to be retroac-

[1] `https://definitelytyped.org`

tively added to existing objects and functions. An object can be wrapped in a proxy to intercept property access and updates. A function can be wrapped in a proxy to intercept the arguments and result during an application. Using proxies, we can apply additional dynamic type checks to existing JavaScript code, without modifying the underlying code. Proxies are now a standard feature in JavaScript run-times making them readily accessible to programmers.

Non-Interference One important property of sound gradual typing is noninterference. Adding dynamic type checks to an existing program should not change its behaviour, except to raise an error should a value fail to conform to its specified type. Proxies have many attributes suitable for implementing contracts, however opaque proxies (as they are implemented in JavaScript) do not satisfy non-interference. Applying a proxy to an object does not retain the object's identity. Consequently, the same equality test can return different results when evaluated with and without proxies. Applying a proxy to primitive values requires changing their type. Consequently, the same type test can return different results when evaluated with and without proxies.

The threat that proxy interference poses to sound gradual typing in practice is unclear. Keil and Thiemann (2013) discuss the consequences of using opaque proxies, but do not evaluate the scale of the problem. Keil et al. (2015) give insight by analysing how often proxies alter equality tests when used to implement contracts for JavaScript benchmarks, however it is unclear if JavaScript libraries exhibit the same issues as JavaScript benchmarks.

Previous work does not tell the whole story: there is an additional source of interference caused by the use of parametric polymorphic contracts. A parametric polymorphic contract employs run-time sealing to enforce data-abstraction, however seals implemented using proxies violate non-interference by altering the sealed value's type. This kind of interference has never previously been measured in practice.

5.1.3 Overview

Section 5.2 introduces *The Prime Directive*, explaining how the tool wraps a JavaScript library to enforce conformance to the TypeScript definition file. Section 5.3 illustrates how proxies are used to implement higher-order contracts, and how contracts may violate non-interference. Section 5.4 discusses related work regarding sound gradual typing for TypeScript and implementing contracts.

Definition: `basic.d.ts`

```
interface Point {
  x: number;
  y: boolean;
}

export function getX(obj: Point): number;
export function getY(obj: Point): boolean;
```

Library: `basic.js`

```
module.exports.getX = obj => true;
module.exports.getY = obj => obj.y;
```

Client: `basic-client.ts`

```
import * as basic from "./basic";

const XZ: any = { x: 3, z: false };
const XY: { x: number; y: boolean } = XZ;
const x: number = basic.getX(XY);
const y: boolean = basic.getY(XY);
```

Figure 5.1: Example Definition, Library, and Client

5.2 The Prime Directive

The Prime Directive is a tool that generates a contract from a definition file, and the contract monitors a library and client for conformance to the definition. When a library does not conform to the definition the contract assigns positive blame; when a client does not conform to the definition the contract assigns negative blame. Figure 5.1 illustrates a basic example of a definition, library, and client.

5.2.1 Definitions, Libraries, and Clients

Definition A definition file (with suffix `.d.ts`) is a set of TypeScript types describing the API of a library. Our evaluation uses libraries that are packaged as node.js modules, though TPD also works for other systems. When a library is deployed for

Definition (Alternate): `basic-assignment.d.ts`

```
1  interface Point {
2    x: number;
3    y: boolean;
4  }
5
6  interface Library {
7    getX(obj: Point): number;
8    getY(obj: Point): boolean;
9  }
10
11 declare const API: Library;
12 export = API;
```

Figure 5.2: Example Definition using export assignment

node.js the API of the library is packaged as a JavaScript object that is then consumed by a client. A definition file for a library describes the type of this object, where the `export` keyword is used to indicate the exported API members, or the the properties of the packaged object. Definition file `basic.d.ts` describes a library that exports two functions: `getX` and `getY`. The former accepts an argument of type `Point` and returns a result of type `number`; the latter accepts an argument of type `Point` and returns a result of type `boolean`. Definitions may define auxiliary types, such as `Point`. The `Point` type is implemented as an interface and describes an object with two properties: `x` of type `number`, and `y` of type `boolean`.

There is an alternate way to write this definition file that directly describes the type of object representing the library, rather than implicitly describing each of the exported members, or properties of the object. An export assignment `export = API` states that the type of the library, or API, is the type of identifier `API`. Figure 5.2 demonstrates an alternate way to write `basic.d.ts`. The interface `Library` directly describes the type of the library with exported members, or properties, `getX` and `getY`. The identifier `API` is assigned the type `Library` and exported using an export assignment, indicating that any client that imports the library will receive an object that conforms to the type `Library`. A definition file is not allowed to have an export assignment and exported members—they are mutually exclusive.

Library A library is a collection of JavaScript files (with suffix `.js`) implementing the API. When a library is packaged for node.js the API of the library is assigned to the predefined JavaScript object `module.exports`. The library `basic.js` assigns functions to the properties `getX` and `getY` on the object `module.exports`. The syntax `obj => true` denotes an anonymous function that accepts a parameter named `obj` and returns the boolean constant `true`. The syntax `obj => obj.y` denotes an anonymous function that accepts a parameter named `obj` and returns the value assigned to property `y` of `obj`.

Client A client is a collection of TypeScript files (with suffix `.ts`) using the API. The library `basic.js` is imported using the syntax:

```
1 import * as basic from "./basic";
```

The entire API (denoted by `*`) is imported and bound to identifier `basic`. Concretely, the `module.exports` object in `basic.js` is assigned to identifier `basic` in `basic-client.ts`. The client code applies each of the exported functions `getX` and `getY` to the argument `XY`, binding the result to identifiers `x` and `y` respectively. Both functions require an argument of type `Point` while the client supplies an argument of type `{ x: number; y: boolean }`, an anonymous object type. TypeScript has a *structural* type system therefore types `Point` and `{ x: number; y: boolean }` are equivalent. Consequently, the client code is deemed valid by the type checker.

Conformance A careful reader may have noted that there are two issues of conformance present in Figure 5.1. The first violation of conformance originates in the library. The implementation of the function `getX` does not conform to the specified type because the it returns a boolean when a number is expected. No checking of the library is done and the error goes without reprimand. The unsuspecting client assumes the library is honest and assigns the result of applying `getX` to an identifier of type `number`, yet at run-time this identifier will be bound to a boolean.

The second violation of conformance originates in the client. Both function applications are supplied the argument `XY` that is assumed to conform to the type `Point`, however the value `XY` lacks the property `y` and therefore does not conform to the type `Point`. During the application of function `getY` the property access will fail and return the distinguished value `undefined`. The identifier `y`, declared to be of type `boolean`, will be assigned the incompatible value `undefined` at run-time. The client code has sabotaged itself through an illegal application of a library function!

Although the client code is type checked no static violation is raised; the client has used the dynamic escape-hatch any to create an ill-typed value.

5.2.2 Contracts for Definitions

TPD enforces conformance by generating a contract for a definition file, where the generated contract is a specification for the object exported by the library. In Figure 5.1, the definition states that the exported object has two properties of function type: `getX` and `getY`. TPD will generate a value that represents the contract that can then be used by monitoring code to enforce conformance. The corresponding contract for the definition in Figure 5.1 is:

```
1 const PointC = TPD.obj({
2   x: TPD.num,
3   y: TPD.bool
4 });
5 const basicC = TPD.obj({
6   getX: TPD.fun([PointC], TPD.num),
7   getY: TPD.fun([PointC], TPD.bool),
8 });
```

The identifier `PointC` corresponds to a contract for the interface `Point`. The contract is constructed by the TPD contract combinator `TPD.obj` that creates an object contract from a mapping of properties to contracts. The contract `TPD.num` represents the contract for the type `number`; the contract `TPD.bool` represents the contract for the type `boolean`.

The identifier `basicC` corresponds to a contract for the entire library, or more specifically, the object exported by the library. This contract similarly uses the combinator `TPD.obj`, and additionally uses the combinator `TPD.fun` to build function contracts. The contract operator `TPD.fun` accepts a list of contracts denoting the function arguments, and a contract denoting the function result.

TPD does not provide an explicit contract operator for all TypeScript types such as functions with overloaded call signatures, or functions with optional arguments. Instead, TPD encodes equivalent contracts using intersection and union combinators.

Intersection TPD uses intersection types to encode overloaded functions. For example, given the following interface that contains overloaded function `negate`:

```
interface Calc {
  negate(x: boolean): boolean;
  negate(x: number): number;
}
```

the corresponding contract would be:

```
const CalcC = TPD.obj({
  negate: TPD.intersection(
    TPD.fun([TPD.bool], TPD.bool),
    TPD.fun([TPD.num], TPD.num)
  )
});
```

The contract for `negate` is an intersection of two function contracts, one for each overload.

Union TypeScript also permits optional properties and arguments. An optional property, written `x?: T`, states that property `x` either has a value of type `T` or is undefined. An option argument, also written `y?: T`, states that argument `y` may be a value of type `T`, or the value is omitted from the application. TPD does not have an explicit representation for optional properties and arguments, instead choosing to use union types. For example, given the following interface that features optional parameters and arguments:

```
interface Math {
  MAX_INT?: number;
  round(x: number, digits?: number): number
}
```

the corresponding contract would be:

```
const MathC = TPD.obj({
  MAX_INT: TPD.union(TPD.num, TPD.undef),
  round: TPD.fun(
    [TPD.num, TPD.union(TPD.num, TPD.undef)],
    TPD.num
  )
});
```

Library (Wrapped): `basic.js`

```
1  module.exports.getX = obj => true;
2  module.exports.getY = obj => obj.y;
3
4  // --- generated wrapper code ---
5  const PointC = TPD.obj({
6    x: TPD.num,
7    y: TPD.bool
8  });
9  const basicC = TPD.obj({
10   getX: TPD.fun([PointC], TPD.num),
11   getY: TPD.fun([PointC], TPD.bool),
12 });
13 module.exports = TPD.assert(module.exports, basicC);
```

Figure 5.3: Wrapped Library

The contract for optional property `MAX_INT` is the union of the contracts `TPD.num` and `TPD.undef`, where the latter is the contract for the unit type `undefined`. The contract for the optional argument `digits` in function `round` is similarly a union contract that accepts numbers or the distinguished value `undefined`.

Note At the time this evaluation was performed TypeScript did not have explicit intersection (&) types. This is no longer the case, though the TPD combinator `inter` can be used to model the explicit TypeScript type.

5.2.3 Applying Generated Contracts

Having determined the contract type for the definition file, TPD must then apply the contract to the library. The function `TPD.assert(v, c)` applies the contract type `c` to the value `v`, ensuring that `v` and its context conform to the type specified by `c`.

One approach to apply the contract to a library is to append wrapper code to the library file. The wrapped version of the library in Figure 5.1 is given in Figure 5.3. Wrapper code assigns any auxiliary contracts that are required and then wraps the library implementation before exporting using the same `module.exports` object. Any client that imports the library will now receive a version that is wrapped

by `TPD.assert`, instead of the original library. This approach avoids modifying existing library code, with the exported library functions unchanged.

An alternate approach that avoids adding *any* code to the library is to place the wrapper code at the point the library is imported by the client. This approach might be preferable if the client does not have access to the library code, or the library code is prone to change while the client is not.

The technique adopted in our evaluation is the first approach, appending wrapper code to the library. This method only requires modifying one file rather than every client test file that uses the library.

5.3 Implementing Contracts

In addition to generating wrapper code from definition files, TPD also implements the contracts that perform dynamic type checking. In this section we discuss the details of these contracts, how they are implemented using proxies, and how they can violate non-interference.

5.3.1 Contract Assertion

The entry-point for enforcing contracts using TPD is the `assert` function. The definition of `assert` and auxiliary function `check` is given in Figure 5.4.

Function `assert` generates a fresh root blame node and then proceeds to call `check`. Our implementation assigns a unique identifier to each root node; an alternate implemenation may also want to provide textual meta-data to the root node to aid debugging.

Function `check` applies the correct checks to the value given the specified contract type. Each type contains a `kind` discriminator that enables dispatch using a `switch` operator. Essentially, `type` is an algebraic data type and the switch performs pattern matching. Every clause in the switch dispatches to the appropriate function; for example, when applying a number contract with kind `Num`, the `checkNumber` function is applied to the value. An alternate approach to contract implementation is to use a pair of projection functions, rather than pattern matching (Findler and Blume, 2006; Findler et al., 2004). Findler and Blume (2006) observe that using projections can yield significant performance improvements.

```
1  function assert(value, type) {
2    return check(value, root(), type);
3  }
4
5  function check(value, p, type) {
6    switch(type.kind) {
7      case TypeKind.Num:
8        return checkNumber(value, p);
9      case TypeKind.Bool:
10       return checkBoolean(value, p);
11     case TypeKind.Fun:
12       return wrapFunction(value, p, type);
13      ···  // other cases omitted
14   }
15 }
```

Figure 5.4: Contract Assertion

5.3.2 First-order Contracts

A contract for a first-order type such as `number` or `boolean` can be implemented as a predicate because a value can be immediately inspected for conformance to a first-order type.

Figure 5.5 illustrates how contracts for the types `number` and `boolean` are implemented. Each function accepts the value to be tested and a blame node `p`. When the value has the correct run-time type tag, obtained using operator `typeof`, then the value is returned immediately having satisfied the contract. When the value does not have the correct type a violation is raised using function `blame`.

The function `blame` is defined using the semantics defined in Chapter 3, with two differences. First, the implementation implicitly stores the blame state in blame node `p`, rather than explicitly passing it as an argument. Second, when blame resolves to a top level contract the violation is logged rather than thrown as an exception. This allows TPD to collect all violations in a single pass, rather than halting at the first. The argument `value` supplied to `blame` is return by the function unmodified.

```
1  function checkNumber(value, p) {
2    return typeof value === "number" ?
3      value :
4      blame(p, value);
5  }
6
7  function checkBoolean(value, p) {
8    return typeof value === "boolean" ?
9      value :
10     blame(p, value);
11 }
```

Figure 5.5: First-order Contracts

5.3.3 Function Contracts

As noted previously, checking that a value conforms to a function type cannot be done by immediate inspection of the value, therefore we wrap each application of that value. When applied, the arguments to the function are wrapped according to the domain type, the underlying application is then evaluated, and the result of the function is wrapped according to the codomain type.

Function wrappers in TPD are implemented using the JavaScript Proxy API proposed by Van Cutsem and Miller (2010, 2013). The proxy constructor accepts two arguments: the **target** object that is to be replaced by the proxy, and a **handler** object that holds the traps to be attached to the proxy. A trap is a function that is designed to intercept an operation on a proxy; traps include property accesses, property update, and function application. A supplied handler may not implement all traps. When a trap is omitted from the handler the default behaviour is implemented. For example:

```
1 const obj = {x: 3, z: false};
2 const proxiedObj = new Proxy(obj, {});
```

wraps `obj` in a proxy with an empty handler. Every trap adopts the default behaviour and no additional functionality is added to `proxiedObj`.

Function contracts, or wrappers, can be implemented using a proxy with an `apply` trap that performs wrapping. Each function contract corresponds to a single proxy which simplifies the implementation but can lead to a significant performance loss.

```
1  function wrapFunction(value, p, type) {
2    if(typeof value !== "function") {
3      return blame(p, value);
4    }
5    const handler = {
6      apply: (target, thisArg, argumentsList) => {
7        const wrappedArguments =
8          checkArr(argumentsList, negate(p), type.dom);
9        const result =
10         target.apply(thisArg, wrappedArguments);
11       return check(result, p, type.cod);
12     }
13   }
14   return new Proxy(value, handler);
15 }
```

Figure 5.6: Function Contracts

Siek and Wadler (2010), Siek et al. (2015a), and Feltey et al. (2018) consider different techniques for merging multiple wrappers.

Figure 5.6 presents a simplified implementation of function wrappers in TPD. A value is first checked for conformance to the JavaScript function type prior to wrapping in a proxy. When the value is not a function, blame is immediately assigned to node `p` and the value is returned without wrapping. This behaviour deviates from the semantics presented in Chapter 3 which do not perform any immediate checks and wraps the value unconditionally. We choose a different behaviour because when emulating the semantics of TypeScript types we are always required to perform the first order check, so including the check in `wrapFunction` simplifies the implementation.

When the supplied value is a function we construct the handler for the corresponding proxy. The handler consist of the single trap, `apply`, that accepts three arguments: `target`, the function being applied; `thisArg`, the *this* argument for the application; and `argumentsList`, the array of arguments for the application. For example, given the following application where function `getY` is wrapped in a proxy:

```
1 const y: boolean = basic.getY(XY);
```

then during execution of an `apply` trap

- `target` will be `getY`
- `thisArg` will be `basic`
- `argumentsList` will be `[XY]`

The details of the trap follow the previously described behaviour of function wrappers: the arguments are checked according to the domain type, the function is applied, and the result is checked according to the codomain type. When the arguments are checked on Line 8 the blame node is negated, denoting that the obligation to respect the domain type belongs to the context, not the subject. We use the function `checkArr` to map `check` over arrays of arguments and types.

Object contracts are implemented in a similar way to function contracts except that the modified traps are `get` and `set`, rather than `apply`. The `set` operation is contravariant so the associated blame node is negated in the implementation of the trap. Function contracts are fundamentally "lazy" and must be implemented using a wrapper, which is not true in general for object contracts. An alternate implementation of object contracts might eagerly traverse the entire object and return the original object without applying a wrapper. TPD implements lazy object contracts rather than eager contracts for the following reasons. First, eager contracts do not guard against update operations so they are unable to handle mutable objects which is the default in TypeScript. Second, eager contracts may incur a significant performance reduction by traversing a large data-structure such as a tree. Last, eager contracts introduce additional property accesses not in the original program. A JavaScript object may have custom "getters" that perform side-effects, so an eager contract that traverses an object may add side-effects which violates non-interference.

5.3.4 Parametric Polymorphic Contracts

TypeScript supports parametric polymorphic types, or generics, and correspondingly TPD supports polymorphic contracts.

Parametricity A function that is polymorphic in the type of its argument must act uniformly on that argument, independently of the particular value or type the argument has. This requirement provides a strong form of data abstraction, known as the *abstraction theorem*, or *parametricity* (Reynolds, 1983; Wadler, 1989). Formally,

parametricity states that a function should map related inputs to related outputs, and in particular, a polymorphic function should preserve *all* relations. For example, given a function f of type $\forall X.X \to X$ and arguments x and y related by *any* relation, then $f(x)$ and $f(y)$ must also be related by the *same* relation. Wadler (1989) demonstrated that parametricity can provide *free theorems* about the behaviour of functions of a given type, without needing to inspect the implementation. For example, given a function of type $\forall X.X \to X$, then the only total function that conforms to this type is the identity function.

Dynamically enforcing conformance to a polymorphic type can be tricky. Suppose we declare a function `special` with the following polymorphic type:

```
1 declare function special<X>(x : X): X;
```

then the following implementation does not satisfy parametricity:

```
1 function special(x) {
2   if (typeof x === "number") return x + 1;
3   else return x;
4 }
```

When the argument is of type number the function will return the incremented argument and when the argument is of any other type the function will return the unmodified value—the function does not treat *all* types the same. A contract that checks `special` always returns a value with the same type as the argument is insufficient: `special` will always preserve the type of its argument, but it will not preserve type abstraction.

RTG We adopt the technique of Matthews and Ahmed (2008) and Ahmed et al. (2017) that uses run-time type generation (RTG) to implement polymorphic contracts that employ *sealing* to enforce parametricity. A seal is an opaque value that cannot be inspected or modified. When a value enters a polymorphic function we seal the value with a freshly generated type, or *name*. When a value exits a polymorphic function we unseal the value, provided the value is a seal of the same name.

Figure 5.7 shows the extension of function `check` to include polymorphic contracts. Checking a value against polymorphic contract $\forall X.A$ requires checking the value against contract A after substituting all occurrences of X in A with a fresh name. Function `substituteNames` implements name generation and substitution, additionally distinguishing names that appear in covariant and contravariant positions. For

```
1  function check(value, p, type) {
2    switch(type.kind) {
3      case TypeKind.Num:
4        return checkNumber(value, p);
5      case TypeKind.Bool:
6        return checkBoolean(value, p);
7      case TypeKind.Fun:
8        return wrapFunction(value, p, type);
9      case TypeKind.Forall:
10       return check(value, p, substituteNames(type));
11     case TypeKind.Name:
12       const fn = type.covariant ? unseal : seal;
13       return fn(value, p, type);
14     ···  // other cases omitted
15   }
16 }
```

Figure 5.7: Contract Assertion (Extends Figure 5.4)

example, when applied to the identity function type `substituteNames` is defined as:

$$\texttt{substituteNames}(\forall X.X \to X) \;=\; Y^- \to Y^+ \text{ for fresh name } Y$$

Type Y denotes a freshly generated name and each name is indexed by a polarity indicating the variance of the name with respect to the original type abstraction. We write + for covariance and − for contravariance.

When checking a generated name contract we inspect the variance of the name. If the name is contravariant with respect to the type abstraction then the currently checked value is entering a polymorphic function, so the value must be sealed. If the name is covariant with respect to the type abstraction then the currently checked value is exiting a polymorphic function, so the value must be unsealed.

Sealing The implementation of the `seal` function is given in Figure 5.8. A store is used to track all seals in the program. Constant `SEAL_STORE` is a mapping from seals to seal data, where seal data records the value and name associated with a seal. A

```
1  const SEAL_STORE = new WeakMap();
2  function seal(value, p, name) {
3    // Create coffer to mask value's type
4    const coffer = { };
5    const handler {
6      get: function(target, property, receiver) {
7        return blame(negate(p), value[property]);
8      },
9      set: function(target, property, x, receiver) {
10       return blame(negate(p), value[property] = x);
11     },
12      ···  // rest of traps omitted
13   }
14   const seal = new Proxy(coffer, handler);
15   SEAL_STORE.set(seal, {value: value, name: name});
16   return seal;
17 }
```

Figure 5.8: Sealing

WeakMap[2] implements the store therefore seals do not have to be explicitly deleted to conserve memory.

The `seal` function consists of three key steps. First, a coffer is created for the sealed value; a proxy will be later applied to the coffer to implement the seal. The purpose of the coffer is to obscure the type of the proxied value. Wrapping an object in a proxy will return a proxy of type `object`; wrapping a function in a proxy will return a proxy of type `function`. By using a coffer the proxy that implements the seal will always have the same type, therefore any run-time type tests applied to the seal will behave uniformly. The coffer also enables primitives to be sealed because primitive values cannot be directly wrapped in proxy.

Second, a handler for the proxy is created. A parametric function should not inspect its argument therefore a seal should raise blame when examined by a context. The handler implements every trap, raising blame and then forwarding the behaviour

[2] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap`

```
1  function unseal(value, p, name) {
2    if(SEAL_STORE.has(value)) {
3      const contents = SEAL_STORE.get(value);
4      if(contents.name === name) {
5        // Unseal
6        return contents.value;
7      }
8      // Sealed under different name;
9      return blame(p, contents.value);
10   }
11   // Not a sealed value
12   return blame(p, value);
13 }
```

Figure 5.9: Unsealing

to the sealed value. When tampered with, a seal will blame the negation of the blame node associated the with seal. We say the context of the seal is to blame for tampering, not the sealed value. Finally, the proxy implementing the seal is created, added to the seal store, then returned. Every seal is associated with the value it seals and the name the value was sealed under.

We use a store to associate seal data rather than recording the data on the seal for two reasons. First, the seal store provides an unforgeable way to identify a seal. A token that is placed on the seal could be extracted and used to clone the seal. Conversely, the store is not publicly accessible and recognises seals by their object identity. A seal is uniquely determined by recording its identity in the store, and only a parametric contract may add a seal to the store. Second, recording data on the seal runs the risk of raising a tamper violation when queried by an unsealing contract. Implementing hidden properties that do not trigger tamper violations when queried is possible, but technical. Recording data in the store avoids this issue and is easier to implement.

Unsealing The implementation of the `unseal` function is given in Figure 5.9. First, the seal store is queried to determine if the supplied value is a seal. In the event the value is not a seal then blame is immediately assigned to the blame node. When attempting to unseal a value known to be a seal the corresponding seal data is extracted

from the store. If the name associated with the seal and the name associated with the unseal operation match this denotes a value entering and then exiting a polymorphic function at the same type parameter—the unsealed value is returned. If the names do not match then an illegal attempt has been made to unseal a value that was sealed under a *different* name—blame is assigned to the corresponding blame node.

Type names play a crucial role in the implementation of polymorphic contracts because sealing alone is insufficient to enforce parametricity. Consider the following definition that declares the type of function noSwap.

```
1 interface Pair<X, Y> {
2   fst: X;
3   snd: Y;
4 }
5 declare function noSwap<X,Y>(
6   pair: Pair<X,Y>
7 ): Pair<Y,X>;
```

Parametricity states that the only total function with the corresponding type is the function that accepts a pair and returns a new pair with the values assigned to `fst` and snd swapped. Suppose a library (incorrectly) implements the function such that it performs no swapping of the values:

```
1 function noSwap(pair) {
2   return { fst: pair.fst, snd: pair.snd };
3 }
4 const swapped = noSwap({ fst: 1, snd: true });
5 const one = swapped.snd;
```

Applying a polymorphic contract to noSwap will seal `pair.fst` and `pair.snd` in the body of the function, and unseal properties `fst` and snd when accessed on a value returned by noSwap.

