The Root Cause of Blame: Contracts for Intersection and Union Types

JACK WILLIAMS, University of Edinburgh
J. GARRETT MORRIS, University of Kansas
PHILIP WADLER, University of Edinburgh

Gradual typing has emerged as the tonic for programmers wanting a mixture of static and dynamic typing, hoping to achieve the best of both. Contracts provide a lightweight form of gradual typing as they can be implemented as a library, rather than requiring a gradual type system.

Intersection and union types are well suited to static and dynamic languages: intersection encodes overloaded functions; union encodes uncertain data arising from branching code. We extend the untyped lambda calculus with contracts for monitoring higher-order intersection and union types, giving a uniform treatment to both. Each operator requires a single reduction rule that does not depend on the constituent types or the context of the operator, unlike existing work.

We present a new method for defining contract satisfaction based on blame behaviour. A value positively satisfies a type if applying a contract of that type can never elicit positive blame. A continuation negatively satisfies a type if applying a contract of that type can never elicit negative blame. We supplement our definition of satisfaction with a series of monitoring properties that satisfying values and continuations should have. These properties ensure that the semantics of contracts are in alignment with the static types they represent.

Additional Key Words and Phrases: Gradual typing, Blame, Contracts, Intersection, Union

1 INTRODUCTION

Gradual typing has emerged as the tonic for programmers wanting a mixture of static and dynamic typing, hoping to achieve the best of both. Sound gradual typing [Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006] is the most potent brew, providing static type-checking, auto-completion, and dynamic assertions.

Contracts [Meyer 1988] provide a lightweight form of gradual typing that enforce types dynamically using assertions. Contracts can be implemented in both typed and untyped languages as a library, without demanding a gradual type system or compilation phase. They provide a gateway for programmers into sound gradual typing.

Gradual typing and contracts are complementary, not competing. The work by Findler and Felleisen [2002] on higher-order function contracts proved to be the catalyst for gradual typing. Their key insight was to assign *blame* when an assertion failed and distinguish when the fault was due to code inside the contract, or due to the context containing the contract. The former is know as *positive* blame and the latter as *negative* blame. The study of correct blame assignment continues to be an important area of research concerning all aspects of gradual typing and contracts.

Composition is the primary method for building rich libraries of assertions. Findler and Felleisen [2002] gave us higher-order function contracts; Keil and Thiemann [2015a] gave us higher-order intersection and union contracts. These operators are well suited to static and dynamic languages: intersection encodes overloaded functions; union encodes uncertain data arising from branching code. For example, we can use all three to build a contract for the following JavaScript program.

Authors' addresses: Jack Williams, University of Edinburgh, jack.williams@ed.ac.uk; J. Garrett Morris, University of Kansas, garrett@ittc.ku.edu; Philip Wadler, University of Edinburgh, wadler@inf.ed.ac.uk.

2018. 2475-1421/2018/8-ART1 $15.00
https://doi.org/

```
function f(x) {
  if (typeof x === 'boolean') {
    if (x) return 'hello world'; else return !x;
  }
  return (x+10);
}
```

When the argument to `f` is a boolean then return the string `'hello world'` for true, and the negated input for false. When the argument to `f` is an integer then return the argument plus 10. The behaviour of this function can be captured using the type $(\mathtt{B} \to (\mathtt{S} \cup \mathtt{B})) \cap (\mathtt{I} \to \mathtt{I})$.

However, the monitoring semantics for contracts of intersection and union types given by Keil and Thiemann [2015a] are not uniform. Intersection requires three monitoring rules and specialises intersections of function types. Union requires a single rule that extracts the union from within an intersection and creates two resulting intersection contracts. If uniformity helps composition, then special cases can hinder composition.

We extend the untyped lambda calculus with contracts for monitoring higher-order intersection and union types, giving a uniform treatment to both. Each operator requires a single monitoring rule that does not depend on the constituent types or the context of the operator. Our calculus has the triumvirate of monitoring rules for function, intersection, and union:

$$\begin{array}{lcl} (V @^{p} A \to B)\, W & \longrightarrow & (V\,(W @^{-p} A)) @^{p} B \\ V @^{p} A \cap B & \longrightarrow & V @^{p \bullet \mathsf{left}_{\cap}} A @^{p \bullet \mathsf{right}_{\cap}} B \\ V @^{p} A \cup B & \longrightarrow & V @^{p \bullet \mathsf{left}_{\cup}} A @^{p \bullet \mathsf{right}_{\cup}} B \end{array}$$

We write $V @^{p} A$ to denote the application of a contract of type A to value V. Contracts are indexed by blame nodes p that track and interpret blame for the different contract operators. To support uniform monitoring of intersection and union we use richer structures for blame tracking that keep trace of the reduction.

We present a new approach for determining blame in the presence of intersection and union. Allocating blame becomes non-trivial with intersection and union because sub-contracts can be violated without implying that a top-level contract was violated. In our example, a client of `f` can violate the domain type in the left branch of the intersection provided they satisfy the domain type in the right branch. Blame allocation is required to aggregate violations throughout the execution of a program.

Verifying the soundness of contract monitoring requires a definition of contract satisfaction. Programs that satisfy a contract should never elicit blame and applying a contract to any program should produce a new program that satisfies the contract. We follow existing work by Dimoulas and Felleisen [2011] and Keil and Thiemann [2015a] that distinguishes positive and negative satisfaction. Values positively satisfy a contract and continuations negatively satisfy a contract.

Dimoulas and Felleisen [2011] define contract satisfaction using observational equivalence between programs contracted with full and partial obligations. A value satisfies a type if the positive obligations of a type are not observable. We refer to this as a *monitoring oriented* approach because satisfaction is defined using contract behaviour.

Keil and Thiemann [2015a] define contract satisfaction using a set of coinductive rules to describe satisfying programs and contexts. A value satisfies a function type if when applied to any argument that satisfies the domain type, the value returns a result that satisfies the codomain type. We refer to this as a *denotational* approach because satisfaction is defined using program structure and does not mention contracts or blame.

We present a new monitoring oriented approach to defining contract satisfaction using *blame*. A value positively satisfies a type if applying a contract of that type can never elicit positive blame. A

continuation negatively satisfies a type if applying a contract of that type can never elicit negative blame. Using blame to define satisfaction confers multiple advantages. Existing techniques that use observational equivalence do not readily extend to intersection and union because they assume that an observable contract denotes a failure to satisfy. This is not true in the presence of intersection where a context may legitimately violate a contract to select a function overload. Existing techniques that use a denotational approach to satisfaction state soundness properties that rely on contract violations terminating a program, and consequently, only apply to top-level contracts. This can hinder compositional reasoning in the presence of intersection and union where violations may not terminate a program. Defining satisfaction using blame allows us to state additional soundness properties that facilitate compositional reasoning.

The definition of contract satisfaction we present says nothing about how satisfying terms and contexts behave. Our system could be sound but useless in the context of gradual typing because the contract semantics do not match the static type semantics. We supplement our definition of contract satisfaction with a series of *sound monitoring properties* that satisfying values and continuations should have. Each type has a positive property for values and a negative property for continuations, ensuring that the semantics of contracts accurately reflect the types they represent.

The contributions of this paper are:

- Extending the untyped lambda calculus with contracts for monitoring higher-order intersection and union types in a uniform way. Previous work has multiple rules for intersection that depend on constituent types, and a rule for union that must extract the type from within an intersection. This means that monitoring nested intersection and union may duplicate contracts. Our calculus has a single rule for each, avoiding contract duplication.
- Defining blame allocation for intersection and union that supports uniform monitoring rules, stratifying blame allocation into two phases: blame *assignment* and blame *resolution*. We believe the distinction is helpful when designing correct blame allocation for practical settings, where terminating the program at the first contract violation may not be desirable.
- Providing a new definition of contract satisfaction in terms of blame behaviour. Our work is the first to define contract satisfaction using monitoring behaviour in the presence of intersection and union. Using our definition of contract satisfaction we are able to state soundness properties that do not depend on termination from contract violations, when existing systems cannot.
- Presenting a series of sound monitoring properties that any contract system with intersection and union types should satisfy, ensuring that satisfying programs behave as expected. Our calculus has the expected properties.

The paper is structured as follows: Section 2 reviews higher-order blame, including intersection and union; Section 3 presents the untyped lambda calculus extended with contracts for intersection and union types; Section 4 adds blame assignment to the calculus; Section 5 gives our technical results about contract semantics; Section 6 compares existing approaches to contract semantics; Section 7 discusses related work; and Section 8 concludes.

2 HIGHER-ORDER BLAME

In this section we give an overview of higher-order blame for function, intersection, and union types. The theory of blame for function types was first presented by Findler and Felleisen [2002]; the theory of blame for intersection and union was first presented by Keil and Thiemann [2015a]. We characterise blame in terms of two concepts: blame *assignment* and blame *resolution*.

In the examples we use M to range over program terms, V and W to range over values, A and B to range over contract types including operators $\rightarrow, \cap, \cup$ and base contracts B (boolean) and

I (integer), and p to range over blame *nodes*. Existing literature only includes blame labels (or identifiers), however, in the presence of intersection and union richer structures are required. We describe the full blame tracking structure in Section 3. For now, we consider p to be abstract.

This work considers contracts in the context of gradual typing and we only define flat contracts for fixed types such as integers and booleans, whereas previous work allows user-defined contracts [Findler and Felleisen 2002; Keil and Thiemann 2015a]. We do not consider this detrimental to the significance of our contributions as they are orthogonal to the choice of flat (or base) contracts.

2.1 Function Types

Immediately checking program conformance to a function type is not possible using a contract. A particular context may supply an illegal argument, or the function may only return illegal results for certain arguments. As a result, function contracts fix to the program they monitor and every application of the function is *wrapped*.

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

The wrap rule (above) tracks blame for domain and codomain contracts differently. The domain contract negates, or complements, blame node p. Negation of a blame node p is denoted by $-p$ and the operation is involutive. By negating the blame node we indicate that should the domain contract fail, raising blame on $-p$, the fault is with the *context* of the contract originally annotated with node p. This makes it possible to correctly attribute blame for higher-order cases.

$$\begin{aligned}
& ((\lambda f.f\ 1) @^{p} (\mathsf{B} \to \mathsf{B}) \to \mathsf{B})\, (\lambda y.y) \\
\xrightarrow{wrap}\ & ((\lambda f.f\ 1)\, ((\lambda y.y) @^{-p} \mathsf{B} \to \mathsf{B})) @^{p} \mathsf{B} \\
\xrightarrow{beta}\ & (((\lambda y.y) @^{-p} \mathsf{B} \to \mathsf{B})\ 1) @^{p} \mathsf{B} \\
\xrightarrow{wrap}\ & (((\lambda y.y)\, (1 @^{p} \mathsf{B})) @^{-p} \mathsf{B}) @^{p} \mathsf{B}
\end{aligned}$$

In the above program blame will not occur until the inner application of the function bound to f. At this point two invocations of the wrap rule will have taken place. First, using type $(\mathsf{B} \to \mathsf{B}) \to \mathsf{B}$ annotated with blame node p; second, using type $\mathsf{B} \to \mathsf{B}$ annotated with blame node $-p$. The argument to f gets wrapped with a boolean contract annotated with node p (after double negation), raising blame on p. This indicates that the fault comes from the code inside the contract, the *subject*, specifically the application of a function of type $\mathsf{B} \to \mathsf{B}$ to argument 1.

2.2 Dissecting Blame

When a contract $M @^{p} A$ fails we say blame is *raised* on p. For a system with simple types the process of raising blame is immediate. However, when adding operators such as intersection and union the process becomes more involved. We dissect blame into two components to make the process clearer once extended to intersection and union. We demonstrate that there is also merit to describing blame in this way, even for simple types.

Raising blame can be viewed in two stages. First is blame *assignment*. When a contract annotated with p fails we must determine whether p should be assigned blame (or responsibility) for the violation. Second is blame *resolution*. When a blame node p is assigned blame we must determine what to do with that blame. In existing systems with simple types this process is immediate. Blame is *always* assigned in a violation and resolution *always* terminates the program with a blame error.

This approach is sound but only under conditions where resolution always terminates the program immediately. Under a different blame resolution strategy the approach of always assigning blame in a violation may be incorrect. Consider a practical setting of contracts where blame

resolution does not terminate the program, instead, failure is logged and execution continues. Such a system might be suitable when a programmer has an existing program that is "correct" according to tests but they wish to scrutinise the program at certain types. In this case failing at the first violation would be inconvenient. Take the following program:

$$((\lambda x.x)@^{p}\mathsf{B} \to \mathsf{B})\,1$$

When blame resolution only logs this program would first raise and assign blame to $-p$ as the argument 1 is not a boolean. Then the program would raise and assign blame to p as the result is not a boolean. Assigning blame to p wrongly indicates that the function fails to satisfy type $\mathsf{B} \to \mathsf{B}$.

The solution is to make blame assignment take into account previous violations from the context. When determining whether to assign blame to a node p, first consider whether the context of the contract, represented by $-p$, has already been assigned blame. If the context has already violated the contract then the subject is no longer obligated to follow the contract and should not be assigned blame. Under this approach the example raises blame on p for the failing codomain contract but does not *assign* blame to p. The earlier assignment of blame to $-p$ means the subject has been relieved of duty by an ill-behaved context. When using this technique care has to be taken to distinguish multiple applications of the same function. To this end, we assign a fresh index to each contract application. We detail this in Section 3.

The distinction of assignment and resolution has no technical impact on previous work with simple types where blame always terminates, although it is a useful dichotomy when considering implementation. This is not the case for intersection and union where assignment and resolution must be carefully defined to guarantee correct blame semantics.

2.3 Intersection Types

Intersection types have been well studied in theory [Coppo and Dezani-Ciancaglini 1978; Davies and Pfenning 2000; Pierce 1993] and are now starting to appear in mainstream languages such as TypeScript [Bierman et al. 2014] and Flow for JavaScript [Chaudhuri et al. 2017]. Applications for intersection types include mixins, describing multiple inheritance, and overloaded functions.

Monitoring. The process of checking (or monitoring) a program against an intersection contract should involve checking the program against both components of the intersection. As was the case with functions, we expect there to be a single rule that decomposes an intersection. Previous work by Keil and Thiemann [2015a] requires three rules: a rule that applies intersections of function types and two rules for extracting base types from within an intersection. In Section 3 we give an operational semantics with a single monitoring rule for intersection. Here, we present a simplified version for the purpose of explaining the examples.

$$V@^{p}A \cap B \longrightarrow V@^{p\bullet\mathsf{left}_\cap}A\,@^{p\bullet\mathsf{right}_\cap}B$$

Monitoring an intersection contract decomposes the contract into its constituent types with each new contract annotated with an augmented blame node. A blame node $p \bullet \mathsf{left}_\cap$ denotes the left branch of an intersection type with parent p and $p \bullet \mathsf{right}_\cap$ denotes the right. The program then proceeds according to the monitoring behaviour of types A and B.

Blame. To understand blame allocation for intersection types it is helpful to start with their static type rules.

$$\text{I-}\cap\ \frac{M : A \qquad M : B}{M : A \cap B} \qquad\qquad \text{E-}\cap_i\ \frac{M : A_1 \cap A_2 \qquad i \in \{1,2\}}{M : A_i}$$

The introduction rule specifies that for a program, or subject, to satisfy an intersection type $A \cap B$ the subject must individually satisfy A and B. In languages with computational effects M must be restricted to a value for soundness under call-by-value [Davies and Pfenning 2000]. Introduction rules correspond to the positive blame behaviour of contract types while elimination rules correspond to the negative blame behaviour of contract types. The elimination rules specify that if a program satisfies type $A \cap B$ then a context may choose to use the program at A or B, eliminating the other type. In languages with subtyping for intersections the elimination rule is admissible. These rules give us an initial candidate for blame resolution when blame is assigned to components of an intersection.

+ Positive blame is assigned to an intersection type $A \cap B$ when positive blame is assigned to A *or* B. Evidently the program did not satisfy them both.
- Negative blame is assigned to an intersection type $A \cap B$ when negative blame is assigned to A *and* B. A context can choose to eliminate one type, denoted by negative blame on that type, but a context cannot choose to eliminate both.

This interpretation introduces a concept not present for simple types: a contract may fail without necessarily causing an error. Consider the following example where an intersection of function types is used to model an overloaded function. The function should return an integer when given an integer and return a boolean when given a boolean.

$$(\lambda x.x)@^{p}(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B})\ \mathtt{true} \longrightarrow (\lambda x.x)@^{p \bullet \mathsf{left}_{\cap}}(\mathtt{I} \to \mathtt{I})\ @^{p \bullet \mathsf{right}_{\cap}}(\mathtt{B} \to \mathtt{B})\ \mathtt{true}$$

The intersection contract is decomposed and evaluation continues with the two function contracts, applying the *wrap* rule to both. This program will assign negative blame on the left branch because the domain of type $\mathtt{I} \to \mathtt{I}$ was violated, but will not assign blame on the right branch. According to our definition of blame resolution this does not violate the intersection type. The function then returns `true`, triggering a violation on the codomain of type $\mathtt{I} \to \mathtt{I}$. Should this assign positive blame to the intersection type? To do so would be unfair as the function is only returning the value it was given. Blame *assignment* must take into account negative blame assigned against type $\mathtt{I} \to \mathtt{I}$, invalidating any positive blame raised against that type. When the context chooses to eliminate the type $\mathtt{I} \to \mathtt{I}$ from the intersection the subject is no longer obligated to satisfy that type.

Our proposed interpretation of negative blame for intersections is close, but not quite correct, as Keil and Thiemann [2015a] observed. The next example demonstrates the problem using an overloaded function applied twice.

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

The first application assigns negative blame against type $\mathtt{B} \to \mathtt{B}$ while the second application assigns negative blame against type $\mathtt{I} \to \mathtt{I}$. If we follow the interpretation that negative blame is assigned to an intersection if both constituents are assigned negative blame this program would blame $-p$, the context of the intersection contract. However, this use of an overloaded function is legitimate. The missing detail in our blame resolution is that we must not only track *if* both branch types have been negatively blamed, but also *where* they were blamed.