A contract implementation that employs sealing without names (alternatively stated as an implementation that uses a single name) will not assign blame when accessing swapped.snd, which is incorrect. Without attributing names to each seal it is impossible to distinguish seals corresponding to different type parameters, and therefore the contract implementation cannot verify that the contents of the pair were swapped. TPD uses fresh names to correctly assign blame in this example. Upon accessing swapped.snd the object contract will attempt to unseal the value using name

Definition

```
interface Box {
  value: number;
}

declare function isY(x: Box): boolean;
```

Library

```
const y = { value: 4, label: "foo" };
function isY(x) {
  return x === y;
}
```

Figure 5.10: Proxy-object Identity Comparison

X, but will receive a value sealed using name Y, triggering a violation.

5.3.5 Non-Interference

Contracts should satisfy the property of *non-interference*, which in this context means that adding contracts to a program should not change the behaviour of the program, except to raise a blame error. The design of proxies in JavaScript means that contracts implemented using proxies may violate non-interference.

Proxy Identity Proxies in JavaScript are implemented as *opaque* proxies, rather than the alternate approach of *transparent* proxies (Keil et al., 2015). An opaque proxy has a distinct identity which means that the identity of a proxy is different to the object it wraps, and furthermore, two proxies of the same object have different identities. Figure 5.10 shows how an equality test may be affected by a proxy-object comparison. Function `isY` captures a references to unwrapped object `y`. If the function is wrapped according to the type declared in the definition then every argument will be wrapped in a contract of type `Box`. Each contract will be a proxy with a fresh identity, therefore this function will always return false, even when passed the original reference `y`.

Figure 5.11 shows how an equality test may be affected by a proxy-proxy comparison. Function `boxEqualsLabelled` compares its arguments against each-other. If the function is wrapped according to the type declared in the definition then both

Definition

```
interface Box {
  value: number;
}

interface Labelled {
  label: string;
}

declare function boxEqualsLabelled(
  x: Box, label: Labelled
): boolean;
```

Library

```
function boxEqualsLabelled(x, labelled) {
  return x === labelled;
}
```

Figure 5.11: Proxy-proxy Identity Comparison

arguments will be wrapped in a different contract. Each contract will be a proxy with a fresh identity, therefore this function will always return false, even if called with the same arguments such as `boxEqualsLabelled(x, x)`.

Proxy Type Proxies in JavaScript allow a programmer to intercept many operations on an object, but `typeof` is not one. Consequently, when implementing dynamic seals it is not possible to raise an error when applying the non-parametric operation `typeof` to a sealed value. Figure 5.12 shows how a type test may be affected by a dynamic seal. If the function `special` is wrapped using the identity function type then inside the body of the function argument `x` will be sealed. As shown earlier, TPD wraps every value in a *coffer* prior to sealing, meaning that every seal has type `object`. The wrapped function will always select the `else` branch because the type comparison always returns false, acting as the identity. Though the contract has enforced parametricity, it has done so at the expense of non-interference. The unwrapped and wrapped versions of the function behave differently, and no blame error is raised.

Definition

```
declare function special<X>(x : X): X;
```

Library

```
function special(x) {
  if (typeof x === "number") return x + 1;
  else return x;
}
```

Figure 5.12: Proxy Type Comparison

5.4 Related Work

5.4.1 Gradual Typing and TypeScript

Swamy et al. (2014) develop a gradually-typed core of JavaScript called TS* with a focus on security, viewing attacks as type errors. Their language provides memory isolation for typed objects, even in the presence of malicious attacks from untyped contexts. Type safety is guaranteed even under flagrant use of *eval*, stack walks, and other object manipulations. Their approach uses run-time type information (RTTI) to tag values, ensuring dynamic safety. Dynamically typed code is instrumented to perform type checks and tag associated values with their RTTI. The RTTI associated with a value evolves monotonically over time, only becoming more precise. They do not consider blame assignment, but suggest that blame should be assigned to the point of tagging. TS* does not satisfy the *gradual guarantee* of Siek et al. (2015b). The gradual guarantee provides two key properties. First, removing type annotations from a well-typed program should produce a well-typed program. Second, when adding type annotations to a program, if the program remains well-typed then the only change in behaviour can be the addition of trapped errors. TS* violates the guarantee because the operation to set RTTI on a value is conservative. Coercing between function types that are not positive subtypes will fail, even if the function is never applied.

Rastogi et al. (2015) present a compiler for TypeScript that employs sound gradual typing with a greater focus on performance, called Safe TypeScript. Their technique also uses RTTI but establishes a novel notion of *differential subtyping*, calculating the minimum amount of information that a value must be tagged with. Another feature

of their approach is tag erasure: a typed value in a fully typed context will have the obsolete type tag removed. Traditional gradual typing relates the dynamic type any to all types, however Rastogi et al. (2015) do not relate any to erased types. This important feature provides a form of abstraction; an erased type may not be viewed at type any and subsequently manipulated using dynamic access operations. Safe TypeScript and TPD differ in how they treat certain types. The former treats classes nominally while TPD adheres to the approach taken by TypeScript, treating classes structurally. Safe TypeScript does not permit casts from generic types to the dynamic type because this would require sealing. TPD does not maintain such a restriction.

Richards et al. (2015) develop a variant of TypeScript called StrongScript that supports hardening via *concrete types*. StrongScript supports three programming flavours: untyped that has no annotations and performs no static or dynamic checking; optional that supports type annotations which are statically checked, but not guaranteed at run-time; and concrete types that are statically checked and retained at run-time for optimisations and correctness. They distinguish an important property called *trace preservation* which is similar to our notion of non-interference, except that it precludes throwing an error as trace preserving. Adding optional types to an untyped program should not change the behaviour of a program. A parallel would be the guarantee that adding a TypeScript definition to a library should not change the behaviour of a library. They also state the *strengthening* property. Transforming all optional types to concrete types in a cast-free program should be trace preserving. Richards et al. (2015) evaluate their use of concrete types against a small set of benchmarks, reporting speedups of 22% when executed on an optimised JavaScript run-time.

5.4.2 Contract Implementations

Guha et al. (2007) present implementations of parametric polymorphic contracts for PLT Scheme and JavaScript. The PLT Scheme implementation makes use of *generative structs* to construct opaque seals with fresh types. There is no need to associate each seal with a name to distinguish values sealed under different names; a type predicate for the struct is used instead. Furthermore, there is no need to wrap a value in a coffer prior to sealing to obscure the type of the value. The JavaScript implementation does not have the luxury of structs, and also precedes the development of JavaScript proxies. Seals are implemented using basic JavaScript objects that hold the sealed value

and the associated name, therefore they are vulnerable to malicious code modifying the properties of the seal. Both implementations handle blame, but only at the point of unsealing; attempting to unseal a value sealed under a different name will raise positive blame. There is no treatment of blame when seals are tampered with; the implementation solely relies on seals appearing opaque and behaving uniformly.

Disney (2015) designs a macro system for JavaScript that is used to build `contracts.js`, a contract library based on proxies. The library supports parametric polymorphic contracts and the implementation of seals applies proxies to *coffer* objects. Blame is given a full treatment: traps on the proxy will raise negative blame; incorrectly unsealing a value will raise positive blame. Disney (2015) also observes the problem posed by proxies being unable trap type tests, suggesting an extended proxy interface for *virtual values* (Austin et al., 2011).

Guha et al. (2007) and Disney (2015) both support contract *inference* for type parameter instantiations. The former provides this as a convenience to the programmer because the implementation of polymorphic contracts requires explicit instantiation. The latter provides this as an aid to detect cases where heterogeneous lists are passed to functions that act on polymorphic lists, suggesting that this indicates a fault rather than correct code. Ahmed et al. (2017) show that inference is unnecessary and each polymorphic contract can be instantiated with the dynamic type `any`: TPD follows this approach. There is also justification for believing that contract inference is unsuitable for languages like TypeScript that support union types, the dynamic type `any`, and the top type `unknown`. Distinguishing an array that is incorrectly heterogeneous, rather than correctly homogeneous with a fixed union type, is not predictable and can potentially lead to confusing false-positives.

Strickland et al. (2012) design chaperones and impersonators: mechanisms for behavioural interposition similar to proxies, and well suited for implementing contracts. A chaperone can wrap a value performing delayed checks, but the only behaviour a check can add is to raise an error. An impersonator can replace a value, behaving differently, or appearing opaque. Chaperones and impersonators are opaque and can be distinguished from a value they wrap using the `eq?` operator, however both are transparent with respect to the more commonly used `equal?` operator. Chaperones are better suited for wrapping immutable data than their JavaScript proxy counterpart. A proxy wrapping an immutable object must return exactly the same underlying value when trapping a property access, preventing recursive wrapping for higher-order contracts. A chaperone is permitted to wrap the underlying value in another

chaperone, enforcing the higher-order contract.

Keil and Thiemann (2015b) implement higher-order contracts for JavaScript using proxies. The implementation supports function and object contracts, intersection and union contracts, and also user-defined contracts; the implementation does not support polymorphic contracts. They study non-interference, in particular focusing on user-defined contracts that may induce side-effects when applied, violating non-interference. Keil and Thiemann (2015b) apply user-defined contracts in a *sandbox* that monitors the execution environment for modifications to the program state. When a contract tries to mutate an object outside the scope of the contract then an error is raised, enforcing non-interference.

Feltey et al. (2018) design and implement *collapsible contracts*, a novel concept aimed at addressing a significant source of performance degradation associated with the use of contracts. Typically, when a value crosses a typed-untyped boundary at a higher-order type a wrapper is applied; repeated crossings causes redundant wrappers to accumulate, affecting performance. To address the degradation they implement collapsible contracts: trees of contracts where branches split client and server obligations and leaves are sequences of flat contracts. When applying a contract to an already wrapped value the two contracts are combined using a merge operation. The merge retains the behaviour of the two contracts while pruning redundant flat checks using a partial order. Feltey et al. (2018) implement their design in Racket (Flatt and PLT, 2010) and Typed Racket (Tobin-Hochstadt and Felleisen, 2008), showing that the technique can significantly improve performance for some cases. Collapsible contracts do incur additional maintenance at run-time. A worst case overhead of 14% is recorded, which they argue can be softened by dynamically transforming traditional contracts to collapsible contracts when a sufficient number of wrappers accumulate.

Chapter 6

Mixed Messages: An Evaluation of Sound Gradual Typing

Proponents of gradual typing argue that providing programmers with a spectrum between safety and dynamism will improve software as a developer is able to choose the best approach for the task. While the vision for gradual typing is definite, the path from theory to practice is less known.

Using our tool TPD we evaluate the application of sound gradual typing against the DefinitelyTyped repository, the primary collection of TypeScript definitions for JavaScript libraries. The evaluation uses our tool to monitor JavaScript libraries and their test code for conformance to the corresponding definition file. Through our evaluation we measure the benefit sound gradual typing can provide as well as considering the impact of issues known to exist in theory, but not necessarily in practice.

Our results yield mixed messages. We show that contracts and sound gradual typing can be used to enforce conformance of JavaScript libraries and TypeScript clients to a definition file, detecting a significant number of violations. However, our evaluation also emphasises technical concerns associated with implementing contracts in JavaScript using proxies. We show that violations of non-interference caused by contracts are a real problem in practice.

6.1 Evaluating Sound Gradual Typing

We performed an evaluation of sound gradual typing against the DefinitelyTyped repository. We applied TPD to the JavaScript libraries in DefinitelyTyped that run on node.js and had a set of unit tests that all passed. When we conducted this evaluation

there were 500 libraries in DefinitelyTyped, of which 122 satisfied our constraints. The evaluation addressed two concerns:

> *Is sound gradual typing justified?* Are there sufficient violations of conformance to justify the use of sound gradual typing with JavaScript libraries and TypeScript clients?
>
> *Is sound gradual typing plausible?* Are the known technical concerns associated with JavaScript proxies—where contracts violate non-interference—a significant problem to practitioners of sound gradual typing?

We recorded the frequency of libraries that failed to conform to their TypeScript definition, classifying the reason for the failure. From the 122 libraries we tested TPD detected violations in 62 libraries with a total of 179 distinct conformance errors.

We recorded the frequency of proxy interference introduced through the use of TPD. Our evaluation considers all kinds of proxy interference. From the 122 libraries we tested there were 22 violations of non-interference caused by TPD. Twelve libraries exhibited interference due to proxy identity, five due to run-time sealing, four due to reflection, and two due to issues in the proxy implementation.

6.1.1 Overview

Section 6.2 begins with an example violation of conformance, obtained through our evaluation using TPD. Section 6.3 demonstrates contracts violating non-interference, also obtained through our evaluation using TPD. Section 6.4 presents the results of our evaluation. Section 6.5 evaluates proposed solutions to contract interference in the context of our results. Section 6.6 discusses related work regarding evaluation of sound gradual typing and JavaScript libraries, in addition to alternate designs for sound gradual typing.

6.2 Violating Conformance

We present an example of a library failing to conform to the definition. The example illustrates higher-order positive blame where a library incorrectly uses a function argument; the example was found by applying TPD and running the unit tests for the library. We refer to blame in the technical sense. Whether the code is wrong, or the type annotation is wrong, depends on the domain. In general we view library

code to have higher authority than definition files. The Definition file is taken from the DefinitelyTyped repository; library and client code is taken from the corresponding JavaScript library. When presenting the example we omit some blank lines and comments. We write "· · ·" to indicate code we have chosen to elide.

This example is taken from the library `swiz`. The `swiz` library provides serialisation and validation for XML and JSON interfaces. The definition[1] was written by Goddard[2], and the library[3] and client[4] were written by Rackspace[5].

Definition Figure 6.1 contains the definition file. The definition file exports class `Valve` with overloaded method `check`. The first overload accepts three arguments: an object to check of any type, a configuration object, and a callback that accepts two arguments of any type. The callback and the `check` method both return `void`. The second overload accepts two arguments: an object to check of any type, and a callback that accepts two arguments of any type. The callback and the `check` method both return `void`.

Library Figure 6.1 contains the library implementation file. We omit initialisation at the beginning of the check method, the purpose of which is to provide a value for `options` if the argument is not provided by the caller. The method applies the `checkSchema` function to the object that requires validation. The `checkSchema` function takes a callback that performs validation and forwards the result to the `callback` passed as argument to `check`.

Client Figure 6.2 contains the client test file that evoked the violation of conformance. The test file imports the `Valve` class on line 3. A test function is defined on line 12 that attempts to validate a bad value that omits a key; the test case checks that the returned error contains the correct information.

Wrapped Behaviour When the test case in Figure 6.2 is executed TPD will assign positive blame to the library. The diagnosis begins with the application of the callback

[1] `https://github.com/DefinitelyTyped/DefinitelyTyped/blob/03f12b0f667d29fe8e15c2e5c56f3ed7e10c8eb9/swiz/swiz.d.ts`

[2] Jeff Goddard: `https://github.com/jedigo`

[3] `https://github.com/racker/node-swiz/blob/4ee9b36a620e7365003d526f195bcf76e05a863a/lib/valve.js`

[4] `https://github.com/racker/node-swiz/blob/4ee9b36a620e7365003d526f195bcf76e05a863a/tests/test-valve.js`

[5] Rackspace: `https://developer.rackspace.com`

Definition `swiz.d.ts`

```
1  ...
2  export class Valve {
3      ...
4      check(obj: any, options: ICheckOptions, callback: (err:
           any, cleaned: any) => void): void;
5      check(obj: any, callback: (err: any, cleaned: any) =>
          void): void;
6      ...
7  }
8  ...
```

Library `valve.js`

```
1  ...
2  Valve.prototype.check = function(_obj, options, callback) {
3    ...
4    checkSchema(obj, this.schema, [], false, this.baton,
       function(err, cleaned) {
5      if (err) {
6        callback(err);
7        return;
8      }
9      if (finalValidator) {
10       finalValidator(cleaned, function(err, finalCleaned) {
11         if (err instanceof Error) {
12           throw new Error('err argument must be a swiz
                error object');
13         }
14         callback(err, finalCleaned);
15       });
16     }
17     else {
18       callback(err, cleaned);
19     }
20   });
21 };
22 ...
```

Figure 6.1: Violating Conformance Example (Definition and Library)

Client test-valve.js

```
1  var swiz = require('../lib/swiz');
2  var async = require('async');
3  var V = swiz.Valve;
4   ...
5  var badExampleNode1 = {
6    'id' : 'xkCD366',
7    'is_active' : true,
8    'name' : 'exmample',
9    'ipaddress' : '42.24.42.24'
10 };
11  ...
12 exports['test_schema_translation_2'] = function(test,
       assert) {

13   var validity = swiz.defToValve(def),
14       v = new V(validity.Node);
15   assert.isDefined(validity.Node);
16   assert.isDefined(validity.NodeOpts);
17
18   v.check(badExampleNode1, function(err, cleaned) {
19     assert.deepEqual(err.message, 'Missing required key (
           agent_name)',
20       'schama translation failure (missing agent_key)');
21     test.finish();
22   });
23 };
24  ...
```

Figure 6.2: Violating Conformance Example (Client)

function at line 6 in the library:

```
callback(err);
```

The validation of the object fails and the callback passed to `check` is called with a single argument, however in both overloads the callback expects two arguments—a violation of conformance. The callback function supplied to `check` comes from the client, and is wrapped in a function contract annotated with a negative blame node. When the wrapped callback function is applied with an incorrect number of arguments the function contract assigns negative blame to the context. Recall that the contract was already annotated with a negative blame node, therefore this will assign *positive* blame to the library. The library has incorrectly used the callback function by only applying the function to one argument. The semantics of JavaScript will assign the value `undefined` to any argument that is omitted at the call site to a function. There is some justification for allowing an argument of type `any` to be omitted because the default value of `undefined` will be used instead, respecting the type `any`. TypeScript does not follow this reasoning and raises a type error based on the arity mismatch alone, and we implement our contract monitors to behave in the same way.

6.3 Violating Non-Interference

We present examples of proxy interference caused by the application of sound gradual typing to JavaScript libraries. The examples illustrate the two fundamental ways interference occurs: by changing object identity through wrapping and by changing type through sealing. Both examples were found by applying TPD and running the unit tests for each library. Definitions are taken from the DefinitelyTyped repository; library and client code is taken from the corresponding JavaScript library. When presenting the examples we omit some blank lines and comments. We write "$\cdots$" to indicate code we have chosen to elide.

6.3.1 Proxy Identity

This example is taken from the library `gulp-if`. The `gulp` library is a streaming build system for JavaScript applications and the `gulp-if` library is a plugin that implements

Definition gulp-if.d.ts

```
declare module "gulp-if" {
    function gulpIf(
        condition: boolean,
        stream: NodeJS.ReadWriteStream,
        elseStream?: NodeJS.ReadWriteStream): NodeJS.
           ReadWriteStream;
    export = gulpIf;
}
```

Library index.js

```
'use strict';

var match = require('gulp-match');
var ternaryStream = require('ternary-stream');
var through2 = require('through2');

module.exports = function (condition, trueChild, falseChild
    , minimatchOptions) {
  if (!trueChild) {
    throw new Error('gulp-if: child action is required');
  }

  if (typeof condition === 'boolean') {
    // no need to evaluate the condition for each file
    // other benefit is it never loads the other stream
    return condition ? trueChild : (falseChild || through2.
       obj());
  }

  function classifier (file) {
    return !!match(file, condition, minimatchOptions);
  }

  return ternaryStream(classifier, trueChild, falseChild);
};
```

Figure 6.3: Proxy Identity Example (Definition and Library)

Client boolean.js

```
1  ...
2  describe('when given a boolean,', function() {
3      var tempFile = './temp.txt';
4      var tempFileContent = 'A test generated this file and
           it is safe to delete';
5      it('should call the function when passed truthy',
           function(done) {
6          // Arrange
7          var condition = true;
8          var called = 0;
9          var fakeFile = {
10             path: tempFile,
11             contents: new Buffer(tempFileContent)
12         };
13         var s = gulpif(condition, through.obj(function (
               file, enc, cb) {

14             // Test that file got passed through
15             (file === fakeFile).should.equal(true);
16
17             called++;
18             this.push(file);
19             cb();
20         }));
21         // Assert
22         s.once('finish', function(){
23
24             // Test that command executed
25             called.should.equal(1);
26             done();
27         });
28         // Act
29         s.write(fakeFile);
30         s.end();
31     });
32      ...
33 }
```

Figure 6.4: Proxy Identity Example (Client)

a conditional operator for streams. The definition[6] was written by Asana[7], and the library[8] and client[9] were written by Richardson[10].

Definition Figure 6.3 contains the definition file. The definition describes a module that exports a single function using an export assignment. The function accepts three arguments: a boolean condition, a `NodeJS.ReadWriteStream` object that denotes the *true* stream, and an optional `NodeJS.ReadWriteStream` object that denotes the *false* stream. An optional argument may be omitted when calling the function; JavaScript fills missing arguments with the `undefined` value at run-time. The function returns a stream of type `NodeJS.ReadWriteStream`.

Library Figure 6.3 contains the library implementation file. The library defines the primary function that accepts a `condition` argument and returns the argument `trueChild` when the condition evaluates to true. If the condition evaluates to false then argument `falseChild` is returned when supplied, otherwise a default stream value is used. We omit discussion of the behaviour when the function in called with a non-boolean condition because this functionality is not exposed in the type of the definition file. When applying TPD we observed that test code for this library exercises the additional functionally and TPD would correspondingly signal a negative blame violation. The violation indicates that the definition file is incomplete with respect to the implementation of the library. We use *incomplete* informally as it is not clear what a *complete* definition file means. In this setting we intend for *incomplete* to mean that a definition file is too strict and rejects programs that the library authors intend to be accepted, evidenced by unit tests that induce negative blame.

Client Figure 6.4 contains the client test file that evoked the interference. The functions `describe` and `it` are provided by a unit-test library. The former describes a set of tests and the latter describes a particular test case. Lines 7-12 initialise the test parameters: `condition` is the branch for the stream ternary, `called` is a counter used to certify a callback was evaluated, and `fakeFile` is an object to pass through the

[6] `https://github.com/DefinitelyTyped/DefinitelyTyped/blob/03f12b0f667d29fe8e15c2e5c56f3ed7e10c8eb9/gulp-if/gulp-if.d.ts`

[7] Asana: `https://asana.com`

[8] `https://github.com/robrich/gulp-if/blob/54550add63670d801d7776486512dbb6e46147c7/index.js`

[9] `https://github.com/robrich/gulp-if/blob/54550add63670d801d7776486512dbb6e46147c7/test/boolean.js`

[10] Rob Richardson: `https://github.com/robrich/gulp-if`

stream. Lines 13-20 define the new stream using the ternary. The stream to be selected when the condition is true asserts that the piped object is equal to `fakeFile`, signals that the callback was evaluated by incrementing `counter`, and passes the file through the stream. Line 29 commences the test by writing `fakeFile` to the stream.

Unwrapped Behaviour The assertion on line 15 will pass as the `fakeFile` is written to the stream, then piped to the stream in the true branch of the ternary. The assertion on line 25 will pass as the callback associated with the true stream is executed, setting the counter to one.

Wrapped Behaviour The assertion on line 15 will fail as the file piped through the stream is not equal to the expected `fakeFile`. The diagnosis begins with the initial step on line 29 that writes the file to the stream. The object `s` is the result of applying the library function `gulpif` and is therefore wrapped in a contract for type `NodeJS.ReadWriteStream`. The type `NodeJS.ReadWriteStream` has a method `write` of type:

```
write(buffer: Buffer | string, cb?: Function): boolean;
```

The application of this method will be wrapped by the contract and therefore the `buffer` argument will be wrapped in a contract for type `Buffer`—a proxy. As a consequence the argument `fakeFile` will be wrapped in a proxy before being passed through the stream; line 15 will compare a wrapped and unwrapped version of the same object, returning false and failing the assertion.

6.3.2 Dynamic Sealing

This example is taken from the library `clone`. The `clone` library provides deep cloning of JavaScript objects and primitives. The definition[11] was written by Simpson[12], and the library[13] and client[14] were written by Vorbach[15].