We refer back to the static elimination rule for guidance. The elimination rule for intersection types does not place any constraint on where the elimination takes places; a context is free to eliminate a branch at any point and may eliminate multiple occurrences of the type differently. For instance, the elimination of the intersection type could have been applied in the following context, although this would not be sound.

$$\mathsf{let}\ f = \square\ \mathsf{in}\ f\ 1; f\ \mathtt{true}$$

Aggregating negative blame on an intersection contract across *all* uses assumes that the context made the elimination choice at the source of the contract. Such an assumption is not always correct as demonstrated by the previous example where the wrapped function is applied twice. Alternatively, the context can (and did) defer the elimination choice until each function application, making a different choice at each. The example below shows the contexts where the eliminations were made. The first context would apply rule E-$\cap_1$ and the second context would apply rule E-$\cap_2$.

$$\begin{array}{l}\mathsf{let}\ f = V@^p(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B})\ \mathsf{in}\ \square_1\, 1; f\ \mathtt{true}\\ \mathsf{let}\ f = V@^p(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B})\ \mathsf{in}\ f\, 1; \square_2\ \mathtt{true}\end{array}$$

Monitoring programs is inherently reactive therefore a contract must conservatively assume that the elimination of the intersection type is made as late as possible by the context. Guha et al. [2007] make a similar observation when implementing polymorphic contracts, though the choice is not eliminating an intersection but instantiating a generic type parameter.

Negative blame should not be aggregated across *all* uses of an intersection, but *per use* of that intersection. We restate negative blame resolution for intersection types.

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

This modified interpretation of blame resolution would correctly permit the previous example as negative blame is assigned to both branches but in *different* contexts: no single elimination context assigns blame to both branches. The next example illustrates when negative blame should be assigned. This program demonstrates that blame can still arise from multiple applications but it is the initial elimination context that matters.

$$V@^p(\mathtt{I} \to \mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B})\, 1\, \mathtt{true}$$

The first application assigns negative blame to the right branch, while the application of the returned value assigns negative blame to the left branch. Blame in each branch originated in different applications however both violations trace back to the same context that eliminated the intersection type. Blame tracking for intersection types must ensure that the monitoring state is distinct for each elimination, and also tie the provenance of that elimination to all sub-contracts.

2.4 Union Types

Union types [Barbanera and Dezani-Ciancaglini 1991] describe values known to satisfy one of a range of types making them suitable for encoding uncertainty arising in programming. For example, a conditional expression that yields values of type A in one branch and values of type B in the other can be described using the union type $A \cup B$. Here we consider untagged unions rather than tagged unions, or sum types. Untagged unions are a good fit for dynamically typed languages, and both TypeScript [Bierman et al. 2014] and Flow [Chaudhuri et al. 2017] support union types.

Monitoring. We expect that monitoring a union type should have a similar rule as an intersection type, decomposing the two branches. Previous work by Keil and Thiemann [2015a] presents a single rule however the rule requires extracting the union type from within an intersection and constructing two new intersection contracts. In Section 3 we give an operational semantics with a single monitoring rule for union that does not require this extraction. Here, we present a simplified version for the purpose of explaining the examples.

$$V@^p A \cup B \longrightarrow V@^{p \bullet \mathsf{left}_\cup} A\, @^{p \bullet \mathsf{right}_\cup} B$$

Monitoring a union contract decomposes the contract into its constituent types with each new contract annotated with an augmented blame node. Branches can be violated without necessarily violating the union type. For example, consider the type $\mathtt{I} \cup \mathtt{B}$. No value can be both an integer and a

boolean, therefore no value can satisfy both these contracts. A value can be an integer or a boolean however, satisfying the union of the types. Blame resolution for union is similar to intersection in that it must record and interpret the state of previous violations.

Blame. There is a natural duality between intersection and union and one might expect the same duality to appear in blame resolution. The expectation is partially fulfilled, although blame resolution for union is not as simple as inverting blame resolution for intersection. To explain blame resolution for union we start with the static type rules.

$$\text{I-}\cup_i \ \frac{M : A_i \qquad i \in \{1,2\}}{M : A_1 \cup A_2} \qquad\qquad \text{E-}\cup \ \frac{\Gamma \vdash M : A \cup B \qquad \Gamma, x : A \vdash E[x] : C \qquad \Gamma, y : B \vdash E[y] : C}{\Gamma \vdash E[M] : C}$$

Introduction is admissible with subtyping but we choose to present the rule for symmetry. Multiple union elimination rules exist; we give the elimination rule by Dunfield and Pfenning [2003].

The introduction rule specifies that for a program, or subject, to satisfy a union type $A \cup B$ the subject must satisfy A or B. This introduces a choice of which type to satisfy, as was the case with intersection, however for union the choice belongs to the subject rather than the context. The elimination rule specifies that for elimination context E to satisfy a union type $A \cup B$ the context must satisfy types A and B individually.

Interpreting the type rules guides the formulation of blame resolution for union, resembling the negation of blame resolution for intersection. The key difference is between the case for positive union blame and the case for negative intersection blame. Both involve selecting a branch for the subject or context to satisfy respectively, however in the case for intersection this choice is made fresh at each elimination. For union this choice is made at the introduction of the type and should therefore apply to *all* uses of the contract. Consequently, blame resolution for union requires less information than intersection and is why the monitoring rule for union presented by Keil and Thiemann [2015a] is less involved than intersection. We state blame resolution for union.

- **+** Positive blame is assigned to a union type $A \cup B$ when positive blame is assigned to A *and* B. A program must satisfy at least one constituent of the union.
- **-** Negative blame is assigned to a union type $A \cup B$ when negative blame is assigned to A *or* B. A context that accepts a union type must accept both types individually, therefore the context must not violate either.

Keil and Thiemann [2015a] observed that higher-order union types have an interesting property: it is sometimes necessary to require multiple applications of the same function to detect a violation. Consider the following program:

$$((\lambda x.\texttt{if } x \texttt{ then 1 else } x)@^{p}(\mathtt{B} \to \mathtt{B}) \cup (\mathtt{B} \to \mathtt{I}))\ \texttt{true}$$

Calling the function with `true` will return an integer and assign blame to the left branch but not the right, which is insufficient to assign blame to the union type. An isolated call with input `false` will also have a similar outcome, instead blaming the right branch rather than the left. Either application alone will not detect the violation, only when the function is applied to `true` and `false` will positive blame be assigned to p and the union type. Note, unlike an intersection of function types, blame is accumulated across multiple uses of the function.

This example also illustrates the difference between types $(\mathtt{B} \to \mathtt{B}) \cup (\mathtt{B} \to \mathtt{I})$ and $\mathtt{B} \to (\mathtt{B} \cup \mathtt{I})$. The former has the union range over the function types so the choice of branch is made at the definition of the function; a union of functions is not free to alternate between return types. The latter has the union range over the return type so every application introduces a union; each value

$$
\begin{array}{lrcl}
\textit{Types} & A, B & ::= & A \cap B \mid A \cup B \mid A \to B \mid \iota \mid \mathsf{any} \\
\textit{Base Types} & \iota & ::= & \mathtt{I} \mid \mathtt{B} \\
\textit{Terms} & M, N & ::= & x \mid k \mid \lambda x.M \mid M\,N \mid \mathtt{blame}\ \pm\ell \mid M@^{p}A \\
\textit{Values} & V, W & ::= & k \mid \lambda x.M \mid V@^{p}A \to B \\
\textit{Continuations} & K & ::= & \mathit{Id} \mid K \circ \square\, N \mid K \circ V\, \square \mid K \circ \square\, @^{p}A \\
\textit{Natural Numbers} & n & \in & \mathbb{N}_0 \\
\textit{Blame Labels} & \ell & & \\
\textit{Branch Directions} & d & ::= & \mathsf{left} \mid \mathsf{right} \\
\textit{Branch Types} & \circ & ::= & \cap \mid \cup \\
\textit{Blame Paths} & P & ::= & \mathsf{nil} \mid \mathsf{dom}_n/P \mid \mathsf{cod}_n/P \\
\textit{Blame Nodes} & p, q & ::= & \pm\ell[P] \mid p \bullet d^{\pm}_{\circ}[P] \\
\textit{Blame States} & \Phi & ::= & \emptyset \mid \{\, p \,\} \mid \Phi \cup \Phi' \\
\textit{Context Trackers} & \Delta & = & p \rightharpoonup n \\
\textit{Configurations} & & ::= & \langle \Phi, \Delta, K, M \rangle
\end{array}
$$

Fig. 1. Syntax

returned from the function can select a different branch to satisfy. Our function does not satisfy type $(\mathtt{B} \to \mathtt{B}) \cup (\mathtt{B} \to \mathtt{I})$ but it *does* satisfy type $\mathtt{B} \to (\mathtt{B} \cup \mathtt{I})$.

3 A CALCULUS OF CONTRACTS FOR INTERSECTION AND UNION TYPES

This section presents the language λ_{IU}, an untyped lambda calculus with contracts for intersection and union types. Our operational semantics are specified using configurations of frame stacks (or continuations) and program terms to match the technical results in Section 5. This section focuses on the operational behaviour of contract monitoring; in Section 4 we add the semantics of blame.

3.1 Syntax

The syntax of λ_{IU} is given in Figure 1 and grouped into the categories: types, programs, blame tracking, and run-time.

Types. Let A and B range over types. Types are either intersection $A \cap B$, union $A \cup B$, function $A \to B$, base types ι, or the type any. Base types are integers $\mathtt{I}$ and booleans $\mathtt{B}$.

Programs. A program consists of terms, values, and continuations. Let M and N range over terms. Terms are either variables x, constants k, function abstraction $\lambda x.M$, function application $M\,N$, errors $\mathtt{blame}\ \pm\ell$, or terms under contract $M@^{p}A$.

Let V and W range over values. Values are either constants, function abstractions, or values wrapped with a function contract.

Let K range over continuations. Continuations are either the identity continuation Id or a continuation K with a frame appended: frames are terms with a hole. The remaining continuation forms are: argument continuations $K \circ \square\, N$, denoting a continuation that accepts a value and applies it to argument N; function continuations $K \circ V\, \square$, denoting a continuation that accepts a value to which V is applied; and contract continuations $K \circ \square\, @^{p}A$, denoting a continuation that accepts a value to which contract A is applied.

Blame Tracking. A blame node annotating a contract at run-time encodes the provenance of the contract in relation to the original source contract, detailing the operations that led to the

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

Fig. 2. Operational Semantics

construction of the contract. First we describe the components of blame nodes. Let ℓ range over blame labels, denoting distinct program identifiers. Let d range over branch directions left and right, denoting a branch of an intersection or union. Let $\circ$ range over $\cap$ and $\cup$. Let P range over blame paths, denoting a contract arising from a series of function contract wrappings. Paths are either the empty path nil, a path starting with a domain contract dom_n/P, or a path starting with a codomain contract cod_n/P. Each element in a path is indexed by a natural number n denoting which particular wrapping the contract came from. We require blame paths to record the provenance of a contract and correctly assign blame. Keil and Thiemann [2015a] have a similar notion of blame provenance but do not make the path explicit in the term. For a function contract $\mathtt{I} \to \mathtt{B}$, we use path $\mathsf{dom}_0/\mathsf{nil}$ to refer to the integer contract created in the first application of the function, while we use path $\mathsf{cod}_2/\mathsf{nil}$ to refer to the boolean contract created in the third application of the function.

Let p and q range over blame nodes where nodes are either a root $\pm\ell[P]$ or a branch $p \bullet d^{\pm}_{\circ}[P]$. A root includes a blame label and a blame path, while a branch has a pointer to parent node p, and includes type and direction information of the branch in addition to a path. When referring to blame nodes we use $\pm\ell$ as an abbreviation for $\pm\ell[\mathsf{nil}]$, and $p \bullet d^{\pm}_{\circ}$ as an abbreviation for $p \bullet d^{\pm}_{\circ}[\mathsf{nil}]$. A branch node is introduced at the elimination of every intersection or union contract, where parent p was associated with the intersection or union. Every root and branch includes a charge, positive (+) or negative (−), used to resolve blame. We write $\pm$ to range over positive and negative charges, and $\mp$ to indicate that a charge in a given position has been negated. For example, if we write $\pm\ell$, then $\mp\ell$ is the same blame label but with its charge negated.

Run-time. A program configuration $\langle \Phi, \Delta, K, M\rangle$ includes blame state Φ, context tracker Δ, continuation K, and term M. Blame states are sets of blame nodes where every node in the set has been assigned blame for a contract violation. A context tracker is a partial function from blame nodes to integers, recording how many times a function contract associated with a blame node has been applied. We write $\emptyset$ for the empty blame state and $\cdot$ for the empty context tracker.

3.2 Operational Semantics

The operational semantics of λ_{IU} is presented in Figure 2 using a small-step reduction relation. The relation has the form:

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

Negation

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

Path Extension

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

Wrap Index

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

where $c \in \{\mathrm{dom}, \mathrm{cod}\}$

Fig. 3. Wrap Operations

When a reduction leaves the blame state or context tracker unchanged we omit them from the source and target configurations. We let $\longrightarrow^*$ denote the transitive and reflexive closure of $\longrightarrow$.

The first three reductions are standard. If the term is an application $M\,N$ then create an argument continuation and evaluate M. If the continuation is an argument continuation $K \circ \Box\, N$ and the term is a value then create a function continuation and evaluate N. If the continuation is a function continuation $K \circ (\lambda x.M)\,\Box$ and the term is a value V then evaluate the lambda body with the bound variable x substituted for V.

If the term is a contract $M@^p A$ and M is not a value create a contract continuation and evaluate M. If the continuation is a contract continuation and the term is a value then reapply the contract to the value. The next rules handle the decomposition of contracts.

If the continuation is a function continuation $K \circ (V@^p A \to B)\,\Box$ and the term is a value then *wrap* the application. The wrap rule requires operations defined in Figure 3. Function δ returns the index of the application for blame node p and increments the index for p in context tracker Δ. Function $\gg$ extends a blame node's path with information about the application. The blame node attached to the codomain contract is extended by the operation $p \gg \mathrm{cod}_n$, denoting that the contract is the codomain for the n'th application of the contract annotated with p. The equivalent extension happens for the blame node attached to the domain contract in addition to negation of the node. Negation flips every charge in the root and branches of a blame node. We follow the convention of Keil and Thiemann [2015a] in that contracts for function types do not check that the value is a function, only the application behaviour is checked. A contract that checks both can be encoded using a combination of a base type and function type. We cover this in Section 5 and show why intersection is not the right operator for this task.

If the term is an intersection or union contract applied to a value then the contract is split. Each new contract is annotated with a branch blame node where p, the original node, acts as parent. The branch in each contract has a positive charge because the new contracts monitor the same subject as the original contract and the path for each node is empty because no applications have occurred.

If the term is the any contract applied to a value V then immediately return V as the any type accepts all values. If the term is a base type contract ι applied to a value V and the value conforms to type ι, denoted $V : \iota$, then return V. We omit the details of conformance but it is defined in the standard way. Integer constants conform to the type I and boolean constants conform to the type B. We assume that function contracts attached to a value do not affect conformance of that value.

If the term is a base type contract ι applied to a value V and the value does *not* conform to type ι then a violation is raised using $blame(p, \Phi, V)$. The function denotes blame being raised on node p in state Φ with offending value V. Section 4 describes the full semantics of blame; here, we only consider the effects of the contract violation. The function *blame* returns an updated state Φ' and term M to be used in the resulting configuration. The new state can only differ by the inclusion of

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

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

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

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

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

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

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

Fig. 4. Blame Semantics

violated node p, and possibly prefixes of p as blame propagates up any branches. The new term M can only be drawn from the set $\{V, \texttt{blame } \pm\ell\}$, where V is the offending value and ℓ is the label at the root of p. Whether the value or `blame` is returned depends on whether the blame state is sufficient to elicit a top-level violation: where blame propagates from a node to a root $\pm\ell[P]$.

If the term is `blame` $\pm\ell$ then the remaining continuation is discarded and the configuration reaches a final state.

4 BLAME

This section completes the description of the dynamic behaviour of λ_{IU}, providing the semantics to blame. Primary definitions are presented in Figure 4; auxiliary definitions are presented in Figure 5.

The operation $blame(\, p_{\pm\ell}, \Phi, V \,)$ is read: raise blame on node $p_{\pm\ell}$ in state Φ when value V triggered a violation. We write $p_{\pm\ell}$ to denote a blame node p with a root that has charge and label $\pm\ell$. Raising blame returns an updated blame state and the result of the contract violation. When assigning blame to $p_{\pm\ell}$ violates a top-level contract the result is an error, otherwise return the original value V. We define *blame* in terms of mutually recursive operations *assign* and *resolve*. Both *assign* and *resolve* return a modified blame state and a truth value indicating whether blame propagated to the root of the node, denoting a top-level violation. We describe the semantics of each in turn.

4.1 Blame Assignment

The operation $assign(\, p, \Phi \,)$ is responsible for determining whether p should be rightfully blamed for a violated contract. When the context of p has previously been assigned blame we ignore the current violation because a subject is only expected to satisfy a contract under the condition that the context satisfies the same contract. If a node *compatible* with $-p$ has already been assigned blame then do not assign blame to p, returning the existing blame state. If there is no such compatible node then blame is assigned to p by adding it to the blame state, blame is then resolved for p.

Compatibly, defined in Figure 5 (a), is denoted $compat(\, p, q \,)$ and specifies whether p and q stem from the same elimination context for a function contract. Root nodes are compatible if they are identical up to path compatibility. Branch nodes are compatible if they share the same parent and

(a)

$$\frac{compat(P, P')}{compat(\pm\ell[P], \pm\ell[P'])}$$

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

$$\frac{compat(P, P') \qquad c \in \{\mathsf{dom}, \mathsf{cod}\}}{compat(c_n/P, c_n/P')}$$

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

(b)

$$\frac{c, c' \in \{\mathsf{dom}, \mathsf{cod}\}}{elim(c_n/P, c'_n/P')}$$

(c)

$$flip(\mathsf{left}) = \mathsf{right}$$
$$flip(\mathsf{right}) = \mathsf{left}$$

(d)

$$parent(p \bullet d^{\pm}_{\circ}[\mathsf{nil}] \bullet d'^{\pm}_{\sigma'}[P']) = p \bullet d^{\pm}_{\circ}[P']$$
$$parent(p \bullet d^{\pm}_{\circ}[P']) = p \quad \text{otherwise}$$