[11] `https://github.com/DefinitelyTyped/DefinitelyTyped/blob/2f47c75835b837777a85287611703d683b0aaa83/clone/clone.d.ts`

[12] Kieran Simpson: `https://github.com/kierans/DefinitelyTyped`

[13] `https://github.com/pvorb/clone/blob/2d907392855214439ee9f9b6c60b3e47c5fae07b/clone.js`

[14] `https://github.com/pvorb/clone/blob/2d907392855214439ee9f9b6c60b3e47c5fae07b/test.js`

[15] Paul Vorbach: `https://github.com/pvorb/clone`

Definition `clone.d.ts`

```
declare module "clone" {
    function clone<T>(val: T, circular?: boolean, depth?:
        number): T;
    module clone {
        function clonePrototype<T>(obj: T): T;
    }
    export = clone
}
```

Library `clone.js`

```
function clone(parent, circular, depth, prototype,
    includeNonEnumerable) {
    if (typeof circular === 'object') {
        ...
    }
    ...
    if (typeof circular == 'undefined')
        circular = true;
    if (typeof depth == 'undefined')
        depth = Infinity;
    // recurse this function so we don't reset allParents
        and allChildren
    function _clone(parent, depth) {
        // cloning null always returns null
        if (parent === null)
            return null;
        if (depth === 0)
            return parent;
        ...
        if (typeof parent != 'object') {
            return parent;
        }
        ...
    }
    return _clone(parent, depth);
}
```

Figure 6.5: Dynamic Sealing Example (Definition and Library)

Client `test.js`

```
1  ...
2 exports["clone number"] = function (test) {
3   test.expect(5); // how many tests?
4
5   var a = 0;
6   test.strictEqual(clone(a), a);
7   a = 1;
8   test.strictEqual(clone(a), a);
9   a = -1000;
10  test.strictEqual(clone(a), a);
11  a = 3.1415927;
12  test.strictEqual(clone(a), a);
13  a = -3.1415927;
14  test.strictEqual(clone(a), a);
15
16  test.done();
17 };
18  ...
```

Figure 6.6: Dynamic Sealing Example (Client)

Definition Figure 6.5 contains the definition file. The definition file exports identifier `clone`, the type of which is constructed by a function signature and module definition. JavaScript permits functions to act like objects and have properties. A mechanism to encode this in a definition file is to specify a function and module of the same name, the types of both will then be merged. The function type specifies that the library exports a generic function that accepts three arguments: an argument of generic type `T` to be cloned, an optional boolean, and an optional number. The function returns a value of the same generic type. The module type specifies that the library exports a property `clonePrototype` that is of function type.

Library Figure 6.5 contains the library implementation file. Lines 2-9 are responsible for initialisation; the majority of the work is done in the inner-recursive function `_clone` defined on line 11 and applied on line 23. Cloning a null pointer will return a null pointer. Exhausting the depth limit will return the current clone target immediately. Cloning a primitive value—indicated by having a non-object type—will return the same value. The definition of identity is vacuous for primitive values so the clone target can be immediately returned.

Client Figure 6.6 contains the client test file that evoked the interference. The code describes a set of tests that evaluate the cloning function on numbers. There are five expected tests and each test checks that cloning a number is equal to itself. The test cases are contained in a function that takes as argument a particular testing API that exports three functions: `expect`, a function to indicate the expected number of cases; `strictEqual`, a generalised (deep) equality function; and `done`, a function to indicate that all cases have been evaluated.

Unwrapped Behaviour Each `strictEqual` application will evaluate to true because the clone function immediately returns the unmodified argument.

Wrapped Behaviour Each `strictEqual` application will evaluate to false because the wrapped clone function no longer returns the original argument. The diagnosis begins by examining the type of the `clone`: a generic function. When the wrapped function is applied to a number the argument is sealed because the contract expects an argument of generic type. The coffer used by the seal causes the type test on line 18 to return false, when the test returned true in unwrapped code—a violation of non-interference. The failing type test causes the clone function to proceed to treat

the seal like an object and clone the seal, returning a new object with a fresh identity. When the cloned seal is returned from the function the contract will attempt to unseal the value, however this will fail. Recall that unsealing recognises seals by their identity, but because the value is not a seal, rather a *clone* of a seal, the value will not be recognised. TPD chooses to log blame errors rather than throw an exception for convenience in automated testing. As a consequence, execution continues and the cloned seal is returned, however the cloned seal is not equal to the original argument and the equality test fails. Throwing an exception is not enough to restore non-interference because the seal has already altered the behaviour of the program.

Assigning a parametric type to a clone function seems nonsensical because any clone function must perform type analysis, but whether a type is sensible—or not—should have no influence on the contract's ability to enforce the specification. Although the example uses a type definition that is clearly wrong we still consider the example useful in highlighting limitations in the proxy design.

6.4 Results

This section presents our results from applying sound gradual typing to the DefinitelyTyped repository using our tool *The Prime Directive*. The evaluation can be obtained in the form of a software artifact (Williams et al., 2017b).

6.4.1 Method

The DefinitelyTyped repository was used as a corpus of JavaScript libraries with a type specification of their API, available as a TypeScript definition file. There were three requirements for a library to be included in the evaluation. First, the library had to be packaged as a node.js module and executable on the node.js JavaScript runtime. This requirement made it easier to automatically install, wrap, and test libraries as they conformed to a regular package structure. Second, it must be possible to install the library without direct intervention. This requirement made it easier to automatically reproduce the evaluation in different environments. Last, the library had to come with a suite of unit tests that all passed once the library was installed. This requirement meant it was possible to attribute any failing tests as a violation of non-interference. At the time the evaluation was conducted there were around 500 libraries in DefinitelyTyped, with 122 libraries meeting our criteria.

	Blame	
Error Kind	Library (Positive)	Client (Negative)
Base Type	47	47
Function Arity	23	43
Void Return Type	14	2
Parametricity	3	0
Distinct Errors	87	92
Distinct Libraries	40	48

Table 6.1: Classification of Failures to Conform.

The evaluation process took each library and added wrapper code generated by TPD to the library file that exported the module. The unit tests of the wrapped library were executed and conformance violations detected by TPD were logged and classified. Additionally, unit tests that subsequently failed after wrapping were recorded as violations of non-interference and classified.

6.4.2 Conformance

Table 6.1 shows the classification of violations where a library or its test code failed to conform to the definition. A library failing to conform corresponds to a TPD contract assigning positive blame, while a client (test code) failing to conform corresponds to a TPD contract assigning negative blame. Each row records the distinct number of errors for a given class, distinguished by error source. From the 122 libraries tested there were 179 distinct errors found in 62 libraries. We informally consider two errors to be distinct if correcting one error does not correct the other.

Base Type A base type violation is where a value fails a first-order check associated with a contract. This includes producing a string where a number is specified, or producing an object where a function is specified.

Function Arity An arity violation is where a function application supplies too many or too few arguments. We conjecture that the skew in frequency between library and client is due to first-order functions being more prevalent than higher-

order functions; an arity error on a first-order function would assign negative blame. In general, arity errors were very frequent which may be due definition authors misunderstanding Typescript semantics. For example, callbacks for asynchronous functions were often ascribed the type `(err: any, val: any) => void`. When the asynchronous function succeeded, `err` would be supplied as a "falsy" value and `val` would be supplied as the result. When the asynchronous function failed, `err` would be supplied as an error object and `val` would be omitted. This would be a violation as TypeScript expects the callback to always be called with two arguments, even if the arguments have dynamic type `any`. The correct type of the callback is one that marks the second argument as optional: `(err: any, val?: any) => void`.

Void Return Type A void return type violation is a distinguished base type violation where a function was expected to return nothing, or specifically the `undefined` value, but instead returned some other value. This class of error captured the common mistake of incorrectly ascribing a synchronous function as asynchronous. A synchronous function returns its result; an asynchronous function returns nothing, but accepts a continuation or callback.

Parametricity A parametricity violation is where a function fails to conform to a polymorphic type. A violation can arise in three ways: a polymorphic function can tamper with a seal, a polymorphic function can attempt to unseal a value sealed using a different name, or a polymorphic function can attempt to unseal a value that is not a seal. This class of violation was comparatively infrequent. The cause of this may be due to definition authors being unfamiliar with generics, favouring `any` instead. Additionally, applying type tests to seals which *should* be a violation of parametricity cannot be reported because proxies do not intercept `typeof`.

6.4.3 Non-Interference

Table 6.2 shows the classification of violations of non-interference. Each row records the number of libraries that exhibited that form of interference. From the 122 libraries tested there were 23 violations of non-interference found in 22 libraries (one library exhibited two forms of interference).

Proxy Identity There were twelve libraries that exhibited interference because equality tests that returned true in unwrapped code now returned false in wrapped

Cause of Interference		
Proxy Identity	TI (proxy-object)	7
	TII (proxy-proxy)	5
Sealing		5
Reflection		4
Proxy Implementation		2
Distinct Libraries		22

Table 6.2: Classification of Interference.

code. The classification distinguishes proxy-object and proxy-proxy equality tests in a similar manner to Keil et al. (2015). In seven cases the failing equality test was between an object and a proxy for that object (TI). In five cases the failing equality test was between two proxies for the same object (TII). Section 6.5.1 discusses the use of identity preserving membranes as a potential solution to the issue of proxy identity. Distinguishing the type of equality test—TI or TII—is helpful when evaluating the suitability of membranes because the different forms of equality test pose different challenges.

Sealing There were five libraries that exhibited interference because of dynamic sealing. Sealing may introduce interference in two ways: either by changing the type of an object or by changing its identity. The former arises because proxies are unable to trap type tests so to enforce parametricity all seals must have the same type. The latter arises because proxies that implement seals are opaque. We distinguish identity interference caused by seals and identity interference caused by function or object contracts because they have different solutions. For general contracts a transparent proxy provides a complete solution, whereas transparent proxies are insufficient for seals. A proxy cannot intercept equality operations to raise blame so a naive transparent proxy will not enforce parametricity when applying equality tests to seals.

Reflection There were four libraries that exhibited interference because of reflection: where a library inspected its own code in some capacity. Two distinct forms of reflection interference were caused by the use of TPD. First, some libraries employed linting to ensure code meets an expected style and the TPD wrapper code would fail

to meet that style. Different libraries have different style guidelines so there is no easy remedy; preventing this source of interference in general would require TPD producing wrapper code that is dependent on the particular linter a library uses. Second, some libraries would analyse the library footprint on the JavaScript global name-space. TPD wrapper code requires access to the contract implementation to perform the dynamic checks and consequently introduces new dependencies into the global name-space. The new dependencies would be flagged as unexpected. Both sources of interference are inconvenient, but not fatal. A library maintainer that adds TPD to their project would presumably make accommodations for TPD in the testing process. The interference exhibited here is primarily due to the evaluation process that unilaterally applies TPD to a swathe of libraries.

Proxy Implementation There were two libraries that exhibited interference because of issues in the proxy implementation. At the time this evaluation was performed the proxy implementation was still in an experimental phase and therefore the implementation was not complete. In particular, certain operations that internally performed dynamic type checking did not correctly unwrap proxy objects, throwing dynamic type errors instead.

6.4.4 Summary of Results

We argue that there are two contributions to be taken from the results of our evaluation. First, TypeScript definition files are prone to error and contracts can be used to detect violations of conformance. When a violation occurs it could be viewed that the library is implemented or tested incorrectly, alternatively, the definition could be incorrectly specified—we assume the latter. Only those libraries that passed all their unit-tests were evaluated; the library was correct according to implicit specification of the tests which we view to have higher authority than definition files. Feldthaus and Møller (2014) also evaluate the correctness of definition files; our results support and extend theirs. They consider the ten largest definition files in DefinitelyTyped while our sample includes very small definition files. We show that even simple definitions are incorrectly specified.

The second contribution of our results is evidence that interference caused by opaque proxies is a significant problem for contract implementations in JavaScript. Issues associated with proxy identity were known in theory and Keil et al. (2015) demonstrated that the problem can manifest in benchmark code. We show that using

proxies to implement sound gradual typing violates non-interference in a significant number of cases. Additionally we measure a new source of interference caused by the use of dynamic sealing. Interference caused by sealing similarly poses a threat to the applicability of sound gradual typing.

6.4.5 Threats to Validity

We discuss four threats to the validity of the presented results.

Completeness The approach adopted in the evaluation to record violations of conformance is incomplete. A library may violate conformance but TPD would not raise a violation because there was no unit test that exercised the appropriate contract. As a consequence, the frequency of violations presented in Table 6.1 is an under-approximation. Similarly, the approach adopted to record violations of non-interference is also incomplete. Proxies used by TPD could violate non-interference in a way that does not cause a unit test to fail, therefore going undetected. As a consequence, the frequency of violations presented in Table 6.2 is also an under-approximation. We consider the primary contribution of this evaluation to be demonstrating that violations of conformance and non-interference are significant in practice: providing an under-approximation does not diminish this contribution.

Sample Bias The evaluation only applies TPD to a sample of libraries from DefinitelyTyped therefore it is possible that the sample is not representative of the entire population. We argue that the criteria for selecting libraries does not favour libraries that exhibit a particular behaviour when TPD is applied.

Esoteric Tests The "client" code used in the evaluation came from the library's suite of unit tests. There is the possibility that unit test code is not representative of real client code, and therefore the results may not characterise experience in practice. This threat is most applicable to the results regarding violations of non-interference; unit-test code could contain a higher number of equality tests than real code, therefore problems regarding proxy identity would be disproportionately high. Running unit-tests is an important part of software maintenance, so even in the event that interference occurs excessively in unit-tests, we still consider this a significant observation. The threat is less significant when considering violations of conformance because a violation of the definition is a problem regardless of where it was detected.

Test Coverage The evaluation crucially relies on unit-tests to exercise wrapped code; when the unit-tests have a low coverage TPD may miss latent violations of conformance or non-interference. We gauge the affect of this threat similarly to the threat of incompleteness. Missing violations does not diminish the results because we argue that the lower bound we record is sufficiently large to be of interest. Detecting additional violations would strengthen our results, rather than weaken them. Mezzetti et al. (2018) use unit-tests from JavaScript libraries to accurately classify version changes. Their study suggests that unit-tests for JavaScript libraries typically have sufficient coverage to faithfully reflect the library interface.

6.5 Solutions to Contract Interference

This section discusses design alternatives intented to ameliorate the problem of interference caused by JavaScript proxies that implement contracts.

6.5.1 Membranes

An identity preserving membrane (Miller, 2006; Van Cutsem and Miller, 2010, 2013) is a distinguished boundary between library and client (also described as *wet* and *dry*). A membrane guarantees two properties. First, an object crossing the boundary at two locations is wrapped with the same proxy. Second, an object that crosses the boundary and then back through is unwrapped rather than being wrapped twice. Identity preserving membranes are *transitive* such that accessing a property on a wrapped object will extend the membrane around the accessed value. The composition of these properties ensures that given two objects x and y that share identity on the *wet* side, then their corresponding representations on the *dry* side will also share identity. This addresses issues associated with proxy-proxy (TII) equality tests. Furthermore, there are never direct comparisons between *wet* and *dry* objects. This addresses issues associated with proxy-object (TI) equality tests.

Designing contracts to implement identity preserving membranes is presented as a solution to the problem of proxy identity, however we identify three challenges associated with combining contracts and identity preserving membranes.

Implementing any The contract for the dynamic type, or `any`, is typically presented in the literature as the identity function: the contract performs no checking

and values pass through immediately. A vital property of identity preserving membranes is that they are transitive to preserve the continuity of the membrane.

```
1 const wetVal = { a: 42 };
2 const wetBox = { value: wetVal };
3 const dryBox = membrane(wetBox);
4 const dryVal = dryBox.value;
```

A membrane is applied to `wetBox` returning a dry reference; accessing the property `value` should extend the membrane around the resulting object. Suppose that `membrane` was a contract for the type:

```
1 interface Box { value: any }
```

Accessing the property `value` through contract `any` will act as the identity function, returning a direct reference to `wetVal` on the dry side: the `any` contract acts like a puncture in the membrane. The consequence of this is that it is possible to have comparisons between wet and dry objects—between unwrapped and wrapped objects—resulting in TI proxy identity interference.

The solution is to implement the `any` contract such that it wraps the value, performing no type checking, but extending the membrane. This is possible but at the cost of additional complexity and with increased run-time overhead.

Contract Merging To resolve the problem of proxy-proxy comparisons (TII) an identity preserving membrane will ensure that the same object is wrapped with the same proxy when crossing a boundary.

```
1 const wetFunction = (f, g) =>
2   (f === g) || (f(1) > 0) || g(true);
3 const dryFunction = membrane(wetFunction);
4 const dryVal = x =>
5   typeof x === "number" ? 42 : !x;
6 const result = dryFunction(dryVal, dryVal);
```

The object `dryVal` crosses the boundary from dry to wet when passed as both arguments to `dryFunction`, and is correspondingly wrapped. To ensure the equality test in the body of `wetFunction` evaluates correctly, both occurrences of `dryVal` must be wrapped in the same proxy.

Suppose that `membrane` was a contract for the type:

```
1 (
2   f: (x: number)  => number,
3   g: (x: boolean) => boolean
4 ) => boolean
```

A basic contract implementation would wrap each argument to `dryFunction` in a separate function wrapper, or proxy. However, when implementing an identity preserving membrane both arguments correspond to the same reference, therefore the behaviour of the two function wrappers must be merged into one contract proxy. Unfortunately there is no correct way to combine the two contracts for `f` and `g` while preserving the original semantics. A simple combination using the *and* contract combinator would be too strict: the new contract would demand that the argument to each application satisfy types number *and* boolean. A combination using an intersection type would be too permissive: the new contract would allow `f` to be called with a boolean while the original type only accepts numbers.

Parametricity When an object crosses a boundary, and then crosses back, the object is unwrapped rather than being wrapped twice.

```
1 const wetFunction = (f, x) => f(x);
2 const dryFunction = membrane(wetFunction);
3 const val = { a: 42 };
4 const fn = x => x === val ? val : x;
5 const result = dryFunction(fn, val);
```

Application of `dryFunction` wraps both arguments because they cross the membrane. Evaluating the body of `wetFunction` involves applying wrapped function `fn` to wrapped argument `val`, and in the process, argument `val` crosses the boundary for a second time. At the second crossing `val` is unwrapped so that the body of `fn` only compares two unwrapped references to `val`, ensuing the equality test evaluates correctly. Suppose that the membrane was a contract for the type:

```
1 (
2   f: <X>(x: X) => X,
3   x: { a: number }
4 ) => any
```

Application of `dryFunction` wraps both arguments in contracts for their corresponding type. Evaluating the body of `wetFunction` involves applying wrapped function

```
const prim_get = Reflect.get; // Original API
const types  = new WeakMap();

function wrapObj(obj, p, type) {
  types.set(obj, { node: p, type: type });
  return obj;
}

// Modified Reflect API
Reflect.get = function(target, property, receiver) {
    const val = prim_get(target, property, receiver);
    if (types.has(target)) {
        const p = types.get(target).node;
        const type = types.get(target).type;
        return wrap(val, p, type[property]);
    }
    return val;
}
```

Figure 6.7: Contracts via Reflection

`fn` to wrapped argument `val`, and in doing so, we face a dilemma. A polymorphic contract demands that wrapped argument `val` is sealed to enforce parametricity. An identity preserving membrane demands that wrapped argument `val` is unwrapped to its original form to ensure that a wet value does not appear in the body of a dry function. Both demands cannot be simultaneously satisfied. Choosing to omit sealing means that it is not possible to correctly detect that `fn` fails to conform to the polymorphic contract; without sealing the argument there is no distinction between a reference to `val` passed as an argument, and a reference originating from the function body. Choosing to omit unwrapping at the double crossing means that the use of contracts introduces interference; the equality test in unwrapped code will evaluate to true, while the equality test in wrapped code will evaluate to false.

6.5.2 Rewriting

A drastic but effective technique is to employ program rewriting: either replacing certain operators with proxy-aware variants, or by replacing proxies with reflective operators.

To address the problem of opaque proxies—those that have a distinct identity and cannot intercept type tests—a proposed solution is to replace equality and type tests with proxy-aware variants. An implementation of proxy-aware operators would require a map from proxies to the objects they wrap, permitting operations to be forwarded to the unwrapped object. This technique would demand source transformation of both library and client, weakening some of the benefits of contracts. Additionally, special care has to be taken in the event of arbitrary code execution through mechanisms like `eval`.

Another rewriting solution is to invert the behavioural modification introduced by contracts. Rather than applying type checking behaviour to each object via a proxy, type checking behaviour is applied at each use of an object through the `Reflect`[16] API. The reflection interface mirrors the proxy interface, exposing the same set of operations. This technique requires rewriting because most code uses default object operations such as `foo.x`, rather than reflective operations such as `Reflect.get(foo, "x")`. Implementing contracts with reflection requires mapping references to contract types and using reflective operations to propagate type information.

Figure 6.7 presents an extract of a contract implementation that uses the reflection API. When an object is wrapped a type is associated with the object, however the original object is returned. When accessing a property via `Reflect.get`, if the object has an associated type then the property is recursively wrapped. A recursive wrapping will either perform a type test in the case of flat contracts, or associate a type in the case of higher-order contracts. The reflection approach does not use proxies and wrapped objects retain their identity, therefore equality tests are unaffected by contracts.

This technique is used by Vitousek et al. (2017) where a blame map associates heap addresses and casts; the weak-map `type` serves as the blame map in our example. There are also similarities between the reflection encoding and gradual typing that employs a first-order embedding (Greenman and Felleisen, 2018). The former dynamically propagates type-information through the reflection API, while the latter

[16] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect`

statically propagates type-information.

There are limitations of the approach that uses reflection. Without proxies there is no way to implement polymorphic contracts that use dynamic sealing. Associating a type to an object is not sufficient to distinguish references that are passed as generic arguments, and references captured in the environment of a function. The reflection approach also suffers from the contract merging problem of identity preserving membranes. Applying multiple contracts to the same object requires merging of the contracts, which does not have a general solution.

6.5.3 Transparent Proxies

Keil et al. (2015) propose transparent proxies as an alternative to opaque proxies. The primary difference is that an equality test applied to a transparent proxy is forwarded to the wrapped object, ensuring that equality tests in wrapped code behave like equality tests in unwrapped code. This design does raise security questions: a transparent proxy could maliciously masquerade as an existing object and a client would be unable to detect the deviant proxy. One solution is to provide an explicit `isProxy` function to expose proxies. However, this solution is not well suited for polymorphic contracts. A proxy predicate is not capable of distinguishing the same object sealed under different names, while using coffers to force a fresh identity would undermine the purpose of using transparent proxies.

Keil et al. (2015) present a nuanced alternative that uses *realms*. Every proxy is associated with a realm responsible for its creation and realms provide an additional equality operator. A proxy is transparent when compared outside the realm, but opaque when compared within. For example:

```
1 const realm = newRealm();
2 const obj = {};
3 const p1 = realm.newProxy(obj, handler);
4 const p2 = realm.newProxy(obj, handler);
5 p1 === p2;              // true
6 realm.equals(p1, p2); // false
```

Realms provide a better solution when implementing polymorphic contracts. In user code (outside the realm) a seal can inherit the identity of the object it wraps, preserving non-interference; when attempting to unseal a value (inside the realm) a proxy seal will be exposed.

6.5.4 Virtual Values

Austin et al. (2011) present the design of *virtual values*, a mechanism to define specialised values that support behavioural modification. A virtual value is like a proxy, supporting traps that implement custom behaviour, though a virtual value can be constructed from a primitive value such as a number or boolean, unlike a proxy.

Virtual values are a natural way to implement dynamic seals for polymorphic contracts. Recall that a primitive value *must* be wrapped in a coffer prior to sealing with a proxy, violating non-interference by changing the type of the value they wrap. Virtual values do not suffer this restriction. Furthermore, the interface of a virtual value is richer than the interface of a proxy, providing the ability to trap more operations such as equality and type tests. By permitting virtual values to trap these operations which violate parametricity, dynamic seals may throw an error, enforcing parametricity without violating non-interference.

6.6 Related Work

6.6.1 Evaluating Gradual Typing

Keil et al. (2015) present the first evaluation of contracts in JavaScript that focuses on non-interference. They modify the canonical set of JavaScript benchmarks to include contract checks implemented using proxies, then they record the frequency of proxy equality comparisons using an instrumented JavaScript run-time. Their results show that proxies can affect equality comparisons in practice, also indicating that identity preserving membranes only address some of the occurrences. The contracts for the benchmarks are created by the authors of the evaluation, and are only applied to selected functions. There is a possibility that their use of contracts is not representative of contracts specified for library interfaces, however our evaluation with TPD suggests that the behaviour is consistent.

Takikawa et al. (2016) perform the first comprehensive evaluation of the performance impact of sound gradual typing. They develop a principled approach to conducting evaluations of sound gradual typing. First, select a suite of fully typed benchmark programs. Second, measure the performance of all gradually typed configurations for the benchmark programs, ranging from fully untyped to fully typed. Finally, record the number of configurations that that do not exceed a specified acceptable overhead. Their results give a poor outlook, with most configurations incur-

ring unacceptable overhead. In particular they demonstrate performance *valleys*: a programmer that adds more type annotations to an already slow program will reduce performance further. Only when the program is full typed is performance regained.

Muehlboeck and Tate (2017) develop a new gradually typed language that aims to avoid the performance loss demonstrated by Takikawa et al. (2016). They specify novel properties *transparency* and *immediate accountability* that they use to inform the design and implementation of a gradual language. Transparency ensures that a successful dynamic type check leaves no trace. Immediate accountability ensures that a failing check can be immediately traced to its source when the check is executed, and is not later blamed in a deferred execution. Their presented language boasts good performance, avoiding the extreme slowdowns exhibited by Takikawa et al. (2016). One component of their successful approach is to sacrifice expressiveness in some cases, using nominal typing rather that structural typing.

Bauman et al. (2017) address the performance issues of gradual typing in a different manner. Instead of reducing the number of wrappers they seek to improve wrapper performance. They develop Pycket, an implementation of Racket that utilises a tracing JIT compiler. A novel implementation of proxies, or impersonators (Strickland et al., 2012), that exploits *hidden classes* is given. A hidden class is a representation of the object layout associated with an object. Code that frequently executes on values of the same hidden class can be turned into efficient machine code that is specialised to the object layout. The modified compiler produces significant increases in performance, reducing some slowdowns by two orders of magnitude. Their approach could be adopted to proxies in JavaScript as most run-times uses hidden classes. One challenge in migrating the technique is that JavaScript objects are mutable, while they exploit immutability. The hidden class of an immutable object will not change at run-time because no extra fields are introduced. The hidden class of a mutable object will change at run-time when a new field is added.

Greenman and Migeed (2018) apply the technique of Takikawa et al. (2016) to Reticulated Python (Vitousek et al., 2017), a gradual typed language based on transient semantics. Their results show that any configuration of untyped-typed code only incurs a slowdown within one order of magnitude, however adding type annotations increases overhead at a linear rate. Consequently, fully typed code experiences the largest slowdown.

6.6.2 Evaluating JavaScript Libraries

Heidegger and Thiemann (2010) implement contracts for JavaScript libraries, applying the contracts through instrumentation that analyses contract signatures in code comments. The contract signatures are additionally used to perform random testing, where refinements can be included in contract signatures to guide test generation. For example, generation of test values for object contracts can be guided to use properties referenced in the body of a function, drastically reducing the search space.

Feldthaus and Møller (2014) perform the first evaluation that measures the correctness of TypeScript definition files. They develop a tool, TSCHECK, which performs a two-stage analysis for detecting incorrect definition files. The first stage conducts a *heap-snapshot*. A library is initialised and the structure of the global object implementing the library is compared against the type specified in the definition file. This technique only works for object types and is unable to verify that a function object conforms to the type in the definition. The second analysis performs a light-weight, but unsound, static analysis that checks function implementations against the expected type. They apply their tool to the ten largest libraries in DefinitelyTyped and their results indicate that errors in definition files are frequent. Their technique implements an offline analysis and requires access to the library source code, making their technique more suitable for library and definition authors, rather than clients that may dynamically import JavaScript libraries at run-time.

Kristensen and Møller (2017a) observe that while the tool developed by Feldthaus and Møller (2014) is successful at detecting libraries that do not conform to their definition, the tool does not help authors write definitions, or maintain them through the life of the library. To address these two weaknesses, Kristensen and Møller (2017a) develop two tools: TSINFER and TSEVOLVE. The former builds directly upon the work by Feldthaus and Møller (2014) to generate a candidate definition file. The latter uses TSINFER to compare changes between two versions of a library. Differences between the two versions are compared, with changes unrelated to the old version removed to reduce spurious warnings. Their approach for definition inference is successful: the majority of all declared classes are inferred and nearly all fields correctly detected.

Kristensen and Møller (2017b) build a tool for detecting errors in definition files, but unlike the static techniques employed by Feldthaus and Møller (2014) and Kristensen and Møller (2017a), they use dynamic analysis similar to TPD. Kristensen and Møller (2017b) evaluate libraries using a *type test script*: a client that is generated

from the definition file of the library. TPD uses generated contracts and existing test code to evaluate the library; a type test script could be viewed as a specialised test file with contract assertions built in. The nature of test scripts means that proxies are not required to implement type checks, and therefore test scripts do not violate non-interference. Furthermore, test scripts are correct-by-construction, rendering blame tracking trivial because the library must always be at fault. Choosing to use a generated client does sacrifice the opportunity to detect errors that are elicited by library unit-tests; TPD found that half of all errors detected were due to faults originating in client code. A unit-test may include permutations of function arguments that are domain specific and unlikely to be randomly generated. Kristensen and Møller (2017b) support test scripts for a wide range of TypeScript types, including generics, but do not enforce parametricity. The choice is justified by suggesting that parametric polymorphism is unsuitable for JavaScript code that is highly dynamic and heavily utilises reflection, such as enumerating all object properties. Parametricity is not necessarily incompatible with this view. An alternate implementation of parametric contracts could assume that universal quantification only ranges over types that implement certain operations such as equality or enumeration.

Mezzetti et al. (2018) present the technique *type regression testing* to detect breaking changes between versions of a library. JavaScript libraries are instrumented with proxies that construct *paths*: a sequence of interactions with the library, including property access and function calls, along with the expected result type of the interaction. Multiple clients that import the library as a dependency are used to construct a model of the library by executing the client test code. Breaking changes between library versions can be detected by comparing the models from each version. The technique is highly accurate, correctly classifying 90% of library changes. Mezzetti et al. (2018) acknowledge proxies violating non-interference when evaluating client code, but do not report the frequency of violations. The model constructed by the technique does not consider parametric polymorphism, which may also suffer from violations of non-interference as experienced by TPD.

6.6.3 Alternate Gradual Typing Design

Vitousek et al. (2014) develop a gradually typed dialect of Python, named Reticulated Python, in which they experiment with alternate approaches to gradual typing design. They acknowledge that the traditional view of gradual typing (Findler and

Felleisen, 2002; Siek and Taha, 2006), also described as a *guarded* semantics, violates non-interference by using proxies that alter object identity (though they do not measure to what extent). To address this, they propose an alternate design called a *transient* semantics that does not require wrappers implemented with proxies. A transient semantics inserts first-order checks at call-sites and object interactions; the name is derived from the observation that checks do not introduce proxies—leaving no trace. However, the choice to forgo proxies means that their presented transient semantics does not provide blame for failing checks.

Vitousek et al. (2017) extend the work on transient gradual typing. First they develop the notion of *open-world soundness*. Inserting a translated source program into a target context should only become stuck due to incorrect behaviour originating in the context. They then add blame to the existing transient semantics (Vitousek et al., 2014), but unlike traditional gradual typing, an error may blame multiple labels rather than one. Finally, they demonstrate that a transient semantics can deliver better performance than a guarded semantics.

Greenman and Felleisen (2018) unify the diverging approaches to gradual typing in a common framework. They distinguish three kinds of gradual typing: the *erasure* embedding that performs no dynamic type checking, including languages such as TypeScript (Bierman et al., 2014); the *higher-order* embedding that eagerly enforces types and uses wrappers, also known as the guarded approach; and the *first-order* embedding that only checks type constructors to prevent against first-order errors, also known as the transient approach. They compare the three approaches formally and practically. A performance evaluation is conducted using the common framework. The erasure embedding is used as the baseline and the slowdown incurred by the first-order and higher-order approaches are compared. Their conclusions align with previous work. The higher-order embedding performs worse on mixed-type programs, in some cases slowing a program by three orders of magnitude. The first-order embedding performs worse on fully-typed programs, always incurring a slowdown while the higher-order embedding achieves speedups in some cases.

Chung et al. (2018) present a unifying approach to gradual typing with a focus on object-oriented languages. They distinguish four kinds of gradual typing in contrast to Greenman and Felleisen (2018) who distinguish three. The classification of Chung et al. (2018) is defined as: *optional*, which is equivalent to the erasure embedding; *behavioural*, which is equivalent to the higher-order embedding; *transient*, which is equivalent to the first-order embedding; and *concrete*, which has no analogue in

the Greenman and Felleisen (2018) classification. The concrete semantics, included in languages such as C#, uses run-time subtype tests on type constructors at casts between typed and untyped code. Chung et al. (2018) develop a *litmus test* for distinguishing approaches to gradual typing; the test consists of three sample programs designed to elicit errors by combining typed and untyped code. A further contribution of their work is KafKa: a statically typed target language with objects, mutable state, and casts. Chung et al. (2018) use KafKa to study the four approaches to gradual typing by providing a translation from each variant into KafKa. To support every class of gradual typing KafKa must provide behavioural casts and structural casts, as well as dynamically typed and statically typed method invocation.

Tunnell Wilson et al. (2018) conduct an evaluation of programmer preference towards the different approaches to gradual typing distinguished by Greenman and Felleisen (2018). They survey a sample of professional developers, computer science students, and Mechanical Turk workers about their preferences and expectations regarding the behaviour of mixed-type code. Respondents were asked to consider if a certain behaviour was expected or not, and liked or not. Tunnell Wilson et al. (2018) conclude that respondents prefer an approach that dynamically enforces static type annotations in full. Furthermore, their study also distinguishes two flavors of the higher-order embedding. Eager, that will immediately traverse the contents of an array to check the elements, and lazy, that will wrap the array and check elements upon access. The study revealed a preference towards the eager approach.

Chapter 7

Conclusion

7.1 Contributions

Gradual typing is a thriving area in research and industry, however the two tracks are not perfectly aligned. Research favours safety and soundness, strongly enforcing the dynamic-static boundary. Industry favours pragmatism, erasing gradual types to simplify implementation and maintain performance.

This work puts theory into practice with the aim of bringing the tracks of research and industry closer in alignment. We provide new insights about what can be done in theory, why practitioners should consider the rigour of sound gradual typing, and which practical concerns must be addressed by research.

In Chapter 2 we review higher-order blame assignment for function, intersection, and union types. Findler and Felleisen (2002) introduced blame and contracts for higher-order functions. Higher-order contracts place obligations on the subject and context of the contract. Positive blame is assigned when the subject violates its obligations; negative blame is assigned when the context violates its obligations. Keil and Thiemann (2015a) introduced higher-order intersection and union contracts, showing that a local contract violation may be insufficient to assign blame to an intersection or union. Contract violations are recorded during the evaluation of a program and interpreted using constraint satisfaction.

In Chapter 3 we show that intersection and union contracts can be implemented in a uniform way, making it easier to design compositional contract libraries. Blame nodes annotate contracts and record the elimination context of function contracts, enabling the immediate decomposition of intersection and union contracts. The immediate reduction of intersection contracts avoids rules based on distributive laws

which duplicate contracts. We stratify blame using assignment and resolution, giving an algorithm for blame in these terms.

In Chapter 4 we present a semantics for contract satisfaction in terms of blame assignment. A value satisfies a contract if monitoring the value using that contract never elicits positive blame. A continuation satisfies a contract if monitoring the continuation using that contract never elicits negative blame. Our definition of contract satisfaction evokes a new form a contract soundness called witness soundness. Witness soundness decouples contract soundness from divergence by explicitly representing the logical implication associated with function contracts. Each contract operator gives rise to a pair of monitoring properties defined using contract satisfaction. One rule relates positive satisfaction for values and one rule relates negative satisfaction for continuations.

In Chapter 5 we introduce our tool based on sound gradual typing—*The Prime Directive*. TPD synthesises wrapper code from a TypeScript definition file which is then placed at the boundary between library and client. The wrapper code enforces conformance to the definition file using a contract. We show how TPD uses proxies to implement higher-order function contracts and parametric polymorphic contracts. A polymorphic contracts seals input data using a proxy, preventing inspection and enforcing abstraction.

In Chapter 6 we present the results of applying TPD to the DefinitelyTyped repository. Our evaluation measures violations of conformance and non-interference and we provide concrete examples of both. We found that from the 122 libraries we evaluated, 62 libraries did not conform to the corresponding definition file. JavaScript proxies are presented as a suitable mechanism for behavioural intercession such as contract checking. We found that from the 122 libraries we tested, 22 libraries exhibited interference due to proxies. Each source of interference was classified. We found that from the 23 distinct violations of non-interference, 17 violations were due to the fundamental design of proxies in JavaScript.

7.2 Future Work

Some questions have been answered—many remain. In this section we discuss future avenues of research prompted by our results.

Contract Performance Taming the performance overhead of contracts and sound gradual typing is an important area of research. Multiple approaches have been pursued, including cast and contract merging (Feltey et al., 2018; Herman et al., 2010; Siek et al., 2015a; Siek and Wadler, 2010), and optimisation of contract run-times (Bauman et al., 2017; Richards et al., 2017). This body of work has targeted dynamic type checking for operators that induce blame from a single violation, such as function or object types. Intersection and union contracts require blame provenance to track contract elimination, and require state to track contract violations. This additional information is likely to impact performance, and in the presence of recursive types, space consumption may be unbounded. Additional work is needed to understand the cost of intersection and union contracts, and whether existing optimisation techniques can be adapted. For example, the technique of collapsible contracts (Feltey et al., 2018) drops inner contracts that are subsumed by outer contracts. Collapsing intersection and union contracts may be non-trivial. Contract subsumption is no longer uniquely determined by the checks that a contract can perform, but also the intermediate violation state associated with that contract. The challenge is to correctly determine which checks may be collapsed and which checks must be retained.

Contract Semantics Our definition of contract satisfaction comes equipped with a notion of contract soundness which we refer to as witness soundness. The definition of witness soundness is coupled to the structure of blame nodes that we present. Generalising our contract semantics to other systems will require an abstract presentation of witness soundness that is parameterised by an algebra for blame tracking. In particular, witness soundness should refer to the abstract operations used to synthesise domain and codomain blame nodes when wrapping a function contract.

Another direction of work is to understand the relationship between our definition of contract satisfaction and that of Dimoulas and Felleisen (2011). The definitions are equivalent for simple types, however the approach by Dimoulas and Felleisen (2011) does not immediately extend to intersection and union. Further work is required to understand the necessary extensions to their work to support intersection and union, and how the resulting system compares to the system we present.

Sound Gradual Typing in Practice Our application of sound gradual typing liberally uses proxies to implement object contracts, function contracts, and parametric polymorphic contracts. This approach is not practical when paired with the current

design of JavaScript proxies; opaque proxies frequently violate non-interference. One direction of further work is to consider a more judicious use of proxies. Eagerly enforcing object contracts sacrifices safety in the presence of mutation, but eliminates the need for proxies. With such a strategy it is unclear how many violations of conformance will be missed, and how many violations of non-interference will be avoided.

TypeScript is a language that evolves at a rapid pace. Since we conducted our evaluation of gradual typing, DefinitelyTyped has grown significantly and TypeScript has added many new types, including *mapped object types* and *conditional types*. This growth presents new questions. Are users now more experienced at authoring definition files? Have the new types made the process of authoring definition files harder? A further challenge is implementing dynamic type checking for the new type operators, in particular, conditional types.

7.3 Summary

This work shows that intersection and union contracts can be used to build a sound gradual typing tool for TypeScript. Practical implementations of gradual typing favour boundary erasure, placing the burden of writing correct type definitions solely on the shoulders of programmers. Our evaluation shows that JavaScript libraries frequently fail to conform their TypeScript definition file. The burden of writing correct type definitions appears to be a heavy one. Sound gradual typing offers a way to lessen the load by dynamically checking conformance.

Our evaluation also reveals practical challenges associated with implementing sound gradual typing. Proxies are presented as one of the canonical mechanisms to implement contracts. However, we found that JavaScript proxies frequently violate non-interference when used to implement contracts.

Combining ideas can produce results that are greater than the sum of their parts. Gradual typing combines static and dynamic typing to get the best of both. In this work we combine theory and practice to evaluate what sound gradual typing has to offer, and to understand what sound gradual typing still has to solve. While we find obstacles in the implementation of sound gradual typing, we also find strong benefits in its application. Practical sound gradual typing remains an unsolved problem, but a problem with a strong motivation. For research, that is a good place to be.

Index of Notation

Appendix A

Freshness and Blame

A.1 Preservation of Freshness

Definition A.1.1 (Blame node length)**.**

$$\begin{aligned} \mathsf{length}(\pm\ell[P]) &= &= \mathsf{length}(P) \\ \mathsf{length}(p \bullet d_\circ^\pm[P]) &= &= \mathsf{length}(p) + 1 + \mathsf{length}(P) \end{aligned}$$

Lemma A.1.2 (Preservation of Freshness)**.** *If* $p \# \langle \Phi, \Delta, K, M\rangle$ *and* $\langle \Phi, \Delta, K, M\rangle \longrightarrow \langle \Phi', \Delta', K', N\rangle$ *then* $p \# \langle \Phi', \Delta', K', N\rangle$.

Proof. First note that if $p \leq q$ or $p \leq_\Delta q$ then q is greater than or equal to p in *length*, defined in Definition A.1.1. We proceed with case analysis on reduction. We start with the interesting cases that interact with blame nodes.

Case 1. $\langle \Delta, K \circ V@^q A \to B\,\square, W\rangle \longrightarrow \langle \Delta', K, (V\,(W@^{-q \gg \mathsf{dom}_i} A))@^{q \gg \mathsf{cod}_i} B\rangle$

Let $(\Delta', i) = \delta(\Delta, q)$. In a wrap reduction only the term and context tracker are affected. First we consider freshness with respect to the new blame nodes. We break the proof down, considering the lengths of p and q. We then consider freshness for the resulting context tracker Δ'.

- $p \not\leq \pm(q \gg \mathsf{dom}_i)$ and $p \not\leq \pm(q \gg \mathsf{cod}_i)$
  - q is greater than or equal to p in length.

    If $p \leq \pm(q \gg \mathsf{dom}_i)$ then $p \leq q$, contradicting the assumption. If $p \leq \pm(q \gg \mathsf{cod}_i)$ then $p \leq q$, contradicting the assumption.
  - q is shorter than p.

If $\pm(q \gg \mathrm{dom}_i)$ remains shorter than p then $p \not\leq \pm(q \gg \mathrm{dom}_i)$. If $\pm(q \gg \mathrm{cod}_i)$ remains shorter than p then $p \not\leq \pm(q \gg \mathrm{cod}_i)$.

If the extension causes them to become equal in length, then they are only "unfresh" when $\pm(q \gg \mathrm{dom}_i) = p$. If that were the case, then in the prior reduction we would have $q \leq_\Delta \pm(q \gg \mathrm{dom}_i)$, where $\Delta(q) = i$, contradicting the assumption that $q \not\leq_\Delta p$. We conclude that $p \not\leq \pm(q \gg \mathrm{dom}_i)$ and $p \not\leq \pm(q \gg \mathrm{cod}_i)$.

- $\pm(q \gg \mathrm{dom}_i) \not\leq_{\Delta'} p$ and $\pm(q \gg \mathrm{cod}_i) \not\leq_{\Delta'} p$

  – q is greater than or equal to p in length.

  If q is greater than or equal to p in length, then so is the extension, therefore $\pm(q \gg \mathrm{dom}_i) \not\leq_{\Delta'} p$ and $\pm(q \gg \mathrm{cod}_i) \not\leq_{\Delta'} p$.

  – q is shorter than p.

  We now show that $\pm(q \gg \mathrm{dom}_i) \not\leq_{\Delta'} p$ when q is shorter than p. We use an inductive argument, where the interesting case is when the extension of q causes them to become equal. We use similar reasoning as before: that if they are equal after the reduction then it must have been the case that $q \leq_\Delta p$, contradicting the assumption.

From these cases we conclude that p is fresh in the resulting term. We show that p is fresh in the new context tracker.

- $p \# \Delta'$

The only adjustment to the new state Δ' is the q may be included with a counter of 1 if $q \notin \Delta$, or the existing counter for q is incremented. Given that p and q are distinct by assumption, then including q does not violate freshness. When incrementing the counter we observe that this could only invalidate freshness if previously $\delta(\Delta, q) = (\Delta', j)$ and $i' > j$ for some j in a wrap index, then after, $\delta(\Delta', q) = (\Delta'', i'')$ and $i \leq j$. Given that the function δ strictly increments values in Δ' this cannot occur.

Case 2. $\langle K, V@^q A \cap B\rangle \longrightarrow \langle K, (V@^{q \bullet \mathsf{left}^+_\cap[\mathsf{nil}]} A)@^{q \bullet \mathsf{right}^+_\cap[\mathsf{nil}]} B\rangle$

- $p \not\leq (q \bullet \mathsf{left}^+_\cap[\mathsf{nil}])$ and $p \not\leq (q \bullet \mathsf{right}^+_\cap[\mathsf{nil}])$

  - q is greater than or equal to p in length.

    If $p \leq (q \bullet \mathsf{left}^+_\cap[\mathsf{nil}])$ or $p \leq (q \bullet \mathsf{right}^+_\cap[\mathsf{nil}])$ then it must be the case that $p \leq q$, contradicting the assumption.

  - q is shorter than p.

    If q is less than p in length, then $p \leq (q \bullet \mathsf{left}^+_\cap[\mathsf{nil}])$ only holds when $p = (q \bullet \mathsf{left}^+_\cap[\mathsf{nil}])$. If this were true, then prior to the reduction we would have $q \leq_\Delta (q \bullet \mathsf{left}^+_\cap[\mathsf{nil}])$, contradicting the assumption that $q \not\leq_\Delta p$. The same reasoning applies to blame node $q \bullet \mathsf{right}^+_\cap[\mathsf{nil}]$.

- $(q \bullet \mathsf{left}^+_\cap[\mathsf{nil}]) \not\leq_\Delta p$ and $(q \bullet \mathsf{right}^+_\cap[\mathsf{nil}]) \not\leq_\Delta p$

  - q is greater than or equal to p in length.

    In such a case then the extension of q is also greater in length than p, so $(q \bullet \mathsf{left}^+_\cap[\mathsf{nil}]) \not\leq_\Delta p$ and $(q \bullet \mathsf{right}^+_\cap[\mathsf{nil}]) \not\leq_\Delta p$.

  - q is shorter than p.

    To show that $(q \bullet \mathsf{left}^+_\cap[\mathsf{nil}]) \not\leq_\Delta p$ when $q \not\leq_\Delta p$ and q is shorter than p we use inductive reasoning. The interesting case is the base case, where p and q can only become "unfresh" if $p = (q \bullet \mathsf{left}^+_\cap[\mathsf{nil}])$. If $p = (q \bullet \mathsf{left}^+_\cap[\mathsf{nil}])$ then we have:

    $$q \leq_\Delta (q \bullet \mathsf{left}^+_\cap[\mathsf{nil}]) \equiv q \leq_\Delta p$$

    which is a contradiction of our initial assumption $q \not\leq_\Delta p$. We use the same reasoning for the blame node $q \bullet \mathsf{right}^+_\cap[\mathsf{nil}]$.

Case 3. $\langle K, V@^q A \cup B\rangle \longrightarrow \langle K, (V@^{q \bullet \mathsf{left}^+_\cup[\mathsf{nil}]} A)@^{q \bullet \mathsf{right}^+_\cup[\mathsf{nil}]} B\rangle$

This case follows the same reasoning as the intersection case; the different type in the branch nodes has no impact on our reasoning.

Case 4. $\langle \Phi, \Delta, K, V@^q \iota\rangle \longrightarrow \langle \Phi', \Delta, K, M\rangle$ where $\Phi', M = \mathit{blame}(q, \Phi, V)$

- $p \# \Phi'$

  When a contract is violated the blame state may be updated. We must show that p remains fresh in the new blame state Φ'.

  Consider that the blame state Φ' can only differ by adding q, or some prefix of q according to the *parent* function. That is, $\Phi' = \Phi \cup \Phi_q$ where Φ_q is a subset of the reflexive and transitive closure of *parent* on q.

We must show that there is no $q' \in \Phi_q$ where $p \leq q'$. Observing that the effect of $\mathit{parent}(q)$ can always be undone by a forward reading of $\leq$, and that if $p \leq q'$ for some $q' \in \Phi_q$, then $p \leq q$, contradicting the assumption.

Case 5. The remaining reduction rules do not introduce new blame nodes into the program configuration. Freshness is preserved by assumption, using congruence to reconstruct freshness for the new configuration.

□

A.2 Absence of Blame

Lemma A.2.1 (Freshness closure)**.** *If* $p \# \langle \Phi, \Delta, K, M \rangle$ *and* $\langle \Phi, \Delta, K, M \rangle \longrightarrow^* \langle \Phi', \Delta', K', N \rangle$ *then* $p \# \langle \Phi', \Delta', K', N \rangle$.

Proof. By induction on the derivation of $\longrightarrow^*$, using Lemma A.1.2 to preserve freshness for each step. □

Lemma A.2.2 (Freshness and Blame Implication)**.** *If* $p \# \Phi$ *then* $\Phi \not\models p$.

Proof. By contrapositive. Namely, if $\Phi \models p$ then $\exists q \in \Phi. p \leq \pm q$. By the definition of $\Phi \models p$ then there is a repathing of p in Φ such that $\mathit{prefix}(\mathit{path}(p), P')$, for some path P'. We pick $q = \mathit{replace\text{-}path}(p, P')$, observing that $p \leq q$ holds under basic path extension. □

Lemma 4.2.3 (Safety by Freshness)**.** *If* $p \# \langle \Phi, \Delta, K, M \rangle$ *then* $\langle \Phi, \Delta, K, M \rangle \not\longrightarrow^* \langle \lightning \pm p \rangle$.