Fig. 5. Auxiliary Blame Definitions

the branches are identical up to path compatibility. Paths are compatible if they share a common prefix and then diverge at the same wrap index. Consider the following example that shows how compatibility enables some contract violations to be disregarded:

$$\textit{let } a = +\ell \bullet \mathsf{left}^{+}_{\cap}; \quad b = -\ell \bullet \mathsf{left}^{-}_{\cap}[\mathsf{dom}_0/\mathsf{nil}]; \quad c = +\ell \bullet \mathsf{left}^{+}_{\cap}[\mathsf{cod}_0/\mathsf{nil}]$$
$$\langle \emptyset, \cdot, K, ((\lambda y.y\ \mathtt{false})@^{a}(\mathtt{B} \to \mathtt{I}) \to \mathtt{I})\ \lambda x.\mathsf{if}\ x\ \mathsf{then}\ 1\ \mathsf{else}\ x \rangle$$
$$\longrightarrow^{*} \quad \langle \emptyset, [a \mapsto 1], K, ((\lambda y.y\ \mathtt{false})\ ((\lambda x.\mathsf{if}\ x\ \mathsf{then}\ 1\ \mathsf{else}\ x)@^{b}\mathtt{B} \to \mathtt{I}))@^{c}\mathtt{I} \rangle \tag{1a}$$
$$\longrightarrow^{*} \quad \langle \emptyset, [a \mapsto 1], K \circ \square\ @^{c}\mathtt{I}, ((\lambda x.\mathsf{if}\ x\ \mathsf{then}\ 1\ \mathsf{else}\ x)@^{b}\mathtt{B} \to \mathtt{I})\ \mathtt{false} \rangle \tag{1b}$$
$$\textit{let } d = +\ell \bullet \mathsf{left}^{+}_{\cap}[\mathsf{dom}_0/\mathsf{dom}_0/\mathsf{nil}]; \quad e = -\ell \bullet \mathsf{left}^{-}_{\cap}[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]$$
$$\longrightarrow^{*} \quad \langle \emptyset, [a \mapsto 1; b \mapsto 1], K \circ \square\ @^{c}\mathtt{I}, ((\lambda x.\mathsf{if}\ x\ \mathsf{then}\ 1\ \mathsf{else}\ x)\ (\mathtt{false}@^{d}\mathtt{B}))@^{e}\mathtt{I} \rangle \tag{1c}$$
$$\longrightarrow^{*} \quad \langle \emptyset, [a \mapsto 1; b \mapsto 1], K \circ \square\ @^{c}\mathtt{I}, \mathtt{false}@^{e}\mathtt{I} \rangle \tag{1d}$$
$$\longrightarrow \quad \langle \{e\}, [a \mapsto 1; b \mapsto 1], K \circ \square\ @^{c}\mathtt{I}, \mathtt{false} \rangle \tag{1e}$$
$$\longrightarrow \quad \langle \{e\}, [a \mapsto 1; b \mapsto 1], K, \mathtt{false}@^{c}\mathtt{I} \rangle \tag{1f}$$
$$\longrightarrow \quad \langle \{e\}, [a \mapsto 1; b \mapsto 1], K, \mathtt{false} \rangle \tag{1g}$$

In the example a wrapped function annotated with blame node $+\ell \bullet \mathsf{left}^{+}_{\cap}$, abbreviated a, is applied to another function. Reduction (1a) wraps the application and updates the context tracker for a. Reduction (1b) pushes the resulting codomain contract to the continuation and then applies beta-reduction. Reduction (1c) wraps the application of function $\lambda x.\mathsf{if}\ x\ \mathsf{then}\ 1\ \mathsf{else}\ x$, creating domain and codomain contracts annotated with abbreviated blame nodes d and e. The context tracker is updated to denote that the function contract annotated with blame node b has been applied once. Reduction (1d) proceeds as follows: apply the boolean contract to `false`; beta-reduce the application; and evaluate the conditional, selecting the else clause. Reduction (1e) applies an integer contract to `false`, triggering a violation. The corresponding *blame* function is $blame(e, \emptyset, \mathtt{false})$, which first assigns blame to e, or $-\ell \bullet \mathsf{left}^{-}_{\cap}[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]$, as there are no compatible violations.

$$assign(e, \emptyset) = resolve(e, \{e\}) \quad \textit{where } e = -\ell \bullet \mathsf{left}^{-}_{\cap}[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]$$

Next, we try to resolve blame for e which corresponds to resolving negative blame for an intersection branch. We explore blame resolution in more detail later. For now, observe that there are no existing violations for the right branch so conditions are insufficient to propagate blame to the parent. Blame

resolution returns the blame state and indicates that no top-level violation was found.

$$\mathit{resolve}(e, \{e\}) = \{e\}, \bot \quad \textit{where } e = -\ell \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]$$

Reduction (1f) pops a contract frame from the continuation and applies it to the currently evaluating value. Reduction (1g) applies another integer contract to `false`, this time annotated with c. Blame node c, or $+\ell \bullet \mathsf{left}^+_\cap[\mathsf{cod}_0/\mathsf{nil}]$, denotes the codomain of the initial function contract. The corresponding *blame* function is $\mathit{blame}(c, \{e\}, \mathtt{false})$, which attempts to assign blame to c in state $\{e\}$. Observe that the negation of c is blame node $-\ell \bullet \mathsf{left}^-_\cap[\mathsf{cod}_0/\mathsf{nil}]$, which is compatible with $-\ell \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]$, or node e. Blame assignment fails as a consequence, returning the same blame state and indicating that no top-level violation was found.

$$\mathit{assign}(+\ell \bullet \mathsf{left}^+_\cap[\mathsf{cod}_0/\mathsf{nil}], \{-\ell \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]\}) = \{-\ell \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]\}, \bot$$

The contract application yields value `false` and judges blame node c to be innocent. The intuition is that the initial function is expected to return an integer, which it does not, but only under the assumption that the input function returns an integer, which it does not. The assignment of blame to node e, or $-\ell \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]$, is evidence that the context violates this assumption.

Compatibility must be carefully defined because not every contract violation caused by a context permits a subject to ignore the contract. We define compability to only consider the longest matching prefix between paths, and paths must diverge at the same wrap index. Consider the following example that extends the previous example by applying y twice:

$$\begin{aligned}
&\textit{let } a = +\ell \bullet \mathsf{left}^+_\cap; \quad b = -\ell \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]; \quad c = +\ell \bullet \mathsf{left}^+_\cap[\mathsf{cod}_0/\mathsf{nil}] \\
&V = \lambda x.\mathsf{if}\ x\ \mathsf{then}\ 1\ \mathsf{else}\ x \\
&\langle \emptyset, \cdot, K, ((\lambda y.y\,\mathtt{false}; y\,42)@^a(\mathtt{B} \to \mathtt{I}) \to \mathtt{I})\,V\rangle \\
\longrightarrow^* \quad &\langle \emptyset, [a \mapsto 1], K \circ \square\,@^c\mathtt{I}, (V@^b\mathtt{B} \to \mathtt{I})\,\mathtt{false}; (V@^b(\mathtt{B} \to \mathtt{I}))\,42\rangle && \text{(2a)} \\
&\textit{let } d = +\ell \bullet \mathsf{left}^+_\cap[\mathsf{dom}_0/\mathsf{dom}_0/\mathsf{nil}]; \quad e = -\ell \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}] \\
\longrightarrow^* \quad &\langle \{e\}, [a \mapsto 1; b \mapsto 1], K \circ \square\,@^c\mathtt{I}, (V@^b(\mathtt{B} \to \mathtt{I}))\,42\rangle && \text{(2b)} \\
&\textit{let } f = +\ell \bullet \mathsf{left}^+_\cap[\mathsf{dom}_0/\mathsf{dom}_1/\mathsf{nil}]; \quad g = -\ell \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{cod}_1/\mathsf{nil}] \\
\longrightarrow^* \quad &\langle \{e\}, [a \mapsto 1; b \mapsto 2], K \circ \square\,@^c\mathtt{I}, (V\,(42@^f\mathtt{B}))@^g\mathtt{I}\rangle && \text{(2c)} \\
\longrightarrow^* \quad &\langle \{e\}, [a \mapsto 1; b \mapsto 2], K \circ \square\,@^c\mathtt{I} \circ \square\,@^g\mathtt{I} \circ V\,\square, 42@^f\mathtt{B}\rangle && \text{(2d)} \\
\longrightarrow \quad &\langle \{e, f, +\ell\}, [a \mapsto 1; b \mapsto 2], K \circ \square\,@^c\mathtt{I} \circ \square\,@^g\mathtt{I} \circ V\,\square, \mathtt{blame}\ +\ell\rangle && \text{(2e)}
\end{aligned}$$

Evaluation begins in a similar manner to the previous example. Reduction (2a) wraps the application and applies beta-reduction. Reduction (2b) evaluates the first application of y in the same way as the previous example. Reduction (2c) wraps the second application of y. Note how the context tracker now indicates that the function contract annotated with b has been applied twice. Reduction (2d) adds two frames to the continuation, leaving the wrapped argument as the currently evaluating term. Reduction (2e) evaluates the contract application, raising blame. The corresponding *blame* function for this reduction is $\mathit{blame}(f, \{e\}, \mathtt{false})$. In this situation the previous negative blame assigned to e should not invalidate the positive blame raised against f. The intuition is that the second application of y to illegal value 42 cannot be predicated on the context supplying an illegal argument. A function should respect its argument in all contexts without assuming ill-behaviour of the argument. Compatibility rejects this case by only considering the smallest possible context, denoted by the longest prefix, that might contain the paths. Blame is assigned to f because $-f$ and

e are not compatible: they share a prefix but diverge at different applications.

$$\mathit{assign}(f, \{e\}) = \mathit{resolve}(f, \{e, f\})$$
$$\text{where } e = -\ell \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{cod}_0/\mathsf{nil}]; \quad f = +\ell \bullet \mathsf{left}^+_\cap[\mathsf{dom}_0/\mathsf{dom}_1/\mathsf{nil}]$$

Blame node f corresponds to a positive intersection branch so blame always propagates to the parent. The parent of f is root $+\ell$ therefore resolution indicates that an error should be raised.

$$\mathit{resolve}(f, \{e, f\}) = \mathit{assign}(+\ell, \{e, f\}) = \mathit{resolve}(+\ell, \{e, f, +\ell\}) = \{e, f, +\ell\}, \top$$

The reduction yields $\texttt{blame}\ +\ell$, indicating that the function has violated contract $(\mathtt{B} \to \mathtt{I}) \to \mathtt{I}$.