Proof. By contradiction. Assume that there is a configuration that implicates p. By Lemma A.2.1 and Lemma A.2.2 we form a contradiction as p must be fresh in the implicating blame state. □

Lemma A.2.3 (Configuration Safety & Blame Safety)**.** *If* $\langle \Phi, \Delta, K, M \rangle \not\longrightarrow^* \langle \lightning \pm \ell[\mathsf{nil}] \rangle$ *then* $\langle \Phi, \Delta, K, M \rangle \not\longrightarrow^* \langle \Phi', \Delta', \mathit{Id}, \mathtt{blame} \pm \ell \rangle$.

Proof. We assume that source programs do not start with blame terms, and that blame only arises from a contract failure during evaluation. Assume that the conclusion is false, such that there is some configuration:

$$
\begin{aligned}
&\langle \Phi, \Delta, K, M \rangle \\
&\longrightarrow^* \langle \Phi', \Delta', K', V @^{p_{\pm \ell}} A \rangle \\
&\longrightarrow \langle \Phi'', \Delta', K', \mathtt{blame} \pm \ell \rangle \\
&\longrightarrow \langle \Phi'', \Delta', \mathit{Id}, \mathtt{blame} \pm \ell \rangle
\end{aligned}
$$

By our definition of *blame*, then the root node of $p_{\pm\ell}$ has the form $\pm\ell[P]$, for some path P, and the root node is in the state $\pm\ell[P] \in \Phi''$. If $\pm\ell[P] \in \Phi''$ then $\Phi'' \models \pm\ell[\mathsf{nil}]$, however by assumption we know $\Phi'' \not\models \pm\ell[\mathsf{nil}]$, forming a contradiction. □

Appendix B

Contract Soundness

B.1 Witness Enriched Calculus

To assist with technical developments we present a *witness enriched* version of $\lambda^{\cap\cup}$ (Figure B.1). The enriched calculus retains blame node information on values after contract application, however the extra information has no run-time effect.

Pre-values P (unambiguous with blame paths) are supplemented with a witness trace. A trace contains the blame node and type for a contract that has been applied to P. We assume that all source programs start with empty traces.

Definition B.1.1 (Enrichment Erasure)**.** *Define enrichment erasure* $\dagger$ *by congruence on program configurations, and these cases:*

$$(k^w)^\dagger = k$$
$$((\lambda x.M)^w)^\dagger = \lambda x.M^\dagger$$

Lemma B.1.2 (Enrichment Simulation)**.** *The witness enriched calculus simulates* $\lambda^{\cap\cup}$*. Assume:*

- $M = M_1^\dagger$ *and* $K = K_1^\dagger$
- *If* $\langle \Phi, \Delta, K, M\rangle \longrightarrow \langle \Phi', \Delta', K', M'\rangle$ *then* $\langle \Phi, \Delta, K_1, M_1\rangle \longrightarrow \langle \Phi', \Delta', K_1', M_1'\rangle$ *where* $M' = {M_1'}^\dagger$ *and* $K' = {K_1'}^\dagger$*, for some* K_1', M_1'
- *If* $\langle \Phi, \Delta, K_1, M_1\rangle \longrightarrow \langle \Phi', \Delta', K_1', M_1'\rangle$ *then* $\langle \Phi, \Delta, K, M\rangle \longrightarrow \langle \Phi', \Delta', K', M'\rangle$ *where* $M' = {M_1'}^\dagger$ *and* $K' = {K_1'}^\dagger$*, for some* K', M'

Proof. By case analysis on $\longrightarrow$. Witness traces have no influence on reduction. We prove that $\dagger$ commutes with substitution using induction in the standard way. □

Mark	m	::=	$\checkmark \mid \mathsf{x}$
Witness Trace	w	::=	$\cdot \mid w;(p, A, \Phi, m)$
Terms	M, N	::=	$\cdots \mid k^w \mid (\lambda x.M)^w$
Pre-Values	P	::=	$k^w \mid (\lambda x.M)^w$
Values	V, W	::=	$P \mid V @^p A \to B$
Contract Context	C	::=	$[] \mid C @^p A$
Value Contract Context	$\mathcal{V}$	::=	$[] \mid \mathcal{V} @^p A \to B$

$$\vdots$$

$$\langle K \circ (\lambda x.M)^w \, \square, V \rangle \longrightarrow \langle K, M[x := V] \rangle$$

$$\langle \Phi, K, \mathcal{V}[P^w] @^p \mathsf{any} \rangle \longrightarrow \langle \Phi, K, \mathcal{V}[P^{w;(p,\mathsf{any},\Phi,\checkmark)}] \rangle$$

$$\langle \Phi, K, \mathcal{V}[P^w] @^p \iota \rangle \longrightarrow \langle \Phi, K, \mathcal{V}[P^{w;(p,\iota,\Phi,\checkmark)}] \rangle$$

if $\mathcal{V}[P^w] : \iota$

$$\langle \Phi, K, \mathcal{V}[P^w] @^p \iota \rangle \longrightarrow \langle \Phi', K, M \rangle$$

otherwise, where $\Phi', M = \mathit{blame}(p, \Phi, \mathcal{V}[P^{w;(p,\iota,\Phi,\mathsf{x})}])$

Figure B.1: Enriched Syntax and Operational Semantics (extends and modifies)

Definition B.1.3 (Witness Trace Static Signature)**.** *Define the static trace of a witness trace by erasing blame states and marks.*

$$\|\cdot\| = \cdot$$
$$\|w;(p,A,\Phi,m)\| = \|w\|;(p,A)$$

Definition B.1.4 (Witness Signature)**.** *Define the signature of a contract context as* $\mathrm{sig}(C)$*. We make liberal use of notation where* $+\!+$ *appends traces, defined in the standard way for cons-lists.*

$$\mathrm{sig}([]) = \cdot$$
$$\mathrm{sig}(C@^p A) = \mathrm{sig}(C) +\!+ \mathrm{sig}(p,A)$$
$$\mathrm{sig}(p,\iota) = (p,\iota)$$
$$\mathrm{sig}(p,\mathsf{any}) = (p,\mathsf{any})$$
$$\mathrm{sig}(p,A\to B) = \cdot$$
$$\mathrm{sig}(p,A\cap B) = \mathrm{sig}(p\bullet \mathsf{left}^+_\cap[\mathsf{nil}],A) +\!+ \mathrm{sig}(p\bullet \mathsf{right}^+_\cap[\mathsf{nil}],B)$$
$$\mathrm{sig}(p,A\cup B) = \mathrm{sig}(p\bullet \mathsf{left}^+_\cup[\mathsf{nil}],A) +\!+ \mathrm{sig}(p\bullet \mathsf{right}^+_\cup[\mathsf{nil}],B)$$

Definition B.1.5 (Function Wrap Signature)**.** *Define the wrap signatures of a value contract context as* $\mathrm{sig}_{\mathsf{dom}}(\Delta,\mathcal{V})$ *and* $\mathrm{sig}_{\mathsf{cod}}(\Delta,\mathcal{V})$*.*

$$\mathrm{sig}_{\mathsf{dom}}(\Delta,[]) = \cdot$$
$$\mathrm{sig}_{\mathsf{cod}}(\Delta,[]) = \cdot$$
$$\mathrm{sig}_{\mathsf{dom}}(\Delta,\mathcal{V}@^p A\to B) = \mathrm{sig}(-p \gg \mathrm{dom}_{\Delta(p)},A) +\!+ \mathrm{sig}_{\mathsf{dom}}(\Delta,\mathcal{V})$$
$$\mathrm{sig}_{\mathsf{cod}}(\Delta,\mathcal{V}@^p A\to B) = \mathrm{sig}_{\mathsf{cod}}(\Delta,\mathcal{V}) +\!+ \mathrm{sig}(p \gg \mathrm{cod}_{\Delta(p)},B)$$

Definition B.1.6 (Rooted Contexts)**.** *We write* C_p *and* $\mathcal{V}_p$ *for contexts where all contracts in the context are annotated with node* q *such that* $p \le q$*.*

Definition B.1.7 (Prefix replacement)**.** *For a blame node* r *where* $p \le r$*, we write* $r[q]$ *to be the replacement of the prefix* p *by* q*. When we write* C_p *and* C_q *we mean that the contexts are the same up to prefix replacement. That is, replacing all nodes* r *in* C_p *with* $r[q]$ *produces* C_q*, and replacing all nodes* r *in* C_q *with* $r[p]$ *produces* C_p*.*

Lemma B.1.8 (Contract Context Reduction)**.** *For any configuration* $\langle\Phi,\Delta,K,C_p[P^w]\rangle$ *then one of the following holds:*

(a) $\langle\Phi,\Delta,K,C_p[P^w]\rangle \longrightarrow^* \langle\Phi',\Delta',K,\mathcal{V}_p[P^{w+\!+w'}]\rangle$ *where* $\|w'\| = \mathrm{sig}(C_p)$*.*

(b) $\langle\Phi,\Delta,K,C_p[P^w]\rangle \longrightarrow^* \langle\Phi,\Delta,Id,\mathtt{blame}\ \pm\ell\rangle$ *where* $\pm\ell$ *is the label for* p*.*

Proof. By induction on C, observing that $\mathrm{sig}(C)$ mirrors the behaviour of contract decomposition in reduction. □

$$\textsc{g-const}\ \frac{}{k^w\ :\ p \text{ guards } q} \qquad \textsc{g-abs}\ \frac{M\ :\ p \text{ guards } q}{(\lambda x.M)^w\ :\ p \text{ guards } q} \qquad \textsc{g-var}\ \frac{}{x\ :\ p \text{ guards } q}$$

$$\textsc{g-app}\ \frac{M\ :\ p \text{ guards } q \qquad N\ :\ p \text{ guards } q}{MN\ :\ p \text{ guards } q} \qquad \textsc{g-blame}\ \frac{}{\mathcal{E}[\texttt{blame}\ \pm\ell]\ :\ p \text{ guards } q}$$

Figure B.2: Predicate $\Phi, \Delta, M\ :\ p$ guards q (Basic)

B.2 Guarding

We describe a predicate on programs that states that blame node p guards blame node q. Any blame on q must happen after equivalent blame for p, and any blame on $-p$ must happen after equivalent blame for $-q$.

In this section we rely on an operation semantics that uses evaluation contexts rather than configurations, implicitly relying on a simulation between the two (Felleisen and Friedman, 1986). We do this to save duplicating work when defining predicates on contract contexts that span continuations and terms.

Definition B.2.1 (Guarding for p and q)**.** *Define predicate* $\Phi, \Delta, M\ :\ p$ guards q *for blame nodes* p *and* q*, where* $p \not\leq \pm q$ *and* $q \not\leq \pm p$*. When we omit* Φ *or* Δ *from a rule we implicitly assume that they are guarded in the conclusion and any premises. The predicate is defined in Figure B.2, Figure B.3, and Figure B.4.*

Lemma B.2.2 (Preservation of guards)**.** *The predicate guards is preserved by reduction. If* $\Phi, \Delta, M\ :\ p$ guards q *and* $\Phi, \Delta, M \longrightarrow \Phi', \Delta', N$ *then* $\Phi', \Delta', N\ :\ p$ guards q*.*

Proof. By induction on $\Phi, \Delta, M\ :\ p$ guards q. We handle each case from the definition in turn.

Case 1. (G-CONST) Trivial as the term does not reduce.

Case 2. (G-ABS) Trivial as the term does not reduce.

Case 3. (G-VAR) Trivial as the term does not reduce.

Case 4. (G-APP) The following cases apply

- M reduces. Apply the IH then apply (G-APP).

G-CONTRACT

$$\frac{q \not\leq r \qquad p \not\leq r \qquad r \not\leq_\Delta q \qquad r \not\leq_\Delta p \qquad \Delta, M \ : \ p \text{ guards } q}{\Delta, M@^r A \ : \ p \text{ guards } q}$$

G-GUARD-TERM

$$\frac{M \neq V \qquad M \ : \ p \text{ guards } q}{C_q[C_p[M]] \ : \ p \text{ guards } q}$$

G-GUARD-VAL

$$\frac{\mathcal{V}[P^{w +\!\!+ w'_p +\!\!+ w''_q}] \ : \ p \text{ guards } q \qquad \|w'_p\| +\!\!+ \text{sig}(C'_p) = \|w''_{[p]}\| +\!\!+ \text{sig}(C_{[p]})}{C_q[C'_p[\mathcal{V}[P^{w +\!\!+ w'_p +\!\!+ w''_q}]]] \ : \ p \text{ guards } q}$$

G-GUARD-APP

$$\frac{\begin{array}{c}\mathcal{V}'_p \neq [] \qquad N = C^2_{-p}[C^1_{-q}[\mathcal{V}[P^{w +\!\!+ w'_{-q} +\!\!+ w''_{-p}}]]] \\ \Delta, V \ : \ p \text{ guards } q \qquad \Delta, \mathcal{V}[P^{w +\!\!+ w'_{-q} +\!\!+ w''_{-p}}] \ : \ p \text{ guards } q \\ \|w'_{-q}\| +\!\!+ \text{sig}(C^1_{-q}) +\!\!+ \text{sig}_{\text{dom}}(\Delta, \mathcal{V}_q) = \|w''_{[-q]}\| +\!\!+ \text{sig}(C^2_{[-q]}) +\!\!+ \text{sig}_{\text{dom}}(\Delta, \mathcal{V}'_{[q]}) \\ \text{sig}_{\text{cod}}(\Delta, \mathcal{V}'_p) +\!\!+ \text{sig}(C^3_p) = \text{sig}_{\text{cod}}(\Delta, \mathcal{V}_{[p]}) +\!\!+ \text{sig}(C^4_{[p]})\end{array}}{\Delta, C^4_q[C^3_p[(\mathcal{V}_q[\mathcal{V}'_p[V]]\, N)]] \ : \ p \text{ guards } q}$$

G-GUARD-TERM-NEG

$$\frac{M \neq V \qquad M \ : \ p \text{ guards } q}{C_{-p}[C_{-q}[M]] \ : \ p \text{ guards } q}$$

G-GUARD-VAL-NEG

$$\frac{\mathcal{V}[P^{w +\!\!+ w'_{-q} +\!\!+ w''_{-p}}] \ : \ p \text{ guards } q \qquad \|w'_{-q}\| +\!\!+ \text{sig}(C'_{-q}) = \|w''_{[-q]}\| +\!\!+ \text{sig}(C_{[-q]})}{C_{-p}[C'_{-q}[\mathcal{V}[P^{w +\!\!+ w'_{-q} +\!\!+ w''_{-p}}]]] \ : \ p \text{ guards } q}$$

G-GUARD-APP-NEG

$$\frac{\begin{array}{c}\mathcal{V}'_{-q} \neq [] \qquad N = C^2_q[C^1_p[\mathcal{V}[P^{w +\!\!+ w'_p +\!\!+ w''_q}]]] \\ \Delta, V \ : \ p \text{ guards } q \qquad \Delta, \mathcal{V}[P^{w +\!\!+ w'_p +\!\!+ w''_q}] \ : \ p \text{ guards } q \\ \|w'_p\| +\!\!+ \text{sig}(C^1_p) +\!\!+ \text{sig}_{\text{dom}}(\Delta, \mathcal{V}_{-p}) = \|w''_{[p]}\| +\!\!+ \text{sig}(C^2_{[p]}) +\!\!+ \text{sig}_{\text{dom}}(\Delta, \mathcal{V}'_{[-p]}) \\ \text{sig}_{\text{cod}}(\Delta, \mathcal{V}'_{-q}) +\!\!+ \text{sig}(C^3_{-q}) = \text{sig}_{\text{cod}}(\Delta, \mathcal{V}_{[-q]}) +\!\!+ \text{sig}(C^4_{[-q]})\end{array}}{\Delta, C^4_{-p}[C^3_{-q}[(\mathcal{V}_{-p}[\mathcal{V}'_{-q}[V]]\, N)]] \ : \ p \text{ guards } q}$$

Figure B.3: Predicate $\Phi, \Delta, M \ : \ p$ guards q (Contracts)

G-BLAME-STATE

$$\frac{\forall q' \in \Phi.\ q \leq q' \Rightarrow q'[p] \in \Phi \qquad \forall p' \in \Phi.\ -p \leq p' \Rightarrow p'[-q] \in \Phi}{\Phi\ :\ p \text{ guards } q}$$

G-TRACKER

$$\frac{\forall p', q' \in \Delta.\ p \leq \pm p' \wedge q \leq \pm q' \Rightarrow \Delta(p') = \Delta(q')}{\Delta\ :\ p \text{ guards } q}$$

Figure B.4: Predicate $\Phi, \Delta, M\ :\ p$ guards q (Blame State and Context Tracker)

- $M = V$ and N reduces. Apply the IH then apply (G-APP).
- $M = V$ and $N = W$. If both are values then either beta reduction or function wrapping applies. (We cannot get stuck by assumption).
  - $M = (\lambda x.M')^w$. The program reduces to $M'[x := W]$. We prove that substitution preserves *guards* in the standard way using induction. The important observation is that we cannot affect witness traces in cases such as (G-GUARD-VAL) because the trace is attached to a pre-value P and can therefore not be replaced by substitution.
  - $M = V@^r A \to B$. The only cases that can apply to M are (G-CONTRACT), (G-GUARD-VAL), (G-GUARD-VAL-NEG).
    - (G-CONTRACT).
      We have $\Phi, \Delta, (V@^r A \to B)\, W \longrightarrow \Phi, \Delta', (V\,(W@^{-r \gg \mathrm{dom}_i} A))@^{r \gg \mathrm{cod}_i} B$
      - $(W@^{-r \gg \mathrm{dom}_i} A)\ :\ p$ guards q by (G-CONTRACT).
      - $V\,(W@^{-r \gg \mathrm{dom}_i} A)$ by (G-APP)
      - $(V\,(W@^{-r \gg \mathrm{dom}_i} A))@^{r \gg \mathrm{cod}_i} B$ by (G-CONTRACT).
      - $\Delta'\ :\ p$ guards q by observing that r is unrelated to p and q.
    - (G-GUARD-VAL). We have $\Phi, \Delta, (\mathcal{V}_q[\mathcal{V}'_p[V]]@^{q_1} A \to B)\, W$ $\longrightarrow \Phi, \Delta', (\mathcal{V}_q[\mathcal{V}'_p[V]]\,(W@^{-q_1 \gg \mathrm{dom}_i} A))@^{q_1 \gg \mathrm{cod}_i} B$. We apply (G-GUARD-APP) with $C_1 = []@^{-q_1 \gg \mathrm{dom}_i} A$, $C_2 = []$, $C_3 = []$, $C_4 = []@^{q_1 \gg \mathrm{cod}_i} A$, $w'_{-q} = \cdot$, $w''_{-p} = \cdot$.
    - (G-GUARD-VAL-NEG). Follows the same reasoning as the previous case but applies (G-GUARD-APP-NEG) instead.

Case 5. (G-BLAME) The term reduces if $\mathcal{E} \neq []$, in which case we apply (G-BLAME) with a new evaluation context $\mathcal{E}' = []$.

Case 6. (G-CONTRACT) The following cases apply

- M reduces. Apply the IH then apply (G-CONTRACT).
- $M = V$ and $A = \mathsf{any}$. Reduces to V which is guarded by assumption.
- $M = V$ and $A = \iota$. The result of the reduction is one of:
  - Φ, Δ, V when the value is conforming. This is guarded by assumption.
  - Φ', Δ, V when the value does not conform. We observe that the blame state can only change in ways independent of p and q, and is therefore guarded. V is guarded by assumption.
  - $\Phi', \Delta, \mathtt{blame}\ \pm\ell$. Apply (G-BLAME).
- $M = V$ and $A = A_1 \cap B_1$. Split the contract and apply (G-CONTRACT) to both branch contracts.
- $M = V$ and $A = A_1 \cup B_1$. Split the contract and apply (G-CONTRACT) to both branch contracts.

Case 7. (G-GUARD-TERM) Apply IH to M. If the result is still a non-value, then apply (G-GUARD-TERM) again. Otherwise apply (G-GUARD-VAL). Pick C_q and C'_p to be the contract contexts from the assumption. Pick w to be the trace on the resulting value, and pick w'_p and w''_q to be $\cdot$.

Case 8. (G-GUARD-VAL) We assume that C' has the same (or) fewer contracts than C, and that when C' is not a value context then $w''_q = \cdot$; we have not yet started evaluating contracts from the outer context. We distinguish the cases where C'_p is a value context or has further reductions.

- $C'_p = C''_p[[]@^{p_i} A]$. Consider the cases for A.
  - $A = \mathsf{any}$. The contract evaluates without failure; apply (G-GUARD-VAL). Pick new w'_p as $w'_p;(p_i, \mathsf{any}, \Phi, \checkmark)$. Pick new C'_p as C''_p. Keep C_q, w, w''_q as before. We note $\|w'_p\|$ ++ $\mathsf{sig}(C''_p[[]@^{p_i}\mathsf{any}]) = \|w'_p;(p_i, \mathsf{any}, \Phi, \checkmark)\|$ + + $\mathsf{sig}(C''_p)$.
  - $A = \iota$. If the value conforms to type ι then the reasoning is similar to the case for any. Instead we pick the new w'_p as $w'_p;(p_i, \iota, \Phi, \checkmark)$.

    If the value does not conform then either we raise blame or return the value (both with a modified blame state). In the case of blame we apply (G-BLAME) with $\mathcal{E} = C_q[C''_p]$. In the case that we do not raise blame then

we pick the new w'_p as $w'_p;(p_i, \iota, \Phi, \mathsf{x})$. We instantiate the remaining variables as was the case for any and apply (G-GUARD-VAL). Again, we note that $\|w'_p\| \mathbin{+\!\!+} \mathrm{sig}(C''_p[[]@^{p_i}\iota]) = \|w'_p;(p_i, \iota, \Phi, \mathsf{x})\| \mathbin{+\!\!+} \mathrm{sig}(C''_p)$.

We know that for resulting blame state Φ', then $\Phi' : p$ guards q as we do not include any nodes p_n such that $q \leq p_n$ or $-p \leq p_n$.

– $A = A_1 \cap B_1$. We split the contract into two branch contracts.

Pick $C'_p = C''_p[([]@^{p_i \bullet \mathsf{left}^+_\cap[\mathsf{nil}]} A_1)@^{p_i \bullet \mathsf{right}^+_\cap[\mathsf{nil}]} B_1]$. All other existential variables are instantiated from the assumption. We note that:

$$\mathrm{sig}(C''_p[[]@^{p_i} A_1 \cap B_1]) = \mathrm{sig}(C''_p[([]@^{p_i \bullet \mathsf{left}^+_\cap[\mathsf{nil}]} A_1)@^{p_i \bullet \mathsf{right}^+_\cap[\mathsf{nil}]}.B_1])$$

– $A = A_1 \cup B_1$. We split the contract into two branch contracts.

Pick $C'_p = C''_p[([]@^{p_i \bullet \mathsf{left}^+_\cup[\mathsf{nil}]} A_1)@^{p_i \bullet \mathsf{right}^+_\cup[\mathsf{nil}]} B_1]$. All other existential variables are instantiated from the assumption. We note that:

$$\mathrm{sig}(C''_p[[]@^{p_i} A_1 \cup B_1]) = \mathrm{sig}(C''_p[([]@^{p_i \bullet \mathsf{left}^+_\cup[\mathsf{nil}]} A_1)@^{p_i \bullet \mathsf{right}^+_\cup[\mathsf{nil}]}.B_1])$$

• $C'_p = \mathcal{V}'_p$, then $C_q = C^1_q[[]@^{q_i} A]$ for some C^1_q. Consider the cases for A.

– $A = \mathsf{any}$. The contract evaluates without failure; apply (G-GUARD-VAL). Pick new w''_q as $w''_q;(q_i, \mathsf{any}, \Phi, \checkmark)$. Pick new C_q as C^1_q. Keep $\mathcal{V}'_p$, w, w'_p as before. We note $\|w''_q\| \mathbin{+\!\!+} \mathrm{sig}(C^1_q[[]@^{q_i}\mathsf{any}]) = \|w''_q;(q_i, \mathsf{any}, \Phi, \checkmark)\| + + \mathrm{sig}(C^1_q)$.

– $A = \iota$. If the value conforms to type ι then the reasoning is similar to the case for any. Instead we pick the new w''_q as $w''_q;(q_i, \iota, \Phi, \checkmark)$.

If the value does not conform then either we raise blame or return the value (both with a modified blame state). In the case of blame we apply (G-BLAME) with $\mathcal{E} = C^1_q[\mathcal{V}'_p]$. In the case that we do not raise blame then we pick the new w''_q as $w''_q;(q_i, \iota, \Phi, \mathsf{x})$. We instantiate the remaining variables as was the case for any and apply (G-GUARD-VAL). Again, we note that $\|w''_q\| \mathbin{+\!\!+} \mathrm{sig}(C^1_q[[]@^{q_i}\iota]) = \|w''_q;(q_i, \iota, \Phi, \mathsf{x})\| \mathbin{+\!\!+} \mathrm{sig}(C^1_p)$.

We must show that for resulting blame state Φ', then $\Phi' : p$ guards q. This equates to showing that when blaming q_i and adding it (and any prefix) to the blame state, that an equivalent p_i is already in the state. The traces w'_p and $w''_q;(q_i, \iota, \Phi, \mathsf{x})$ are equal up to the addition of q_i (modulo prefix substitution), and therefore denote equal calls to *blame*. As we know that

$p \not\leq q$ and $q \not\leq p$, then the sequence that blames w'_p does not affect the sequence that blames $w''_q;(q_i, \iota, \Phi, \mathrm{x})$. In effect, we could blame them in parallel and combine the result. The sequence of contract violations and blame calls denoted by the trace directly mirror each-other (up to prefix substitution), and therefore any q_i in the state is mirrored by an equivalent p_i, thus the resulting blame state is guarded for p and q.

The only way that the resulting blame state is not guarded is if q_i is added to the blame state with no corresponding p_i; given that the traces denote equal calls, this can only happen if there is some node compatible with $-p_i$, but there is no node compatible with $-q_i$. This situation is prevented by the definition of guarded on blame states.

– $A = A_1 \cap B_1$. We split the contract into two branch contracts.

Pick $C_q = C^1_q[([]@^{q_i \bullet \mathsf{left}^+_\cap[\mathsf{nil}]} A_1)@^{q_i \bullet \mathsf{right}^+_\cap[\mathsf{nil}]} B_1]$. All other existential variables are instantiated from the assumption.

We note that:

$$\mathrm{sig}(C^1_q[[]@^{q_i} A_1 \cap B_1]) = \mathrm{sig}(C^1_q[([]@^{q_i \bullet \mathsf{left}^+_\cap[\mathsf{nil}]} A_1)@^{q_i \bullet \mathsf{right}^+_\cap[\mathsf{nil}]}.B_1])$$

– $A = A_1 \cup B_1$. We split the contract into two branch contracts.

Pick $C_q = C^1_q[([]@^{q_i \bullet \mathsf{left}^+_\cup[\mathsf{nil}]} A_1)@^{q_i \bullet \mathsf{right}^+_\cup[\mathsf{nil}]} B_1]$. All other existential variables are instantiated from the assumption.

We note that:

$$\mathrm{sig}(C^1_q[[]@^{q_i} A_1 \cup B_1]) = \mathrm{sig}(C^1_q[([]@^{q_i \bullet \mathsf{left}^+_\cup[\mathsf{nil}]} A_1)@^{q_i \bullet \mathsf{right}^+_\cup[\mathsf{nil}]}.B_1])$$

Case 9. (G-GUARD-APP) The reasoning for this case is similar to (G-GUARD-VAL), except there may be applications of the wrap rule interspersing evaluation of the contracts on the function argument. We distinguish the cases when the argument contracts are all value contract contexts, or when there is a reduction.

- $C^2_q[C^1_p[]] = \mathcal{V}^2_q[\mathcal{V}^1_p[]]$ for some $\mathcal{V}^1, \mathcal{V}^2$. Performing a wrap essential shuffles the contracts from the function to the argument and context of the application. In such a case the equivalence between witness traces and contract context signatures is preserved.

  We distinguish the case when we perform "the last" wrap: $\mathcal{V}^2_q = []$ and $\mathcal{V}^1_p = []@^{p_i} A \to B$. In this case we cannot apply (G-GUARD-APP). However we may apply (G-GUARD-VAL-NEG) to N.

- Pick $w'_{-q} = w''_{-p} = \cdot$.
- Pick $C_{-p} = C^2_{-p}$.
- Pick $C'_{-q} = C^1_{-q}$.

We may apply (P-GUARD-TERM) to the codomain contracts, noting that $\mathrm{sig}(C^3_p) = \mathrm{sig}(C^4_{[p]})$ as $\mathrm{sig}_{\mathrm{cod}}(\Delta, \mathcal{V}'_p) = \cdot$ and $\mathrm{sig}_{\mathrm{cod}}(\Delta, \mathcal{V}_{[p]}) = \cdot$. We have:

- $C^2_{-p}[C^1_{-q}[\mathcal{V}[P^w]]]$ by (G-GUARD-VAL-NEG).
- $V\,(C^2_{-p}[C^1_{-q}[\mathcal{V}[P^w]]])$ by (G-APP).
- $C^4_q[C^3_p[(V\,(C^2_{-p}[C^1_{-q}[\mathcal{V}[P^w]]]))]]$ by (P-GUARD-TERM).

When we are not performing the "the last" wrap, then we are free to apply (P-GUARD-APP) again.

- $C^2_q[C^1_p[]]$ contains a reducible contract. When there is a reducible contract in the contract context for the argument we apply the same reasoning as the case (G-GUARD-VAL), localised to the argument.

Case 10. (G-GUARD-TERM-NEG), (G-GUARD-VAL-NEG), (G-GUARD-APP-NEG) The three cases are essentially identical to their non-negated counterparts, except we replace p for $-q$, and q for $-p$. Otherwise the reasoning is the same.

□

B.3 Contract Soundness

Theorem 4.3.1 (Contract Soundness)**.**

(a) $M@^{\pm\ell[P]}A \in [\![A]\!]^+$

(b) $K \circ \Box\,@^{\pm\ell[P]}B \in [\![B]\!]^-$

Proof. Translating from configurations into programs with contexts we have:

(a) $\mathcal{E}[(V@^{\pm\ell[P]}A)@^p A] \longrightarrow\!\!\!\!\!\rightarrow^* \langle \lightning p \rangle$ for all $\mathcal{E}$.

(b) $\mathcal{E}[(V@^p B)@^{\pm\ell[P]}B] \longrightarrow\!\!\!\!\!\rightarrow^* \langle \lightning -p \rangle$ for all V.

We observe that:

(a) $\mathcal{E}[(V@^{\pm\ell[P]}A)@^p A]\ :\ \pm\ell[P]$ guards p.

(b) $\mathcal{E}[(V@^{p}B)@^{\pm\ell[P]}B]$: $\mp\ell[P]$ guards $-p$.

We proceed with a proof by contradiction for each case.

(a) Assume $\mathcal{E}[(V@^{\pm\ell[P]}A)@^{p}A] \longrightarrow^{*} \langle \lightning\, p \rangle$. By induction on $\longrightarrow^{*}$ with Lemma B.2.2, then in the state Φ, where $\Phi \models p$, then $\Phi \models \pm\ell[P]$. By inversion of guards for the contract that implicated p we know that $\pm\ell[P]$ was implicated in the state *before*. However if this were true then the program would reduce to $\texttt{blame} + \ell$, and not evaluate the contract that implicated p, deriving a contradiction that p was implicated.

(b) We follow the same proof by contradiction, instead claiming that the program reduces to $\texttt{blame} - \ell$ before $-p$ is implicated.

□

Theorem 4.3.2 (Witness Soundness)**.**

(a) $M@^{-p \gg \mathsf{dom}_n} A \in [\![A]\!]^{+}_{p \gg \mathsf{cod}_n}$

(b) $K \circ \square\, @^{p \gg \mathsf{cod}_n} B \in [\![B]\!]^{-}_{-p \gg \mathsf{dom}_n}$

Proof. Translating from configurations into programs with contexts we have:

1. $\mathcal{E}[(V@^{-p \gg \mathsf{dom}_i} A)@^{p \gg \mathsf{cod}_i} A] \mathrel{-\!\!\!\rightarrow}^{*} \langle \lightning\, p \gg \mathsf{cod}_i \rangle$ for all $\mathcal{E}$.
2. $\mathcal{E}[(V@^{-p \gg \mathsf{dom}_i} B)@^{p \gg \mathsf{cod}_i} B] \mathrel{-\!\!\!\rightarrow}^{*} \langle \lightning\, p \gg \mathsf{dom}_i \rangle$ for all V.

We observe that:

1. $\mathcal{E}[(V@^{-p \gg \mathsf{dom}_i} A)@^{p \gg \mathsf{cod}_i} A]$: $-p \gg \mathsf{dom}_i$ guards $p \gg \mathsf{cod}_i$.
2. $\mathcal{E}[(V@^{-p \gg \mathsf{dom}_i} B)@^{p \gg \mathsf{cod}_i} B]$: $-p \gg \mathsf{cod}_i$ guards $p \gg \mathsf{dom}_i$.

We proceed with a proof by contradiction for each case.

1. Assume $\mathcal{E}[(V@^{-p \gg \mathsf{dom}_i} A)@^{p \gg \mathsf{cod}_i} A] \longrightarrow^{*} \langle \lightning\, p \gg \mathsf{cod}_i \rangle$. By induction on $\longrightarrow^{*}$ with Lemma B.2.2, then in the state Φ, where $\Phi \models p \gg \mathsf{cod}_i$, then $\Phi \models -p \gg \mathsf{dom}_i$. By inversion of guards for the contract that implicated $p \gg \mathsf{cod}_i$ we know that $-p \gg \mathsf{dom}_i$ was implicated in the state *before*. However we observe:

$$\mathit{compat}(-p \gg \mathsf{dom}_i, -(p \gg \mathsf{cod}_i))$$

As the nodes are compatible it cannot be the case that $p \gg \mathsf{cod}_i$ is ever assigned blame, contradicting the assumption that

$$\mathcal{E}[(V@^{-p \gg \mathsf{dom}_i} A)@^{p \gg \mathsf{cod}_i} A] \longrightarrow^{*} \langle \lightning\, p \gg \mathsf{cod}_i \rangle$$

2. We use similar reasoning to the previous case, arguing that the nodes are compatible and so only one can ever be implicated. By inversion of guards we see that it must be $-p \gg \mathsf{cod}_i$ and not $p \gg \mathsf{dom}_i$.

□

Appendix C

Monitoring Properties

C.1 Commuting

Definition C.1.1 (Node Abbreviation)**.** *We write $p\cdots$ for a blame node with zero or more branches stemming from parent p. We write $p\cdots_n$ to indicate that there are exactly n branches. As a consequence $p = p\cdots_0$.*

Definition C.1.2 (Root Equality for Blame States)**.** *We write $\Phi = \Phi'$ for equality between blame states, defined as:*

$$\Phi = \Phi' \stackrel{def}{=} \forall q.\ q \in \Phi \Leftrightarrow q \in \Phi'$$

We write $\Phi =_p \Phi'$ for root equality between blame states, defined as:

$$\Phi =_{\pm\ell[c_i/P]} \Phi' \stackrel{def}{=} \Phi = \Phi'$$

$$\Phi =_{\pm\ell[\mathsf{nil}]} \Phi' \stackrel{def}{=} \forall(P_1, q \neq \pm\ell[P_1]).\ (q \in \Phi \Leftrightarrow q \in \Phi') \wedge (\exists P.\ \pm\ell[P] \in \Phi \Leftrightarrow \exists P'.\ \pm\ell[P'] \in \Phi')$$

$$\Phi =_{p_1 \bullet d^{\pm}_{\diamond}[c_i/P]} \Phi' \stackrel{def}{=} \Phi = \Phi'$$

$$\Phi =_{p_1 \bullet d^{+}_{\cap}[\mathsf{nil}]} \Phi' \stackrel{def}{=} \forall(P_1, q \neq p_1 \bullet d^{+}_{\cap}[P_1]).\ (q \in \Phi \Leftrightarrow q \in \Phi') \wedge (\exists P.\ p_1 \bullet d^{+}_{\cap}[P] \in \Phi \Leftrightarrow \exists P'.\ p_1 \bullet d^{+}_{\cap}[P'] \in \Phi')$$

$$\Phi =_{p_1 \bullet d^{+}_{\cup}[\mathsf{nil}]} \Phi' \stackrel{def}{=} \forall(P_1, q \neq p_1 \bullet d^{+}_{\cup}[P_1]).\ (q \in \Phi \Leftrightarrow q \in \Phi') \wedge (\exists P.\ p_1 \bullet d^{+}_{\cup}[P] \in \Phi \Leftrightarrow \exists P'.\ p_1 \bullet d^{+}_{\cup}[P'] \in \Phi')$$

$$\Phi =_{p_1 \bullet d^{-}_{\cup}[\mathsf{nil}]} \Phi' \stackrel{def}{=} \forall(P_1, q \neq p_1 \bullet d^{-}_{\cup}[P_1]).\ (q \in \Phi \Leftrightarrow q \in \Phi') \wedge \forall i.\ (\exists c_i/P.\ p_1 \bullet d^{-}_{\cup}[c_i/P] \in \Phi \Leftrightarrow \exists c'_i/P'.\ p_1 \bullet d^{-}_{\cup}[c'_i/P'] \in \Phi')$$

$$\Phi =_{p_1 \bullet d^{-}_{\cap}[\mathsf{nil}]} \Phi' \stackrel{def}{=} \forall(P_1, q \neq p_1 \bullet d^{-}_{\cap}[P_1]).\ (q \in \Phi \Leftrightarrow q \in \Phi') \wedge \forall i.\ (\exists c_i/P.\ p_1 \bullet d^{-}_{\cap}[c_i/P] \in \Phi \Leftrightarrow \exists c'_i/P'.\ p_1 \bullet d^{-}_{\cap}[c'_i/P'] \in \Phi')$$

We motivate Definition C.1.2 as follows. We use the definition $\Phi =_p \Phi'$when considering whether blame for branch nodes with parent p commute. We observe that the resulting blame states may not be *exactly* equal. This occurs when p has an empty blame path and resolving blame for the children of p will hoist blame paths. In this case the order of the nodes can affect which path is chosen to hoist. With this definition we show that the choice does not matter.

When p does not have an empty path we expect the resulting blame states to match because we do not hoist paths and always blame p directly.

When p is a root node with an empty path the paths in the blame state do not matter. We have already assigned blame to a root node with same label. No further blame assignment depends upon the path.

When p is a branch node with an empty path we consider the polarity and branch type. We observe that any sequence of nested intersection or union (denoted by empty paths) will always have the same polarity, this is why the positive intersection and union cases, and negative intersection and union cases, match.

In the positive case the paths may differ, but we note that paths do not matter both for positive intersection blame (we always resolve), and positive union blame (there is no constraint on paths).

In the negative case the paths must start with a matching elimination context. For negative union this does not matter, but for negative intersection we inspect the first element. However from the definition we are guaranteed a path that will start with a matching element, so the choice does not matter and consequently any nested intersection that contains p will match in either Φ or Φ'.

To clarify presentation we assume a version of $assign(p, \Phi)$ that only returns a modified blame state (and not a truth value).

Definition C.1.3 (Assignment without judgement)**.**

$$\mathsf{assign} \stackrel{def}{=} \mathsf{fst} \circ \mathsf{assign}$$

It should be clear that the truth value is not important because the blame state determines the truth value; we can reconstruct the truth value by looking at the presence or absence of a root in the resulting blame state. For Lemma C.1.4 and Lemma C.1.5 we use the definition of assign given in Definition C.1.3.

The following lemma states that blame assignment commutes for blame nodes that are in the same branch of an intersection or union.

Lemma C.1.4 (Blame Assignment Commutes for Different Paths)**.**

$$assign(p \bullet d^{\pm}_{\diamond}[P], assign(p \bullet d^{\pm}_{\diamond}[P'], \Phi)) = assign(p \bullet d^{\pm}_{\diamond}[P'], assign(p \bullet d^{\pm}_{\diamond}[P], \Phi))$$

Proof. By induction on the structure of blame nodes. We observe that the two nodes have the same polarity up to the root, so they cannot affect assignment for each other. We also observe that they will have the same branch directions (all the way up), so they cannot affect resolution for each other. The assignments are therefore independent and commute. □

The following lemma states that blame assignment commutes for blame nodes that are in different branches of the same intersection or union.

Lemma C.1.5 (Blame Assignment Commutes for Different Branches)**.**

$$\begin{aligned} &assign(p \bullet \mathsf{left}^{\pm}_{\diamond}[P] \cdots_m, assign(p \bullet \mathsf{right}^{\pm}_{\diamond}[P'] \cdots_n, \Phi)) =_p \\ &assign(p \bullet \mathsf{right}^{\pm}_{\diamond}[P'] \cdots_n, assign(p \bullet \mathsf{left}^{\pm}_{\diamond}[P] \cdots_m, \Phi)) \end{aligned}$$

Proof. By simultaneous induction on the number of branches denoted by m and n.

We split the proof into cases for positive and negative, intersection and union. For brevity we let $L = p \bullet \mathsf{left}^{\pm}_{\diamond}[P] \cdots_m$ and $R = p \bullet \mathsf{right}^{\pm}_{\diamond}[P'] \cdots_n$.

Case 1. $+$ and $\cap$.

$$\begin{aligned} &assign(p \bullet \mathsf{left}^{+}_{\cap}[P] \cdots_m, assign(p \bullet \mathsf{right}^{+}_{\cap}[P'] \cdots_n, \Phi)) =_p \\ &assign(p \bullet \mathsf{right}^{+}_{\cap}[P'] \cdots_n, assign(p \bullet \mathsf{left}^{+}_{\cap}[P] \cdots_m, \Phi)) \end{aligned}$$

- Suppose that blame is not assigned for $p \bullet \mathsf{right}^{+}_{\cap}[P'] \cdots_n$ because there is some $q \in \Phi$ such that $compat(q, -(p \bullet \mathsf{right}^{+}_{\cap}[P'] \cdots_n))$.

  Then $assign(p \bullet \mathsf{right}^{+}_{\cap}[P'] \cdots_n, \Phi) = \Phi$.

  Let $assign(p \bullet \mathsf{left}^{+}_{\cap}[P] \cdots_m, \Phi) = \Phi \cup \Phi'$. Then we have:

  1. $$\begin{aligned} &assign(p \bullet \mathsf{left}^{+}_{\cap}[P] \cdots_m, assign(p \bullet \mathsf{right}^{+}_{\cap}[P'] \cdots_n, \Phi)) \\ &= assign(p \bullet \mathsf{right}^{+}_{\cap}[P'] \cdots_n, assign(p \bullet \mathsf{left}^{+}_{\cap}[P] \cdots_m, \Phi)) \end{aligned}$$

2. $$assign(p \bullet \mathsf{left}^{+}_{\cap}[P] \cdots_{m}, \Phi) = assign(p \bullet \mathsf{right}^{+}_{\cap}[P'] \cdots_{n}, \Phi \cup \Phi')$$

3. $\Phi \cup \Phi' = assign(p \bullet \mathsf{right}^{+}_{\cap}[P'] \cdots_{n}, \Phi \cup \Phi')$

4. $\Phi \cup \Phi' = \Phi \cup \Phi'$ as $q \in (\Phi \cup \Phi')$.

- Suppose that blame is not assigned for $p \bullet \mathsf{left}^{+}_{\cap}[P] \cdots_{m}$ because there is some $q \in \Phi$ such that $compat(q, -(p \bullet \mathsf{left}^{+}_{\cap}[P] \cdots_{m}))$.

  Then $assign(p \bullet \mathsf{left}^{+}_{\cap}[P] \cdots_{m}, \Phi) = \Phi$.

  Let $assign(p \bullet \mathsf{right}^{+}_{\cap}[P'] \cdots_{n}, \Phi) = \Phi \cup \Phi'$. Then we have:

  1. $$\begin{aligned} &assign(p \bullet \mathsf{left}^{+}_{\cap}[P] \cdots_{m}, assign(p \bullet \mathsf{right}^{+}_{\cap}[P'] \cdots_{n}, \Phi)) \\ &= assign(p \bullet \mathsf{right}^{+}_{\cap}[P'] \cdots_{n}, assign(p \bullet \mathsf{left}^{+}_{\cap}[P] \cdots_{m}, \Phi)) \end{aligned}$$

  2. $$assign(p \bullet \mathsf{left}^{+}_{\cap}[P] \cdots_{m}, \Phi \cup \Phi') = assign(p \bullet \mathsf{right}^{+}_{\cap}[P'] \cdots_{n}, \Phi)$$

  3. $assign(p \bullet \mathsf{left}^{+}_{\cap}[P] \cdots_{m}, \Phi \cup \Phi') = \Phi \cup \Phi'$

  4. $\Phi \cup \Phi' = \Phi \cup \Phi'$ as $q \in (\Phi \cup \Phi')$.

In the following cases we know that blame is assigned to both nodes. We consider the lengths of m and n.

- m and n are non-zero. The proof proceeds by considering whether blame resolves for the left and right, observing that resolution is independent because both branches do not share a parent (as m and n are greater than 0). That is, we can lift additions L and R to the blame state out because they do not affect resolution for the other branch.

  – Both resolve.

    1. $$\begin{aligned} &assign(p \bullet \mathsf{left}^{+}_{\cap}[P] \cdots_{m}, assign(p \bullet \mathsf{right}^{+}_{\cap}[P'] \cdots_{n}, \Phi)) \\ &=_{p} assign(p \bullet \mathsf{right}^{+}_{\cap}[P'] \cdots_{n}, assign(p \bullet \mathsf{left}^{+}_{\cap}[P] \cdots_{m}, \Phi)) \end{aligned}$$

    2. $assign(L, assign(p \bullet \mathsf{right}^{+}_{\cap}[P'] \cdots_{n-1}, \Phi \cup \{R\})) =_{p} assign(R, assign(p \bullet \mathsf{left}^{+}_{\cap}[P] \cdots_{m-1}, \Phi \cup \{L\}))$

    3. $assign(L, assign(p \bullet \mathsf{right}^{+}_{\cap}[P'] \cdots_{n-1}, \Phi)) \cup \{R\} =_{p} assign(R, assign(p \bullet \mathsf{left}^{+}_{\cap}[P] \cdots_{m-1}, \Phi)) \cup \{L\}$

4. $$assign(p \bullet \mathsf{right}^+_\cap[P'] \cdots_{n-1}, assign(L, \Phi)) \cup \{R\}$$
$$=_p assign(p \bullet \mathsf{left}^+_\cap[P] \cdots_{m-1}, assign(R, \Phi)) \cup \{L\}$$
by IH.
5. $assign(p \bullet \mathsf{right}^+_\cap[P'] \cdots_{n-1}, assign(p \bullet \mathsf{left}^+_\cap[P] \cdots_{m-1}, \Phi \cup \{L\})) \cup \{R\} =_p$ $assign(p \bullet \mathsf{left}^+_\cap[P] \cdots_{m-1}, assign(p \bullet \mathsf{right}^+_\cap[P'] \cdots_{n-1}, \Phi \cup \{R\})) \cup \{L\}$ as L and R both resolve.
6. $assign(p \bullet \mathsf{right}^+_\cap[P'] \cdots_{n-1}, assign(p \bullet \mathsf{left}^+_\cap[P] \cdots_{m-1}, \Phi)) \cup \{L\} \cup \{R\} =_p$ $assign(p \bullet \mathsf{left}^+_\cap[P] \cdots_{m-1}, assign(p \bullet \mathsf{right}^+_\cap[P'] \cdots_{n-1}, \Phi)) \cup \{R\} \cup \{L\}$ as L and R do not share a parent.
7. Apply IH.

– L does not resolve, R does.

1. $$assign(p \bullet \mathsf{left}^+_\cap[P] \cdots_m, assign(p \bullet \mathsf{right}^+_\cap[P'] \cdots_n, \Phi))$$
$$=_p assign(p \bullet \mathsf{right}^+_\cap[P'] \cdots_n, assign(p \bullet \mathsf{left}^+_\cap[P] \cdots_m, \Phi))$$
2. $assign(L, assign(p \bullet \mathsf{right}^+_\cap[P'] \cdots_{n-1}, \Phi \cup \{R\})) =_p assign(R, \Phi \cup \{L\})$
3. $assign(L, assign(p \bullet \mathsf{right}^+_\cap[P'] \cdots_{n-1}, \Phi)) \cup \{R\} =_p assign(R, \Phi) \cup \{L\}$
4. $assign(p \bullet \mathsf{right}^+_\cap[P'] \cdots_{n-1}, assign(L, \Phi)) \cup \{R\} =_p assign(R, \Phi) \cup \{L\}$
by IH on LHS.
5. $assign(p \bullet \mathsf{right}^+_\cap[P'] \cdots_{n-1}, \Phi) \cup \{L\} \cup \{R\} =_p assign(R, \Phi) \cup \{L\}$
as L does not resolve and does not affect resolution for R.
6. $$assign(p \bullet \mathsf{right}^+_\cap[P'] \cdots_{n-1}, \Phi) \cup \{R\} \cup \{L\}$$
$$=_p assign(p \bullet \mathsf{right}^+_\cap[P'] \cdots_{n-1}, \Phi) \cup \{L\} \cup \{R\}$$
as R resolves.

– R does not resolve, L does. Same as previous case.

– L does not resolve and R does not resolve.