4.2 Blame Resolution

The operation $\mathit{resolve}(p, \Phi)$ is responsible for propagating blame towards the root to find top-level contract violations. At each level the blame state is queried to determine whether blame that is assigned to a branch is sufficient to propagate to the parent intersection or union.

When blaming a root $\pm\ell[P]$ return the blame state and $\top$, indicating that blame has propagated to the top-level.

When blaming a positive intersection branch $p \bullet d^+_\cap[P]$, or a negative union branch $p \bullet d^-_\cup[P]$, blame is propagated by assigning blame to the parent of the node. Both cases have no condition on propagation because blame is assigned to an intersection or union when *either* branch is assigned positive or negative blame respectively.

When blaming a negative intersection branch $p \bullet d^-_\cap[P]$ blame is propagated to the parent if the other branch has also been assigned negative blame in the same elimination of the intersection. This condition is satisfied when a node with the same parent and inverse branch direction exists in the blame state, whilst also having a path that starts with the same elimination context, denoted $\mathit{elim}(P, P')$. Figure 5 defines *elim* (b) and *flip* (c). Note that *elim* only examines the first context in a path because this is where the intersection is eliminated. Consider the following program where a context violates an intersection contract:

$$\begin{aligned}
& \mathit{let}\ a = +\ell \bullet \mathsf{left}^+_\cap; \quad b = +\ell \bullet \mathsf{right}^+_\cap \\
& \langle \emptyset, \emptyset, K, ((\lambda x.x)@^{+\ell}(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B}))\ \text{“foo”} \rangle \\
\longrightarrow^* \quad & \langle \emptyset, \emptyset, K \circ \square\ \text{“foo”}, ((\lambda x.x)@^a \mathtt{I} \to \mathtt{I})@^b \mathtt{B} \to \mathtt{B} \rangle && \text{(3a)} \\
& \mathit{let}\ c = -\ell \bullet \mathsf{right}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]; \quad d = +\ell \bullet \mathsf{right}^+_\cap[\mathsf{cod}_0/\mathsf{nil}] \\
\longrightarrow^* \quad & \langle \emptyset, [b \mapsto 1], K, (((\lambda x.x)@^a \mathtt{I} \to \mathtt{I})\,(\text{“foo”}@^c \mathtt{B}))@^d \mathtt{B} \rangle && \text{(3b)} \\
\longrightarrow^* \quad & \langle \{c\}, [b \mapsto 1], K \circ \square\ @^d \mathtt{B} \circ (\lambda x.x)@^a \mathtt{I} \to \mathtt{I}\ \square, \text{“foo”} \rangle && \text{(3c)} \\
& \mathit{let}\ e = -\ell \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]; \quad f = +\ell \bullet \mathsf{left}^+_\cap[\mathsf{cod}_0/\mathsf{nil}] \\
\longrightarrow \quad & \langle \{c\}, [b \mapsto 1; a \mapsto 1], K \circ \square\ @^d \mathtt{B}, ((\lambda x.x)\,(\text{“foo”}@^e \mathtt{I}))@^f \mathtt{I} \rangle && \text{(3d)} \\
\longrightarrow^* \quad & \langle \{c\}, [b \mapsto 1; a \mapsto 1], K \circ \square\ @^d \mathtt{B} \circ \square\ @^f \mathtt{I} \circ \lambda x.x\ \square, \text{“foo”}@^e \mathtt{I} \rangle && \text{(3e)} \\
\longrightarrow \quad & \langle \{c, e, -\ell\}, [b \mapsto 1; a \mapsto 1], K \circ \square\ @^d \mathtt{B} \circ \square\ @^f \mathtt{I} \circ \lambda x.x\ \square, \texttt{blame}\ -\ell \rangle && \text{(3f)}
\end{aligned}$$

Reduction (3a) splits the intersection contract. Reduction (3b) wraps the application for the right branch. Reduction (3c) applies the domain contract from the right branch. Blame is assigned to c as the argument is a string not a boolean, however, blame does not propagated from branch to parent. Node c denotes a negative intersection branch and blame is only propagated if the other branch has also been blamed, which is not the case. Reduction (3d) wraps the application for the left branch. Reduction (3e) adds two frames to the continuation, leaving the wrapped argument as the currently evaluating term. Reduction (3f) applies the domain contract from the left branch,

triggering a violation. The corresponding *blame* function for this reduction is $\mathit{blame}(e, \{c\}, \text{“foo”})$. Blame is first assigned to e, a negative intersection branch, then blame must be resolved.

$$\mathit{assign}(e, \{c\}) = \mathit{resolve}(e, \{c, e\})$$

We observe that flipping the direction of e, or $-\ell \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]$, yields blame node $-\ell \bullet \mathsf{right}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]$ which is equivalent to node c. Both branches have been assigned negative blame in the same elimination context therefore blame is propagated to parent $-\ell$. Assigning blame to the root indicates that an error should be raised.

$$\mathit{resolve}(e, \{c, e\}) = \mathit{assign}(-\ell, \{c, e\}) = \mathit{resolve}(-\ell, \{c, e, -\ell\}) = \{c, e, -\ell\}, \top$$

The reduction yields $\texttt{blame}\ -\ell$, indicating that the context has violated contract $(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B})$.

When blaming a positive union branch $p \bullet d^+_\cup[P]$ blame is propagated to the parent if the other branch has also been assigned positive blame. Positive blame for a union type is aggregated across all uses so there is no constraint on path P and P'. Positive blame for union is less constrained than negative blame for intersection. Assigning blame to nodes $-\ell \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]$ and $-\ell \bullet \mathsf{right}^-_\cap[\mathsf{dom}_1/\mathsf{nil}]$ would not blame $-\ell$ as the eliminations contexts differ, while assigning blame to nodes $+\ell \bullet \mathsf{left}^+_\cup[\mathsf{cod}_0/\mathsf{nil}]$ and $+\ell \bullet \mathsf{right}^+_\cup[\mathsf{cod}_1/\mathsf{nil}]$ would blame $+\ell$.

When propagating blame from a branch we extract the parent from a node p using operation $\mathit{parent}(p)$ defined in Figure 5 (d). If the parent of p is a branch node with an empty blame path then the parent is returned after hoisting the child's path to the parent. In all other cases the parent of p is returned without modification. Hoisting blame paths is necessary when using nested intersection or union. Consider the following example:

$$\begin{aligned} & \langle K, (V @^p ((\mathtt{I} \to \mathtt{I}) \cup (\mathtt{I} \to \mathtt{B})) \cap (\mathtt{B} \to \mathtt{B})) \text{“foo”} \rangle \\ \longrightarrow^* \quad & \langle K \circ \square \text{“foo”}, ((V @^{p \bullet \mathsf{left}^+_\cap \bullet \mathsf{left}^+_\cup} (\mathtt{I} \to \mathtt{I})) @^{p \bullet \mathsf{left}^+_\cap \bullet \mathsf{right}^+_\cup} (\mathtt{I} \to \mathtt{B})) @^{p \bullet \mathsf{right}^+_\cap} (\mathtt{B} \to \mathtt{B}) \rangle \end{aligned}$$

Evaluation splits the contract into three components. As the argument violates the domain contract for each component blame will be assigned to nodes $-p \bullet \mathsf{left}^-_\cap \bullet \mathsf{right}^-_\cup[\mathsf{dom}_0/\mathsf{nil}]$ and $-p \bullet \mathsf{right}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]$. Assigning negative blame to the right union branch is sufficient to blame the union, propagating to the left intersection branch. However, if we were to naively blame the parent then blame would be assigned to $-p \bullet \mathsf{left}^-_\cap$. This node would not match the node $-p \bullet \mathsf{right}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]$ on the right because of the lost path information, and incorrectly, would not assign blame to $-p$. By hoisting blame paths we blame the left intersection branch using node $-p \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]$, correctly matching the node blamed on the right and assigning blame to $-p$.

4.3 Blame by Example

We revisit the example from the introduction to present an extended demonstration of blame. We assume a predicate function (M is $\mathtt{B}$) in place of using the JavaScript operator `typeof`. Additionally, we have insidiously introduced a bug. Figure 6 shows the reduction sequence.