1. $$assign(p \bullet \mathsf{left}^+_\cap[P] \cdots_m, assign(p \bullet \mathsf{right}^+_\cap[P'] \cdots_n, \Phi))$$
$$=_p assign(p \bullet \mathsf{right}^+_\cap[P'] \cdots_n, assign(p \bullet \mathsf{left}^+_\cap[P] \cdots_m, \Phi))$$
2. $assign(L, \Phi \cup \{R\}) =_p assign(R, \Phi \cup \{L\})$
3. $assign(L, \Phi) \cup \{R\} =_p assign(R, \Phi) \cup \{L\}$
4. $\Phi \cup \{L\} \cup \{R\} =_p \Phi \cup \{R\} \cup \{L\}$ as neither resolve in Φ.

- $n = 0$ and $m > 0$, then $R = p \bullet \mathsf{right}^+_\cap[P']$. We consider whether L resolves or not (which is independent of R).

  - L resolves.

    1. $$\begin{aligned} &\mathit{assign}(p \bullet \mathsf{left}^+_\cap[P] \cdots_m, \mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \Phi)) \\ &=_p \mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \mathit{assign}(p \bullet \mathsf{left}^+_\cap[P] \cdots_m, \Phi)) \end{aligned}$$

    2. $$\begin{aligned} &\mathit{assign}(p \bullet \mathsf{left}^+_\cap[P] \cdots_m, \mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \Phi)) \\ &=_p \mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \mathit{assign}(p \bullet \mathsf{left}^+_\cap[P] \cdots_{m-1}, \Phi \cup \{L\})) \end{aligned}$$

       by resolving 1 step on RHS.

    3. $$\begin{aligned} &\mathit{assign}(p \bullet \mathsf{left}^+_\cap[P] \cdots_m, \mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \Phi)) \\ &=_p \mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \mathit{assign}(p \bullet \mathsf{left}^+_\cap[P] \cdots_{m-1}, \Phi)) \cup \{L\} \end{aligned}$$

       as L is independent of R

    4. $$\begin{aligned} &\mathit{assign}(p \bullet \mathsf{left}^+_\cap[P] \cdots_m, \mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \Phi)) \\ &=_p \mathit{assign}(p \bullet \mathsf{left}^+_\cap[P] \cdots_{m-1}, \mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \Phi)) \cup \{L\} \end{aligned}$$

       by IH on RHS.

    5. $$\begin{aligned} &\mathit{assign}(p \bullet \mathsf{left}^+_\cap[P] \cdots_m, \Phi \cup \Phi_R) \\ &=_p \mathit{assign}(p \bullet \mathsf{left}^+_\cap[P] \cdots_{m-1}, \Phi \cup \Phi_R) \cup \{L\} \end{aligned}$$

       where $\mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \Phi) = \Phi \cup \Phi_R$.

    6. $$\begin{aligned} &\mathit{assign}(p \bullet \mathsf{left}^+_\cap[P] \cdots_{m-1}, \Phi \cup \Phi_R) \cup \{L\} \\ &=_p \mathit{assign}(p \bullet \mathsf{left}^+_\cap[P] \cdots_{m-1}, \Phi \cup \Phi_R) \cup \{L\} \end{aligned}$$

       by assumption that L resolves in Φ, and Φ_R does not prevent resolution.

  - L does not resolve.

    1. $$\begin{aligned} &\mathit{assign}(p \bullet \mathsf{left}^+_\cap[P] \cdots_m, \mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \Phi)) \\ &=_p \mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \mathit{assign}(p \bullet \mathsf{left}^+_\cap[P] \cdots_m, \Phi)) \end{aligned}$$

2.
$$\begin{aligned}&\mathit{assign}(p \bullet \mathsf{left}^+_\cap[P] \cdots_m, \mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \Phi))\\ &=_p \mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \Phi \cup \{L\})\end{aligned}$$
as L is assigned blame but does not resolve.

3.
$$\begin{aligned}&\mathit{assign}(p \bullet \mathsf{left}^+_\cap[P] \cdots_m, \mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \Phi))\\ &=_p \mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \Phi) \cup \{L\}\end{aligned}$$
as L does not affect R (they share no parent).

4.
$$\begin{aligned}&\mathit{assign}(p \bullet \mathsf{left}^+_\cap[P] \cdots_m, \Phi \cup \Phi_R)\\ &=_p \Phi \cup \Phi_R \cup \{L\}\end{aligned}$$
where $\mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \Phi) = \Phi \cup \Phi_R$.

5. $\Phi \cup \Phi_R \cup \{L\} =_p \Phi \cup \Phi_R \cup \{L\}$ observing that L does not resolve even with the addition of Φ_R because it contains no nodes with a matching parent, and all nodes have the same polarity as L.

- $m = 0$ and $n > 0$. Same as previous case.
- $m = n = 0$.

1.
$$\begin{aligned}&\mathit{assign}(p \bullet \mathsf{left}^+_\cap[P], \mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \Phi))\\ &=_p \mathit{assign}(p \bullet \mathsf{right}^+_\cap[P'], \mathit{assign}(p \bullet \mathsf{left}^+_\cap[P], \Phi))\end{aligned}$$
Blame for positive intersection branches always resolves, so both sides resolve blame to the parent p. By Lemma C.1.4 the order of the blame for the parents does not matter.

Case 2. $-$ and $\cap$.
$$\begin{aligned}&\mathit{assign}(p \bullet \mathsf{left}^-_\cap[P] \cdots_m, \mathit{assign}(p \bullet \mathsf{right}^-_\cap[P'] \cdots_n, \Phi))\\ &=_p \mathit{assign}(p \bullet \mathsf{right}^-_\cap[P'] \cdots_n, \mathit{assign}(p \bullet \mathsf{left}^-_\cap[P] \cdots_m, \Phi))\end{aligned}$$

The proof is similar to the previous case. The only significant difference is the base case, that is when: $m = n = 0$ and L and R are assigned blame.

We split the case depending on whether the nodes comes from matching elimination contexts (and may affect each other), or if they do not (and are therefore independent).

- $\mathit{elim}(P, P')$

1. $$\begin{aligned}&\mathit{assign}(p \bullet \mathsf{left}^-_\cap[c_i/P], \mathit{assign}(p \bullet \mathsf{right}^-_\cap[c'_i/P'], \Phi))\\ =_p\ &\mathit{assign}(p \bullet \mathsf{right}^-_\cap[c'_i/P'], \mathit{assign}(p \bullet \mathsf{left}^-_\cap[c_i/P], \Phi))\end{aligned}$$

   It should be clear that blame will always propagate to the parent because a matching elimination context has been found. It remains to show that blame assignment for the parent remains the same.

   We note that the latter node is chosen to resolve the blame upwards, so we must show that the order does not matter. That is, $\mathit{parent}(L)$ and $\mathit{parent}(R)$ are assigned blame in unison. By case analysis there are two cases:

   - $\mathit{parent}(L) = \mathit{parent}(R) = p$: when the path of the parent p is not empty. This case is immediate as the parent node is the same regardless of order.
   - When the path of the parent is empty and either P or P' are hoisted upwards. The only way this could matter is if there was a (complemented) path compatible with one, but not the other. This is not possible; there is no such node that could be compatible with either.
     We observe that if there were, it would have to come from a node that was also hoisted upwards, however in this case then there would be a (complemented) node compatible with L or R, which we know not to be true by assumption.

- $\neg \mathit{elim}(P, P')$

  1. $$\begin{aligned}&\mathit{assign}(p \bullet \mathsf{left}^-_\cap[c_i/P], \mathit{assign}(p \bullet \mathsf{right}^-_\cap[c'_j/P'], \Phi))\\ =_p\ &\mathit{assign}(p \bullet \mathsf{right}^-_\cap[c'_j/P'], \mathit{assign}(p \bullet \mathsf{left}^-_\cap[c_i/P], \Phi))\end{aligned}$$

     Blame resolves for the two nodes independently. They have matching parents therefore any blame that extends up cannot cause interference between the two nodes. The most interesting case is when they both resolve, in which case we appeal to Lemma C.1.4.

Case 3. $+$ and $\cup$.

$$\begin{aligned}&\mathit{assign}(p \bullet \mathsf{left}^+_\cup[P]\cdots_m, \mathit{assign}(p \bullet \mathsf{right}^+_\cup[P']\cdots_n, \Phi))\\ =_p\ &\mathit{assign}(p \bullet \mathsf{right}^+_\cup[P']\cdots_n, \mathit{assign}(p \bullet \mathsf{left}^+_\cup[P]\cdots_m, \Phi))\end{aligned}$$

The case is similar to negative intersection. The only difference is that we do not need to distinguish the cases for $\mathit{elim}(P, P')$ and $\neg\mathit{elim}(P, P')$ as this constraint does not appear in blame resolution.

Case 4. $-$ and $\cup$.

$$assign(p \bullet \mathsf{left}^{-}_{\cup}[P] \cdots_m, assign(p \bullet \mathsf{right}^{-}_{\cup}[P'] \cdots_n, \Phi))$$
$$=_p assign(p \bullet \mathsf{right}^{-}_{\cup}[P'] \cdots_n, assign(p \bullet \mathsf{left}^{-}_{\cup}[P] \cdots_m, \Phi))$$

This case is identical to positive intersection as blame resolution for negative union branches is the same as positive intersection branches.

□

Lemma C.1.6 (Branch Contracts Commute)**.** *Ordering of intersection and union contracts does not affect blame for the parent.*

1. $\langle \Phi, \Delta, K, (V@^{p \bullet \mathsf{left}^{\pm}_{\diamond}[\mathsf{nil}]} A)@^{p \bullet \mathsf{right}^{\pm}_{\diamond}[\mathsf{nil}]} B \rangle \longrightarrow^* \langle \lightning \pm p \rangle$

   iff $\langle \Phi, \Delta, K, (V@^{p \bullet \mathsf{right}^{\pm}_{\diamond}[\mathsf{nil}]} B)@^{p \bullet \mathsf{left}^{\pm}_{\diamond}[\mathsf{nil}]} A \rangle \longrightarrow^* \langle \lightning \pm p \rangle$

2. $\langle \Phi, \Delta, K, (V@^{p \bullet \mathsf{left}^{\pm}_{\diamond}[\mathsf{nil}]} A)@^{p \bullet \mathsf{right}^{\pm}_{\diamond}[\mathsf{nil}]} B \rangle \longrightarrow^* \langle \lightning \mp p \rangle$

   iff $\langle \Phi, \Delta, K, (V@^{p \bullet \mathsf{right}^{\pm}_{\diamond}[\mathsf{nil}]} B)@^{p \bullet \mathsf{left}^{\pm}_{\diamond}[\mathsf{nil}]} A \rangle \longrightarrow^* \langle \lightning \mp p \rangle$

Proof.

- We first observe that during execution of a program the blame state Φ' is determined by a sequence of blame assignments. This sequence can be characterised by a witness trace of the form:

  $$w; (p_i, \iota, \Phi_i, \mathsf{x})$$

  where all witnesses fail, and Φ_i was the blame state at the failure. The blame state can be constructed by traversing the trace and assigning blame to each p_i in state Φ_i, with an initial state $\emptyset$.

- We can group contracts from an intersection or union into a contract context with a common parent. For example: $(V@^{p \bullet \mathsf{left}^{+}_{\cap}[\mathsf{nil}]} A)@^{p \bullet \mathsf{right}^{+}_{\cap}[\mathsf{nil}]} B = C_p[V]$, and $(V@^{p \bullet \mathsf{right}^{+}_{\cap}[\mathsf{nil}]} B)@^{p \bullet \mathsf{left}^{+}_{\cap}[\mathsf{nil}]} A = C'_p[V]$. Consider the forms during evaluation that such contexts take, they are:

  1. $C_p[M]$ where M is a non-value.
  2. $C_p[\mathcal{V}[P^{w \,+\!+\, w'_p}]]$ where the contracts in the context are under evaluation.
  3. $C'_p[\mathcal{V}_p[(V\,(C_{-p}[\mathcal{V}[P^{w \,+\!+\, w'_{-p}}]]))]]$ where we are wrapping function contracts that were in the intersection or union, alternating between performing a wrap, and evaluating a contract in the domain.

Consider when the contexts denoted by C and $\mathcal{V}$ are commuted in some form to C' and $\mathcal{V}'$.

1. In this case the witness trace generated by evaluating M is unaffected.
2. In this case we observe the witness trace for the program will be $w \mathbin{+\!\!+} w'_p$ for context C, and will be $w \mathbin{+\!\!+} w''_p$ for context C'. We note that w'_p and w''_p will be in a different order, but will contain the same set of witnesses. That is, there is a re-ordering of w'_p and w''_p such that they correspond to the same trace.
3. This case is similar to the previous, except the witness trace is for $-p$ rather than p. The same reasoning applies: the witnesses may be in a different order when commuting the contracts, but the set of witnesses remain the same.

- When commuting intersection or union contracts we observe that the witness traces induced by the two programs will be of the form:

$$\cdots w \mathbin{+\!\!+} w_{\pm p} \mathbin{+\!\!+} w_1 \cdots$$

$$\cdots w \mathbin{+\!\!+} w'_{\pm p} \mathbin{+\!\!+} w_1 \cdots$$

where the *set* of witness in the $w_{\pm p}$ and $w'_{\pm p}$ are the same, and that all nodes in the sub-traces are of the form $\pm p \bullet d^{\pm}_{\diamond}[P] \cdots_n$. By appealing to Lemma C.1.5 we argue that if one of the sub-traces is sufficient to assign blame to $\pm p$, then the other sub-trace will also assign blame to $\pm p$ as it blames the same set of nodes but in a different order. As all the nodes are children of $\pm p$ the order does not matter when assigning blame to $\pm p$. That is, if Φ_1 is the blame state the results from blaming trace $w_{\pm p}$, and Φ_2 is the blame state the results from blaming trace $w'_{\pm p}$, then $\Phi_1 =_{\pm p} \Phi_2$.

□

Lemma C.1.7 (Satisfaction Commutes)**.** *Commuting of* $\cap$ *and* $\cup$ *extends to satisfaction.*

1. $V \in [\![A \diamond B]\!]^+$ *iff* $V \in [\![B \diamond A]\!]^+$
2. $K \in [\![A \diamond B]\!]^-$ *iff* $K \in [\![B \diamond A]\!]^-$

Proof. Proof of 1 and 2 appeals to the same case in Lemma C.1.6. The proof of each case is nearly identical; we consider the positive case.

- We are required to show that:

  $\langle \Phi, \Delta, K, (V @^{p \bullet \mathsf{left}^+_\diamond[\mathsf{nil}]} A) @^{p \bullet \mathsf{right}^+_\diamond[\mathsf{nil}]} B \rangle \longrightarrow^* \langle \lightning + p \rangle$

  iff

  $\langle \Phi, \Delta, K, (V @^{p \bullet \mathsf{left}^+_\diamond[\mathsf{nil}]} B) @^{p \bullet \mathsf{right}^+_\diamond[\mathsf{nil}]} A \rangle \longrightarrow^* \langle \lightning + p \rangle$

- By Lemma C.1.6(1) we have:

  $\langle \Phi, \Delta, K, (V @^{p \bullet \mathsf{left}^+_\diamond[\mathsf{nil}]} A) @^{p \bullet \mathsf{right}^+_\diamond[\mathsf{nil}]} B \rangle \longrightarrow^* \langle \lightning + p \rangle$

  iff

  $\langle \Phi, \Delta, K, (V @^{p \bullet \mathsf{right}^+_\diamond[\mathsf{nil}]} B) @^{p \bullet \mathsf{left}^+_\diamond[\mathsf{nil}]} A \rangle \longrightarrow^* \langle \lightning + p \rangle$

- By the freshness constraint on p we know that this is the only contract that mentions p. Informally we argue that we may flip the direction in both branch nodes without affecting semantics. This is justified as our definition of *blame* is parametric in branch direction d. Specifically:

  $\langle \Phi, \Delta, K, (V @^{p \bullet \mathsf{left}^+_\diamond[\mathsf{nil}]} B) @^{p \bullet \mathsf{right}^+_\diamond[\mathsf{nil}]} A \rangle \longrightarrow^* \langle \lightning + p \rangle$

  iff

  $\langle \Phi, \Delta, K, (V @^{\mathsf{flip}(p \bullet \mathsf{left}^+_\diamond[\mathsf{nil}])} B) @^{\mathsf{flip}(p \bullet \mathsf{right}^+_\diamond[\mathsf{nil}])} A \rangle \longrightarrow^* \langle \lightning + p \rangle$

□

C.2 Sound Monitoring

Lemma C.2.1. $V : \iota \Rightarrow V \in [\![\iota]\!]^+$

Proof. We wish to show that $\langle \Phi, \Delta, K, V @^p \iota \rangle \not\longrightarrow^* \langle \lightning\, p \rangle$ for all K.

- By assumption $\langle \Phi, \Delta, K, V @^p \iota \rangle \longrightarrow \langle \Phi, \Delta, K, V \rangle$
- By assumption $p \# \langle \Phi, \Delta, K, V \rangle$.
- $\langle \Phi, \Delta, K, V \rangle \not\longrightarrow^* \langle \lightning\, p \rangle$ by Lemma 4.2.3.
- Therefore $\langle \Phi, \Delta, K, V @^p \iota \rangle \longrightarrow \langle \Phi, \Delta, K, V \rangle \not\longrightarrow^* \langle \lightning\, p \rangle$ as required.

□

Lemma C.2.2. $K \in [\![\iota]\!]^-$

Proof. We wish to show that $\langle \Phi, \Delta, K, V@^p \iota \rangle \rightarrowtail^* \langle \lightning -p \rangle$ for all V.

- Suppose that the contract fails and terminates, then it does so with blame on p and can therefore never implicate $-p$.
- If evaluation of the contract does not raise blame then we have:

  $\langle \Phi, \Delta, K, V@^p \iota \rangle \longrightarrow \langle \Phi \cup \Phi', \Delta, K, V \rangle$

  where Φ' includes p and some possible prefixes (but not $-p$). By assumption we know that $p \# \langle \Phi, \Delta, K, V \rangle$, and by Lemma 4.2.3 the resulting term can never implicate $-p$. The state Φ' is irrelevant because there will be no subcontract mentioning $-p$ with which to interact.

□

Lemma C.2.3. $V \in [\![\mathsf{any}]\!]^+$

Proof. We wish to show that $\langle \Phi, \Delta, K, V@^p \mathsf{any} \rangle \rightarrowtail^* \langle \lightning p \rangle$ for all K.

- By definition of reduction $\langle \Phi, \Delta, K, V@^p \mathsf{any} \rangle \longrightarrow \langle \Phi, \Delta, K, V \rangle$
- By assumption $p \# \langle \Phi, \Delta, K, V \rangle$.
- $\langle \Phi, \Delta, K, V \rangle \rightarrowtail^* \langle \lightning p \rangle$ by Lemma 4.2.3.
- Therefore $\langle \Phi, \Delta, K, V@^p \mathsf{any} \rangle \longrightarrow \langle \Phi, \Delta, K, V \rangle \rightarrowtail^* \langle \lightning p \rangle$ as required.

□

Lemma C.2.4. $K \in [\![\mathsf{any}]\!]^-$

Proof. We wish to show that $\langle \Phi, \Delta, K, V@^p \mathsf{any} \rangle \rightarrowtail^* \langle \lightning -p \rangle$ for all V.

- By definition of reduction $\langle \Phi, \Delta, K, V@^p \mathsf{any} \rangle \longrightarrow \langle \Phi, \Delta, K, V \rangle$
- By assumption $p \# \langle \Phi, \Delta, K, V \rangle$.
- $\langle \Phi, \Delta, K, V \rangle \rightarrowtail^* \langle \lightning p \rangle$ by Lemma 4.2.3.
- Therefore $\langle \Phi, \Delta, K, V@^p \mathsf{any} \rangle \longrightarrow \langle \Phi, \Delta, K, V \rangle \rightarrowtail^* \langle \lightning -p \rangle$ as required.

□

Lemma C.2.5. $\forall N \in [\![A]\!]^+_{p \gg \mathsf{cod}_i}.\ V\,N \in [\![B]\!]^+_{p \gg \mathsf{cod}_i} \wedge$
$\forall K \in [\![B]\!]^-_{-p \gg \mathsf{dom}_i}.\ K \circ V\,\Box \in [\![A]\!]^-_{-p \gg \mathsf{dom}_i}$
$\Rightarrow V \in [\![A \to B]\!]^+_p$

Proof. We wish to show that $\langle \Phi, \Delta, K, V@^p A \to B\rangle \not\longrightarrow^* \langle \lightning\, p\rangle$ for all K. Proceed with a proof by contradiction.

- Assume $\langle \Phi, \Delta, K, V@^p A \to B\rangle \longrightarrow^* \langle \lightning\, p\rangle$ for some K, p.
- By assumption we know:

  $\langle \Phi, \Delta, K, V@^p A \to B\rangle \longrightarrow^* \langle \Phi', \Delta', K', (V\,(W@^{-p \gg \mathsf{dom}_i} A))@^{p \gg \mathsf{cod}_i} B\rangle \longrightarrow^* \langle \lightning\, p\rangle$ for some K', W, i
- There are two cases

  1. $\langle K', (V\,(W@^{-p \gg \mathsf{dom}_i} A))@^{p \gg \mathsf{cod}_i} B\rangle \longrightarrow^* \langle \lightning\, p \gg \mathsf{dom}_i\rangle$
     - Expanding reductions for (1) we have:
       $\langle K', (V\,(W@^{-p \gg \mathsf{dom}_i} A))@^{p \gg \mathsf{cod}_i} B\rangle \longrightarrow$
       $\langle K' \circ \Box\,@^{p \gg \mathsf{cod}_i} B, (V\,(W@^{-p \gg \mathsf{dom}_i} A))\rangle \longrightarrow$
       $\langle K' \circ \Box\,@^{p \gg \mathsf{cod}_i} B \circ V\,\Box, W@^{-p \gg \mathsf{dom}_i} A\rangle \longrightarrow^* \langle \lightning\, p \gg \mathsf{dom}_i\rangle$
     - By Theorem 4.3.2 we know that $K' \circ \Box\,@^{p \gg \mathsf{cod}_i} B \in [\![B]\!]^-_{-p \gg \mathsf{dom}_i}$.
     - By assumption $K' \circ \Box\,@^{p \gg \mathsf{cod}_i} B \circ V\,\Box \in [\![A]\!]^-_{-p \gg \mathsf{dom}_i}$.
     - Therefore $\langle K' \circ \Box\,@^{p \gg \mathsf{cod}_i} B \circ V\,\Box, W@^{-p \gg \mathsf{dom}_i} A\rangle \not\longrightarrow^* \langle \lightning\, p \gg \mathsf{dom}_i\rangle$ for all W. We derive a contradiction with (1) that witnesses blame.
  2. $\langle K', (V\,(W@^{-p \gg \mathsf{dom}_i} A))@^{p \gg \mathsf{cod}_i} B\rangle \longrightarrow^* \langle \lightning\, p \gg \mathsf{cod}_i\rangle$
     - By Theorem 4.3.2 we know that $W@^{-p \gg \mathsf{dom}_i} A \in [\![A]\!]^+_{p \gg \mathsf{cod}_i}$.
     - By assumption $V\,(W@^{-p \gg \mathsf{dom}_i} A) \in [\![B]\!]^+_{p \gg \mathsf{cod}_i}$.
     - Therefore $\langle K_1, (V\,(W@^{-p \gg \mathsf{dom}_i} A))@^{p \gg \mathsf{cod}_i} B\rangle \not\longrightarrow^* \langle \lightning\, p \gg \mathsf{cod}_i\rangle$ for all K_1. We derive a contradiction with (2) that witnesses blame.

□

Lemma C.2.6. $(\forall K', N.\ K \longrightarrow^*_{\Box} K' \circ \Box\,N \Rightarrow N \in [\![A]\!]^+ \wedge K' \in [\![B]\!]^-) \Rightarrow K \in [\![A \to B]\!]^-$

Proof. We wish to show that $\langle \Phi, \Delta, K, V@^p A \to B\rangle \not\longrightarrow^* \langle \lightning\, -p\rangle$ for all V. Proceed with a proof by contradiction.

- Assume $\langle \Phi, \Delta, K, V@^p A \to B\rangle \longrightarrow^* \langle \lightning\, -p\rangle$ for some V, p.