Reduction (4a) splits the intersection contract. Reduction (4b) wraps the application of the function contract in the right branch. Reduction (4c) applies domain contract $\mathtt{I}$, succeeding, then adds two frames to the continuation. Reduction (4d) wraps the application of the function contract in the left branch. Reduction (4e) adds two frames to the continuation, leaving the wrapped argument as the currently evaluating term. Reduction (4f) applies the boolean contract, triggering a violation. The corresponding *blame* function for this reduction is $\mathit{blame}(e, \emptyset, 42)$. Blame is first assigned to e, a negative intersection branch, then blame must be resolved. This is the first violation therefore blame is not propagated to the parent of e.

$$\mathit{blame}(e, \emptyset, 42) = \{e\}, 42 \quad \text{as } \mathit{assign}(e, \emptyset) = \mathit{resolve}(e, \{e\}) = \{e\}, \bot$$

$$
\begin{array}{lll}
 & \textit{let } a = +\ell \bullet \mathsf{left}^+_\cap; \quad b = +\ell \bullet \mathsf{right}^+_\cap & \\
 & V = \lambda x.\mathsf{if}\ (x\ \mathsf{is}\ \mathsf{B})\ \mathsf{then}\ (\mathsf{if}\ x\ \mathsf{then}\ \text{“hello world”}\ \mathsf{else}\ !x)\ \mathsf{else}\ \lambda y.x + 10 & \\
 & \langle \emptyset, \cdot, K, (V@^{+\ell}(\mathsf{B} \to (\mathsf{S} \cup \mathsf{B})) \cap (\mathsf{I} \to \mathsf{I}))\ 42 \rangle & \\
\longrightarrow^* & \langle \emptyset, \cdot, K \circ \square\ 42, (V@^a\mathsf{B} \to (\mathsf{S} \cup \mathsf{B}))@^b\mathsf{I} \to \mathsf{I} \rangle & (4a) \\
 & \textit{let } c = -\ell \bullet \mathsf{right}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]; \quad d = +\ell \bullet \mathsf{right}^+_\cap[\mathsf{cod}_0/\mathsf{nil}] & \\
\longrightarrow^* & \langle \emptyset, [b \mapsto 1], K, ((V@^a\mathsf{B} \to (\mathsf{S} \cup \mathsf{B}))\ (42@^c\mathsf{I}))@^d\mathsf{I} \rangle & (4b) \\
\longrightarrow^* & \langle \emptyset, [b \mapsto 1], K \circ \square\ @^d\mathsf{I} \circ (V@^a\mathsf{B} \to (\mathsf{S} \cup \mathsf{B}))\ \square, 42 \rangle & (4c) \\
 & \textit{let } e = -\ell \bullet \mathsf{left}^-_\cap[\mathsf{dom}_0/\mathsf{nil}]; \quad f = +\ell \bullet \mathsf{left}^+_\cap[\mathsf{cod}_0/\mathsf{nil}] & \\
\longrightarrow & \langle \emptyset, [b \mapsto 1; a \mapsto 1], K \circ \square\ @^d\mathsf{I}, (V\ (42@^e\mathsf{B}))@^f(\mathsf{S} \cup \mathsf{B}) \rangle & (4d) \\
\longrightarrow^* & \langle \emptyset, [b \mapsto 1; a \mapsto 1], K \circ \square\ @^d\mathsf{I} \circ \square\ @^f(\mathsf{S} \cup \mathsf{B}) \circ V\ \square, 42@^e\mathsf{B} \rangle & (4e) \\
\longrightarrow & \langle \{e\}, [b \mapsto 1; a \mapsto 1], K \circ \square\ @^d\mathsf{I} \circ \square\ @^f(\mathsf{S} \cup \mathsf{B}) \circ V\ \square, 42 \rangle & (4f) \\
\longrightarrow^* & \langle \{e\}, [b \mapsto 1; a \mapsto 1], K \circ \square\ @^d\mathsf{I} \circ \square\ @^f(\mathsf{S} \cup \mathsf{B}), \lambda y.42 + 10 \rangle & (4g) \\
 & \textit{let } g = +\ell \bullet \mathsf{left}^+_\cap[\mathsf{cod}_0/\mathsf{nil}] \bullet \mathsf{left}^+_\cup; \quad h = +\ell \bullet \mathsf{left}^+_\cap[\mathsf{cod}_0/\mathsf{nil}] \bullet \mathsf{right}^+_\cup & \\
\longrightarrow^* & \langle \{e\}, [b \mapsto 1; a \mapsto 1], K \circ \square\ @^d\mathsf{I}, ((\lambda y.42 + 10)@^g\mathsf{S})@^h\mathsf{B} \rangle & (4h) \\
\longrightarrow^* & \langle \{e, g\}, [b \mapsto 1; a \mapsto 1], K \circ \square\ @^d\mathsf{I}, (\lambda y.42 + 10)@^h\mathsf{B} \rangle & (4i) \\
\longrightarrow^* & \langle \{e, g, h\}, [b \mapsto 1; a \mapsto 1], K \circ \square\ @^d\mathsf{I}, \lambda y.42 + 10 \rangle & (4j) \\
\longrightarrow & \langle \{e, g, h\}, [b \mapsto 1; a \mapsto 1], K, (\lambda y.42 + 10)@^d\mathsf{I} \rangle & (4k) \\
\longrightarrow & \langle \{e, g, h, d, +\ell\}, [b \mapsto 1; a \mapsto 1], K, \mathtt{blame}\ +\ell \rangle & (4l)
\end{array}
$$

Fig. 6. Extended Example

The blame function returns 42 in the new blame state, and evaluation continues. Reduction (4g) applies beta-reduction and evaluates the function body. The else clause of the outer conditional is selected and the abstraction $\lambda y.42 + 10$ is returned. Reduction (4h) applies the union contract from the codomain of the left branch, splitting into two sub-contracts. Reduction (4i) applies the contract in the left branch of the union, triggering a violation. The corresponding *blame* function for this reduction is $\mathit{blame}(g, \{e\}, \lambda y.42 + 10)$. Blame is first assigned to g because $-g$ is not compatible with e (they do not share a parent). Blame is resolved for g, which corresponds to a union left branch. There is no corresponding violation for the right branch so blame is not propagated to the parent.

$$\mathit{blame}(g, \{e\}, \lambda y.42 + 10) = \{e, g\}, \lambda y.42 + 10 \quad \text{as } \mathit{assign}(g, \{e\}) = \mathit{resolve}(g, \{e, g\}) = \{e, g\}, \bot$$

Reduction (4j) applies the contract in the right branch of the union, triggering a violation. The corresponding *blame* function for this reduction is $\mathit{blame}(h, \{e, g\}, \lambda y.42 + 10)$. Like g, there is no node compatible with $-h$ therefore blame is assigned to h. When resolving blame we observe that there is an existing violation for the left union branch (node g), therefore blame is propagated to the parent of h as the union contract has been violated.

$$\mathit{assign}(h, \{e, g\}) = \mathit{resolve}(h, \{e, g, h\}) = \mathit{assign}(f, \{e, g, h\}) \quad \text{where } \mathit{parent}(h) = f$$

When assigning blame to the union contract, denoted by node f, we observe that the context has previously violated the domain contract in the left intersection branch. This indicates that it is safe

to ignore the violation of the union contract because the context has chosen not to select the left branch. Node $-f$ is compatible with node e which has previously been assigned blame, therefore blame assignment halts without further propagation.

$$\mathit{assign}(f, \{e, g, h\}) = \{e, g, h\}, \bot \quad \text{as } \mathit{compat}(-f, e)$$

Reduction (4k) extracts the contract frame from continuation. Reduction (4l) applies the contract from the codomain of the right branch, triggering a violation. The corresponding *blame* function for this reduction is $\mathit{blame}(d, \{e, g, h\}, \lambda y.42 + 10)$. Blame is assigned to d as there are no existing violations by the context for the right branch of the intersection. We observe that node d is a positive intersection branch so blame is propagated unconditionally.

$$\mathit{assign}(d, \{e, g, h\}) = \mathit{resolve}(d, \{e, g, h, d\}) = \mathit{assign}(+\ell, \{e, g, h, d\}) \quad \text{where } \mathit{parent}(d) = +\ell$$

Blame is assigned to the root, indicating that a top-level violation has occurred.

$$\mathit{assign}(+\ell, \{e, g, h, d\}) = \mathit{resolve}(+\ell, \{e, g, h, d, +\ell\}) = \{e, g, h, d, +\ell\}, \top$$

Evaluation raises `blame` $+\ell$ indicating that the function was at fault.

5 CONTRACT SEMANTICS AND TECHNICAL RESULTS

This section presents our main technical results for monitoring contracts with higher-order intersection and union types. We proceed by defining contract satisfaction using sets of closed values that positively satisfy a type and sets of closed continuations that negatively satisfy a type. We then give a series of sound monitoring properties about satisfaction that we expect a system to obey. These properties ensure that the monitoring semantics for a type are in alignment with the static semantics one might expect.

5.1 Satisfaction

The definitions for contract satisfaction are defined in Figure 7. We write $[\![A]\!]^+_p$ to denote the set of closed values that positively satisfy type A for witness p and write $[\![B]\!]^-_p$ to denote the set of closed continuations that negatively satisfy type B for witness p—we refer to this as *witness satisfaction*. A witness is a blame node that monitors for contract satisfaction, where assigning blame to the node can be thought of as witnessing a satisfaction counter-example.

A value V is in the set $[\![A]\!]^+_p$ if for all program configurations where p is fresh, monitoring V with blame node p at type A never evaluates to a blame configuration for p, denoted $\langle \lightning\, p \rangle$. We write $p \# \langle \Phi, \Delta, K, V \rangle$ to denote that p is fresh for the configuration. A blame node p is fresh if there exists no blame node q (positive or negative) in the configuration that is greater than (or equal to) p according to blame ordering, denoted by $p \leq \pm q$, or that is less than (or equal to) p according to context ordering, denoted by $\pm q \leq_\Delta p$. The ordering is required because freshness defined in terms of strict equality does not hold during reduction. Blame resolution may blame a prefix that matches an initially fresh node, or a contract may decompose into branches that match an initially fresh node. Blame ordering prevents the former; context ordering prevents the latter. Context ordering takes into account context tracker Δ, permitting us a degree of flexibility. For example, we may consider blame node $(p \gg \mathrm{cod}_n)$ fresh even in the presence of p, under the condition that we make the context tracker sufficiently large to ensure we never extend p using index n.