- By assumption we know:

  $\langle \Phi, \Delta, K, V@^{p} A \to B \rangle \longrightarrow^{*} \langle \Phi', \Delta', K', (V\,(N@^{-p \gg \mathsf{dom}_i} A))@^{p \gg \mathsf{cod}_i} B \rangle \longrightarrow^{*} \langle \lightning\, -p \rangle$ for some K', N, i

- There are two cases

  1. $\langle K', (V\,(N@^{-p \gg \mathsf{dom}_i} A))@^{p \gg \mathsf{cod}_i} B \rangle \longrightarrow^{*} \langle \lightning\, -p \gg \mathsf{dom}_i \rangle$

     - Expanding reduction for (1) we have:
       $\langle K', (V\,(N@^{-p \gg \mathsf{dom}_i} A))@^{p \gg \mathsf{cod}_i} B \rangle \longrightarrow$
       $\langle K' \circ \Box\,@^{p \gg \mathsf{cod}_i} B, (V\,(N@^{-p \gg \mathsf{dom}_i} A)) \rangle \longrightarrow$
       $\langle K' \circ \Box\,@^{p \gg \mathsf{cod}_i} B \circ V\,\Box, N@^{-p \gg \mathsf{dom}_i} A \rangle \longrightarrow^{*} \langle \lightning\, -p \gg \mathsf{dom}_i \rangle$
     - By assumption we know that if for some V_1 $\langle K, V_1 \rangle \longrightarrow^{*} \langle K' \circ V_1\,\Box, N \rangle$ then $N \in [\![A]\!]^{+}$.
     - Pick $V_1 = V@^{p} A \to B$.
     - By assumption that $N \in [\![A]\!]^{+}$ then $\langle K_2, N@^{p_1} A \rangle \not\longrightarrow^{*} \langle \lightning\, p_1 \rangle$ for all K_2, p_1.
     - We derive a contradiction that $N \in [\![A]\!]^{+}$ picking $p_1 = -p \gg \mathsf{dom}_i$, $K_2 = K' \circ \Box\,@^{p \gg \mathsf{cod}_i} B \circ V\,\Box$.

  2. $\langle K', (V\,(N@^{-p \gg \mathsf{dom}_i} A))@^{p \gg \mathsf{cod}_i} B \rangle \longrightarrow^{*} \langle \lightning\, -p \gg \mathsf{cod}_i \rangle$

     - Expanding reduction for (2) we have:
       $\langle K', (V\,(N@^{-p \gg \mathsf{dom}_i} A))@^{p \gg \mathsf{cod}_i} B \rangle \longrightarrow$
       $\langle K' \circ \Box\,@^{p \gg \mathsf{cod}_i} B, V\,(N@^{-p \gg \mathsf{dom}_i} A) \rangle \longrightarrow^{*}$
       $\langle K' \circ \Box\,@^{p \gg \mathsf{cod}_i} B, W_1 \rangle \longrightarrow$
       $\langle K', W_1@^{p \gg \mathsf{cod}_i} B \rangle \longrightarrow^{*} \langle \lightning\, -p \gg \mathsf{cod}_i \rangle$
     - By assumption we know that if for some V_1 $\langle K, V_1 \rangle \longrightarrow^{*} \langle K' \circ V_1\,\Box, N \rangle$ then $K' \in [\![B]\!]^{-}$.
     - Pick $V_1 = V@^{p} A \to B$.
     - By assumption that $K' \in [\![B]\!]^{+}$ then $\langle K', W@^{p_1} B \rangle \not\longrightarrow^{*} \langle \lightning\, -p_1 \rangle$ for all W, p_1.
     - We derive a contradiction that $K' \in [\![B]\!]^{+}$ picking $p_1 = p \gg \mathsf{cod}_i$, $W = W_1$.

□

Lemma C.2.7. $V \in [\![A]\!]^{+} \wedge V \in [\![B]\!]^{+} \Rightarrow V \in [\![A \cap B]\!]^{+}$

Proof. By contrapositive.

- Assume $\langle K, V@^p A \cap B\rangle \longrightarrow^* \langle \lightning\, p\rangle$ for some K, p.
- There are two cases:
  1. $\langle K, V@^p A \cap B\rangle \longrightarrow^* \langle \lightning\, p \bullet \mathsf{left}^+_\cap[\mathsf{nil}]\rangle$
     - Expanding reduction for (1):
       $\langle K, V@^p A \cap B\rangle \longrightarrow$
       $\langle K, (V@^{p\bullet\mathsf{left}^+_\cap[\mathsf{nil}]} A)@^{p\bullet\mathsf{right}^+_\cap[\mathsf{nil}]} B\rangle \longrightarrow$
       $\langle K \circ \square\, @^{p\bullet\mathsf{right}^+_\cap[\mathsf{nil}]} B, V@^{p\bullet\mathsf{left}^+_\cap[\mathsf{nil}]} A\rangle \longrightarrow^* \langle \lightning\, p \bullet \mathsf{left}^+_\cap[\mathsf{nil}]\rangle$
     - Pick $K = K \circ \square\, @^{p\bullet\mathsf{right}^+_\cap[\mathsf{nil}]} B$, $p = p \bullet \mathsf{left}^+_\cap[\mathsf{nil}]$ to show that $V \notin [\![A]\!]^+$
  2. $\langle K, V@^p A \cap B\rangle \longrightarrow^* \langle \lightning\, p \bullet \mathsf{right}^+_\cap[\mathsf{nil}]\rangle$
     - By Lemma C.1.6 then $\langle K, (V@^{p\bullet\mathsf{right}^+_\cap[\mathsf{nil}]} B)@^{p\bullet\mathsf{left}^+_\cap[\mathsf{nil}]} A\rangle \longrightarrow^* \langle \lightning\, p\rangle$
     - We observe that (positive) contracts from right branch (B) are evaluated before the left, so we will still get blame on the right:
       $\langle K, (V@^{p\bullet\mathsf{right}^+_\cap[\mathsf{nil}]} B)@^{p\bullet\mathsf{left}^+_\cap[\mathsf{nil}]} A\rangle \longrightarrow^* \langle \lightning\, p \bullet \mathsf{right}^+_\cap[\mathsf{nil}]\rangle$
     - Perform one step of reduction:
       $\langle K, (V@^{p\bullet\mathsf{right}^+_\cap[\mathsf{nil}]} B)@^{p\bullet\mathsf{left}^+_\cap[\mathsf{nil}]} A\rangle \longrightarrow$
       $\langle K \circ \square\, @^{p\bullet\mathsf{left}^+_\cap[\mathsf{nil}]} A, V@^{p\bullet\mathsf{right}^+_\cap[\mathsf{nil}]} B\rangle \longrightarrow^* \langle \lightning\, p \bullet \mathsf{right}^+_\cap[\mathsf{nil}]\rangle$
     - Pick $K = K \circ \square\, @^{p\bullet\mathsf{left}^+_\cap[\mathsf{nil}]} A$, $p = p \bullet \mathsf{right}^+_\cap[\mathsf{nil}]$ to show that $V \notin [\![B]\!]^+$

□

Lemma C.2.8. $K \in [\![A]\!]^- \vee K \in [\![B]\!]^- \Rightarrow K \in [\![A \cap B]\!]^-$

Proof. By contrapositive.

- Assume $\langle K, V@^p A \cap B\rangle \longrightarrow^* \langle \lightning\, -p\rangle$ for some V, p.
- There is one case:
  - $\langle K, V@^p A \cap B\rangle \longrightarrow \langle K, (V@^{p\bullet\mathsf{left}^+_\cap[\mathsf{nil}]} A)@^{p\bullet\mathsf{right}^+_\cap[\mathsf{nil}]} B\rangle$
    $\longrightarrow^*$
    $\langle \lightning\, -p \bullet \mathsf{left}^-_\cap[c_i/P]\rangle$
    and

  - $\langle K, V@^{p} A \cap B\rangle \longrightarrow \langle K, (V@^{p \bullet \mathsf{left}^{+}_{\cap}[\mathsf{nil}]} A)@^{p \bullet \mathsf{right}^{+}_{\cap}[\mathsf{nil}]} B\rangle$
    $\longrightarrow^{*}$
    $\langle \lightning\, -p \bullet \mathsf{right}^{-}_{\cap}[c_i'/P']\rangle$.

- By Lemma C.1.6 then:
  - $\langle K, (V@^{p \bullet \mathsf{right}^{+}_{\cap}[\mathsf{nil}]} B)@^{p \bullet \mathsf{left}^{+}_{\cap}[\mathsf{nil}]} A\rangle \longrightarrow^{*} \langle \lightning\, -p \bullet \mathsf{left}^{-}_{\cap}[c_j/P]\rangle$ and
  - $\langle K, (V@^{p \bullet \mathsf{right}^{+}_{\cap}[\mathsf{nil}]} B)@^{p \bullet \mathsf{left}^{+}_{\cap}[\mathsf{nil}]} A\rangle \longrightarrow^{*} \langle \lightning\, -p \bullet \mathsf{right}^{-}_{\cap}[c_j'/P']\rangle$.

- To show $K \notin [\![A]\!]^{-}$:
  $\langle K, (V@^{p \bullet \mathsf{right}^{+}_{\cap}[\mathsf{nil}]} B)@^{p \bullet \mathsf{left}^{+}_{\cap}[\mathsf{nil}]} A\rangle \longrightarrow$
  $\langle K \circ \square\, @^{p \bullet \mathsf{left}^{+}_{\cap}[\mathsf{nil}]} A, V@^{p \bullet \mathsf{right}^{+}_{\cap}[\mathsf{nil}]} B\rangle \longrightarrow^{*}$
  $\langle K \circ \square\, @^{p \bullet \mathsf{left}^{+}_{\cap}[\mathsf{nil}]} A, W\rangle \longrightarrow$
  $\langle K, W@^{p \bullet \mathsf{left}^{+}_{\cap}[\mathsf{nil}]} A\rangle \longrightarrow^{*} \langle \lightning\, -p \bullet \mathsf{left}^{-}_{\cap}[c_j'/P']\rangle$.
  Pick $V = W, p = p \bullet \mathsf{left}^{+}_{\cap}[\mathsf{nil}]$

- To show $K \notin [\![B]\!]^{-}$:
  $\langle K, V@^{p} A \cap B\rangle \longrightarrow$
  $\langle K, (V@^{p \bullet \mathsf{left}^{+}_{\cap}[\mathsf{nil}]} A)@^{p \bullet \mathsf{right}^{+}_{\cap}[\mathsf{nil}]} B\rangle \longrightarrow$
  $\langle K \circ \square\, @^{p \bullet \mathsf{right}^{+}_{\cap}[\mathsf{nil}]} B, V@^{p \bullet \mathsf{left}^{+}_{\cap}[\mathsf{nil}]} A\rangle \longrightarrow^{*}$
  $\langle K \circ \square\, @^{p \bullet \mathsf{right}^{+}_{\cap}[\mathsf{nil}]} B, W\rangle \longrightarrow$
  $\langle K, W@^{p \bullet \mathsf{right}^{+}_{\cap}[\mathsf{nil}]} B\rangle \longrightarrow^{*} \langle \lightning\, -p \bullet \mathsf{right}^{-}_{\cap}[c_j'/P']\rangle$.
  Pick $V = W, p = p \bullet \mathsf{right}^{+}_{\cap}[\mathsf{nil}]$

□

Lemma C.2.9. $V \in [\![A]\!]^{+} \vee V \in [\![B]\!]^{+} \Rightarrow V \in [\![A \cup B]\!]^{+}$

Proof. Essentially the same as negative intersection. □

Lemma C.2.10. $K \in [\![A]\!]^{-} \wedge K \in [\![B]\!]^{-} \Rightarrow K \in [\![A \cup B]\!]^{-}$

Proof. Essentially the same as positive intersection. □

Theorem 4.4.1 (Monitoring Properties)**.** $\lambda^{\cap\cup}$ *satisfies the properties in Figure 4.3.*

Proof. Using each lemma from Section C. □

Bibliography

Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. 2012. A Core Calculus for Provenance. In *Principles of Security and Trust*, Pierpaolo Degano and Joshua D. Guttman (Eds.). Springer Berlin Heidelberg, 410–429.

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

Thomas H. Austin, Tim Disney, and Cormac Flanagan. 2011. Virtual Values for Language Extension. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 921–938. https://doi.org/10.1145/2048066.2048136

F. Barbanera, M. Dezaniciancaglini, and U. Deliguoro. 1995. Intersection and Union Types. *Inf. Comput.* 119, 2 (June 1995), 202–230. https://doi.org/10.1006/inco.1995.1086

Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy G. Siek, and Sam Tobin-Hochstadt. 2017. Sound Gradual Typing: Only Mostly Dead. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 54 (Oct. 2017), 24 pages. https://doi.org/10.1145/3133878

Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 257–281.

Matthias Blume and David Mcallester. 2006. Sound and complete models of contracts. *Journal of Functional Programming* 16, 4-5 (2006), 375–414. https://doi.org/10.1017/S0956796806005971

Rajendra Bose and James Frew. 2005. Lineage Retrieval for Scientific Data Processing: A Survey. *ACM Comput. Surv.* 37, 1 (March 2005), 1–28. https://doi.org/10.1145/1057977.1057978

Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *Proceedings of the 8th International Conference on Database Theory (ICDT '01)*. Springer-Verlag, Berlin, Heidelberg, 316–330. http://dl.acm.org/citation.cfm?id=645504.656274

Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 41 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110285

Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual Typing: A New Perspective. *Proc. ACM Program. Lang.* 3, POPL, Article 16 (Jan. 2019), 32 pages. https://doi.org/10.1145/3290329

Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and Precise Type Checking for JavaScript. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 48 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133872

James Cheney. 2011. A Formal Framework for Provenance Security. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium (CSF '11)*. IEEE Computer Society, Washington, DC, USA, 281–293. https://doi.org/10.1109/CSF.2011.26

James Cheney, Amal Ahmed, and Umut A. Acar. 2007. Provenance as Dependency Analysis. In *Database Programming Languages*, Marcelo Arenas and Michael I. Schwartzbach (Eds.). Springer Berlin Heidelberg, 138–152.

James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009a. Provenance in Databases: Why, How, and Where. *Found. Trends databases* 1, 4 (April 2009), 379–474. https://doi.org/10.1561/1900000006

James Cheney, Stephen Chong, Nate Foster, Margo Seltzer, and Stijn Vansummeren. 2009b. Provenance: A Future History. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 957–964. https://doi.org/10.1145/1639950.1640064

Benjamin Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. 2018. KafKa: Gradual Typing for Objects. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 12:1–12:24. https://doi.org/10.4230/LIPIcs.ECOOP.2018.12

M. Coppo and M. Dezani-Ciancaglini. 1978. A new type assignment for λ-terms. *Archiv für mathematische Logik und Grundlagenforschung* 19, 1 (01 Dec 1978), 139–156. https://doi.org/10.1007/BF02011875

M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. 1981. Functional Characters of Solvable Terms. *Mathematical Logic Quarterly* 27, 26 (1981), 45–58. https://doi.org/10.1002/malq.19810270205

H. B. Curry. 1934. Functionality in Combinatory Logic. *Proceedings of the National Academy of Sciences* 20, 11 (1934), 584–590. https://doi.org/10.1073/pnas.20.11.584 arXiv:http://www.pnas.org/content/20/11/584.full.pdf

Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, New York, NY, USA, 198–208. https://doi.org/10.1145/351240.351259

Christos Dimoulas and Matthias Felleisen. 2011. On Contract Satisfaction in a Higher-order World. *ACM Trans. Program. Lang. Syst.* 33, 5, Article 16 (Nov. 2011), 29 pages. https://doi.org/10.1145/2039346.2039348

Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. 2011. Correct Blame for Contracts: No More Scapegoating. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 215–226. https://doi.org/10.1145/1926385.1926410

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete Monitors for Behavioral Contracts. In *Proceedings of the 21st European Conference on Programming Languages and Systems (ESOP'12)*. Springer-Verlag, Berlin, Heidelberg, 214–233. https://doi.org/10.1007/978-3-642-28869-2_11

Tim Disney. 2015. *Hygienic Macros For Javascript*. Ph.D. Dissertation. University of California Santa Cruz.

Tim Disney, Cormac Flanagan, and Jay McCarthy. 2011. Temporal Higher-order Contracts. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 176–188. https://doi.org/10.1145/2034773.2034800

Joshua Dunfield and Frank Pfenning. 2003. Type Assignment for Intersections and Unions in Call-by-Value Languages. In *Foundations of Software Science and Computation Structures*, Andrew D. Gordon (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 250–266.

Asger Feldthaus and Anders Møller. 2014. Checking Correctness of TypeScript Interfaces for JavaScript Libraries. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 1–16. https://doi.org/10.1145/2660193.2660215

Matthias Felleisen and Daniel P. Friedman. 1986. Control Operators, the SECD-Machine, and the λ-Calculus. In *Formal description of programming concepts-III*. North-Holland, 193–217.

Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. Collapsible Contracts: Fixing a Pathology of Gradual Typing. 2, OOPSLA (2018), 133:1–133:27. https://doi.org/10.1145/3276503

Luminous Fennell and Peter Thiemann. 2013. The Blame Theorem for a Linear Lambda Calculus with Type Dynamic. In *Proceedings of the 2012 Conference on Trends in Functional Programming - Volume 7829 (TFP 2012)*. Springer-Verlag New York, Inc., New York, NY, USA, 37–52. https://doi.org/10.1007/978-3-642-40447-4_3

Robert Bruce Findler and Matthias Blume. 2006. Contracts as Pairs of Projections. In *Functional and Logic Programming*, Masami Hagiya and Philip Wadler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 226–241.

Robert Bruce Findler, Matthias Blume, and Matthias Felleisen. 2004. *An investigation of contracts as projections.* Technical Report.

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. https://doi.org/10.1145/581478.581484

Matthew Flatt and PLT. 2010. *Reference: Racket.* Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

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 '16)*. ACM, New York, NY, USA, 429–442. https://doi.org/10.1145/2837614.2837670

Ben Greenman and Matthias Felleisen. 2018. A Spectrum of Type Soundness and Performance. *Proc. ACM Program. Lang.* 2, ICFP, Article 71 (July 2018), 32 pages. https://doi.org/10.1145/3236766

Ben Greenman and Zeina Migeed. 2018. On the Cost of Type-tag Soundness. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '18)*. ACM, New York, NY, USA, 30–39. https://doi.org/10.1145/3162066

Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. 2007. Relationally-parametric Polymorphic Contracts. In *Proceedings of the 2007 Symposium on Dynamic Languages (DLS '07)*. ACM, New York, NY, USA, 29–40. https://doi.org/10.1145/1297081.1297089

Phillip Heidegger and Peter Thiemann. 2010. (Contract-driven Testing of Javascript Code). In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns (TOOLS'10)*. Springer-Verlag, Berlin, Heidelberg, 154–172. http://dl.acm.org/citation.cfm?id=1894386.1894395

Fritz Henglein. 1994. Dynamic Typing: Syntax and Proof Theory. *Sci. Comput. Program.* 22, 3 (June 1994), 197–230. https://doi.org/10.1016/0167-6423(94)00004-2

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient Gradual Typing. *Higher-Order and Symbolic Computation* 23, 2 (June 2010), 167–189. https://doi.org/10.1007/s10990-011-9066-z

Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. 2017b. Gradual Session Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 38 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110282

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

Limin Jia, Hannah Gommerstadt, and Frank Pfenning. 2016. Monitors and Blame Assignment for Higher-order Session Types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 582–594. https://doi.org/10.1145/2837614.2837662

Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. 2015. Transparent Object Proxies in JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 149–173. https://doi.org/10.4230/LIPIcs.ECOOP.2015.149

Matthias Keil and Peter Thiemann. 2013. On the Proxy Identity Crisis. *CoRR* abs/1312.5429 (2013). arXiv:1312.5429 http://arxiv.org/abs/1312.5429

Matthias Keil and Peter Thiemann. 2015a. 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. https://doi.org/10.1145/2784731.2784737

Matthias Keil and Peter Thiemann. 2015b. TreatJS: Higher-Order Contracts for JavaScripts. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 28–51. https://doi.org/10.4230/LIPIcs.ECOOP.2015.28

Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. 2016. Occurrence Typing Modulo Theories. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 296–309. https://doi.org/10.1145/2908080.2908091

Erik Krogh Kristensen and Anders Møller. 2017a. Inference and Evolution of TypeScript Declaration Files. In *Fundamental Approaches to Software Engineering*, Marieke Huisman and Julia Rubin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 99–115.

Erik Krogh Kristensen and Anders Møller. 2017b. Type Test Scripts for TypeScript Testing. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 90 (Oct. 2017), 25 pages. https://doi.org/10.1145/3133914

Jacob Matthews and Amal Ahmed. 2008. Parametric Polymorphism Through Run-time Sealing or, Theorems for Low, Low Prices!. In *Proceedings of the Theory and Practice of Software, 17th European Conference on Programming Languages and Systems (ESOP'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 16–31. http://dl.acm.org/citation.cfm?id=1792878.1792881

Hernán Melgratti and Luca Padovani. 2017. Chaperone Contracts for Higher-order Sessions. *Proc. ACM Program. Lang.* 1, ICFP, Article 35 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110279

Bertrand Meyer. 1988. *Object-Oriented Software Construction* (1st ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Bertrand Meyer. 1992. *Eiffel: The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type Regression Testing to Detect Breaking Changes in Node.js Libraries. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 7:1–7:24. https://doi.org/10.4230/LIPIcs.ECOOP.2018.7

Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. Johns Hopkins University, Baltimore, Maryland, USA.

Fabian Muehlboeck and Ross Tate. 2017. Sound Gradual Typing is Nominally Alive and Well. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 56 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133880

Fabian Muehlboeck and Ross Tate. 2018. Empowering Union and Intersection Types with Integrated Subtyping. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*. ACM, New York, NY, USA, Article 112, 29 pages. https://doi.org/10.1145/3276482

Benjamin Crawford Pierce. 1992. *Programming with Intersection Types and Bounded Polymorphism*. Ph.D. Dissertation. Pittsburgh, PA, USA. UMI Order No. GAX92-16028.

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 167–180. https://doi.org/10.1145/2676726.2676971

John Reynolds. 1983. Types, abstraction, and parametric polymorphism. In *Information Processing*.

Gregor Richards, Ellen Arteca, and Alexi Turcotte. 2017. The VM Already Knew That: Leveraging Compile-time Knowledge to Optimize Gradual Typing. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 55 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133879

Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete Types for TypeScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 76–100. https://doi.org/10.4230/LIPIcs.ECOOP.2015.76

Tiark Rompf and Nada Amin. 2016. Type Soundness for Dependent Object Types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 624–641. https://doi.org/10.1145/2983990.2984008

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

Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP 2007 – Object-Oriented Programming*, Erik Ernst (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–27.

Jeremy G. Siek, Peter Thiemann, and Philip Wadler. 2015a. Blame and Coercion: Together Again for the First Time. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 425–435. https://doi.org/10.1145/2737924.2737968

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015b. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015c. Monotonic References for Efficient Gradual Typing. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 432–456.

Jeremy G. Siek and Philip Wadler. 2010. Threesomes, with and Without Blame. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 365–376. https://doi.org/10.1145/1706299.1706342

Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. 2005. A Survey of Data Provenance in e-Science. *SIGMOD Rec.* 34, 3 (Sept. 2005), 31–36. https://doi.org/10.1145/1084805.1084812

T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. Chaperones and Impersonators: Run-time Support for Reasonable Interposition. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 943–962. https://doi.org/10.1145/2384616.2384685

Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. 2014. Gradual Typing Embedded Securely in JavaScript. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 425–437. https://doi.org/10.1145/2535838.2535889

Cameron Swords, Amr Sabry, and Sam Tobin-Hochstadt. 2018. An Extended Account of Contract Monitoring Strategies as Patterns of Communication. *Journal of Functional Programming* 28 (2018), e4. https://doi.org/10.1017/S0956796818000047

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 456–468. https://doi.org/10.1145/2837614.2837630

Thiemann, Peter. 2014. Session types with Gradual Typing. In *International Symposium on Trustworthy Global Computing*. Springer, 144–158.

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs.. In *Dynamic Languages Symposium (DLS) (DLS '06)*.

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 395–406. https://doi.org/10.1145/1328438.1328486

Matías Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual Parametricity, Revisited. *Proc. ACM Program. Lang.* 3, POPL, Article 17 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290330

Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. 2018. The Behavior of Gradual Types: A User Study. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages (DLS 2018)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3276945.3276947

Tom Van Cutsem and Mark S. Miller. 2010. Proxies: Design Principles for Robust Object-oriented Intercession APIs. In *Proceedings of the 6th Symposium on Dynamic*

Languages (DLS '10). ACM, New York, NY, USA, 59–72. https://doi.org/10.1145/1869631.1869638

Tom Van Cutsem and Mark S. Miller. 2013. Trustworthy Proxies. In *European Conference on Object-Oriented Programming (ECOOP)*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 154–178.

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS '14)*. ACM, New York, NY, USA, 45–56. https://doi.org/10.1145/2661088.2661101

Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big Types in Little Runtime: Open-world Soundness and Collaborative Blame for Gradual Type Systems. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 762–774. https://doi.org/10.1145/3009837.3009849

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*. ACM, New York, NY, USA, 347–359. https://doi.org/10.1145/99370.99404

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009 (ESOP '09)*. Springer-Verlag, Berlin, Heidelberg, 1–16. https://doi.org/10.1007/978-3-642-00590-9_1

Jack Williams, J. Garrett Morris, and Philip Wadler. 2018. The Root Cause of Blame: Contracts for Intersection and Union Types. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 134 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276504

Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. 2017a. Mixed Messages: Measuring Conformance and Non-Interference in TypeScript. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 74. 28:1–28:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.28

Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. 2017b. Mixed Messages: Measuring Conformance and Non-Interference in TypeScript (Artifact). *Dagstuhl Artifacts Series* 3, 2 (2017), 8:1–8:2. https://doi.org/10.4230/DARTS.3.2.8