A blame configuration is any program configuration where the blame state Φ implicates p, denoted $\Phi \models p$. A blame state implicates a blame node if the node, up to any extension of its path, is included in the blame state. This condition means that blame for any application that stems from the initial contract annotated with p will implicate p. We add a second condition that states that a node is only implicated if there does not exist another node already in the blame state with

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

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

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

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

$$\begin{array}{lcl}
p \# \langle \Phi, \Delta, K, V \rangle & = & \nexists q. \in \{\Delta, K, V\} \text{ s.t. } (p \leq \pm q) \vee (\pm q \leq_\Delta p) \text{ and } \nexists q. \in \Phi \text{ s.t. } (p \leq \pm q) \\
\Phi \models \pm \ell[P] & = & (\exists P'. \pm \ell[P'] \in \Phi \wedge \mathit{prefix}(P, P')) \wedge (\nexists P'. \pm \ell[P'] \in \Phi \wedge \mathit{compat}(P, P')) \\
\Phi \models p \bullet d_\circ^\pm[P] & = & (\exists P'. p \bullet d_\circ^\pm[P'] \in \Phi \wedge \mathit{prefix}(P, P')) \wedge (\nexists P'. p \bullet d_\circ^\pm[P'] \in \Phi \wedge \mathit{compat}(P, P')) \\
\langle \lightning p \rangle & = & \langle \Phi, \Delta, K, M \rangle \text{ for some } \Phi, \Delta, K, M \text{ where } \Phi \models p \\
\\
V \in [\![A]\!]_p^+ & \stackrel{\text{def}}{=} & \forall \Phi, \Delta, K. \langle \Phi, \Delta, K, V @^p A \rangle \rightarrowtail\!\!\!\!\twoheadrightarrow^* \langle \lightning p \rangle \qquad \text{when } p \# \langle \Phi, \Delta, K, V \rangle \\
K \in [\![B]\!]_p^- & \stackrel{\text{def}}{=} & \forall \Phi, \Delta, V. \langle \Phi, \Delta, K, V @^p B \rangle \rightarrowtail\!\!\!\!\twoheadrightarrow^* \langle \lightning -p \rangle \qquad \text{when } p \# \langle \Phi, \Delta, K, V \rangle \\
V \in [\![A]\!]^+ & \stackrel{\text{def}}{=} & \forall p. V \in [\![A]\!]_p^+ \\
K \in [\![B]\!]^- & \stackrel{\text{def}}{=} & \forall p. K \in [\![B]\!]_p^-
\end{array}$$

Fig. 7. Satisfaction and Auxiliary Definitions

a compatible path. This condition means that blame for any application that p stems from will prevent p from being implicated; we only consider cases where p is the first violation associated with any application. The condition is mostly technical because in practice we only care about the first violation of a function type during an application. We justify the condition because adding p to the blame state when a compatible node already exists will not affect the program behaviour (blame assignment is the same with-or-without the inclusion of p).

The negative satisfaction set, denoted $[\![B]\!]_p^-$, is defined similarly. The differences with negative witness satisfaction is that we quantify over all values (fresh for p) instead of continuations and seek to avoid negative blame configurations $\langle \lightning -p \rangle$ rather than positive.

Witness satisfaction gives us a fine granularity of control that is useful for reasoning about satisfaction involving multiple components such as function contracts, where satisfaction of the codomain is predicated on satisfaction of the domain. However, we would also like a definition of satisfaction that is independent of any particular witness so that we may freely monitor a value or continuation using any contract.

We define general positive and negative satisfaction, denoted $[\![A]\!]^+$ and $[\![B]\!]^-$ respectively. The definition mirrors the presentation of satisfaction by Keil and Thiemann [2015a] though the semantics are significantly different. Their definition of satisfaction makes no mention of contracts, whilst ours is defined entirely in terms of monitoring behaviour. Our definition states that a value (continuation) positively (negatively) satisfies a type if the value (continuation) is in the witness satisfaction set *for all* possible witnesses: there is no witness that is able to discover a counter-example to satisfaction.

The definition of positive satisfaction can be extended to include terms in addition to values using the same definition. Theorems 5.1 and 5.2 relate satisfaction for terms and values.

Theorem 5.1 (Term Witness Satisfaction). *If* $M \longrightarrow^* V$ *then* $V \in [\![A]\!]^+_p \Rightarrow M \in [\![A]\!]^+_p$.

Theorem 5.2 (Term Satisfaction). *If* $M \longrightarrow^* V$ *then* $V \in [\![A]\!]^+$ *iff* $M \in [\![A]\!]^+$.

Note that the relation between values and terms is weaker for witness satisfaction than satisfaction generally. The reason is that in reducing a term M to a value V we might elicit negative violations that mean when monitoring $V@^p A$, p can never be assigned blamed. When monitoring the reduced value V in isolation, per definition $V \in [\![A]\!]^+_p$, we cannot assume the negative violation is in the blame state and therefore $V@^p A$ may assign blame to p. General satisfaction gives us a stronger property because it quantifies over all possible witnesses and therefore the satisfaction of M cannot be predicated on eliciting a negative violation for a particular witness.

5.2 Soundness

Contract soundness states the essential property that contracts enforce satisfaction. Applying a contract to a term always delivers a satisfying term and applying a contract to a continuation always delivers a satisfying continuation.

Theorem 5.3 (Contract Soundness).
- $M@^{\pm\ell[P]}A \in [\![A]\!]^+$
- $K \circ \square\ @^{\pm\ell[P]}B \in [\![B]\!]^-$

For example, we expect that 42 should not satisfy type B, however applying a contract of type B to the value will produce a satisfying program. Concretely, while 42 is not in the set $[\![\mathsf{B}]\!]^+$, program $42@^{+\ell}\mathsf{B}$ is. Assuming $42@^{+\ell}\mathsf{B} \in [\![\mathsf{B}]\!]^+$ then contract satisfaction states that the following must hold:

$$\langle K, (42@^{+\ell}\mathsf{B})@^{+\ell'}\mathsf{B}\rangle \longrightarrow\!\!\!\!\!\rightarrow^* \langle \mathit{Id}, \mathtt{blame}\ +\ell'\rangle$$

Note that contract satisfaction guarantees nothing about blame assignment for $+\ell$, only that positive blame is not assigned to the outer contract annotated with $+\ell'$. In this instance assigning blame to $+\ell$ is crucial as it generates an error, preventing evaluation of the outer contract:

$$\langle \Phi, K, (42@^{+\ell}\mathsf{B})@^{+\ell'}\mathsf{B}\rangle \longrightarrow^* \langle \Phi \cup \{+\ell\}, K \circ \square\ @^{+\ell'}\mathsf{B}, \mathtt{blame}\ +\ell\rangle \longrightarrow \langle \Phi \cup \{+\ell\}, \mathit{Id}, \mathtt{blame}\ +\ell\rangle$$

Contract soundness is useful when reasoning about functions and contracts. In the following program the codomain contract will never be assigned positive blame—despite the context providing an argument that does not satisfy $\mathtt{I} \to \mathtt{I}$—because the domain contract guards the codomain contract. Theorem 5.3 tells us that $(\lambda y.\mathtt{true})@^{-\ell[\mathsf{dom}_0/\mathsf{nil}]}\mathtt{I} \to \mathtt{I} \in [\![\mathtt{I} \to \mathtt{I}]\!]^+$, therefore in the final reduction the outer contract annotated with node $+\ell[\mathsf{cod}_0/\mathsf{nil}]$ will never be assigned positive blame.

$$\begin{aligned}
& \langle \cdot, K \circ \square\, \lambda y.\mathtt{true}, (\lambda x.x)@^{+\ell}(\mathtt{I} \to \mathtt{I}) \to (\mathtt{I} \to \mathtt{I})\rangle \\
\longrightarrow^* \quad & \langle [+\ell \mapsto 1], K, ((\lambda x.x)\,((\lambda y.\mathtt{true})@^{-\ell[\mathsf{dom}_0/\mathsf{nil}]}\mathtt{I} \to \mathtt{I}))@^{+\ell[\mathsf{cod}_0/\mathsf{nil}]}\mathtt{I} \to \mathtt{I}\rangle \\
\longrightarrow^* \quad & \langle [+\ell \mapsto 1], K, ((\lambda y.\mathtt{true})@^{-\ell[\mathsf{dom}_0/\mathsf{nil}]}\mathtt{I} \to \mathtt{I})@^{+\ell[\mathsf{cod}_0/\mathsf{nil}]}\mathtt{I} \to \mathtt{I}\rangle
\end{aligned}$$

As was the case with the program $(42@^{+\ell}\mathsf{B})@^{+\ell'}\mathsf{B}$, any devious behaviour of value will be exposed by the guarding contract before the outer contract is violated.

Contract soundness mandates that the annotating blame node is a root. If we were to guard 42 using a contract annotated with a blame node from a union branch then we could not guarantee that the outer contract is not evaluated:

$$\langle \Phi, K, (42@^{+\ell \bullet \mathsf{left}^+_\cup}\mathsf{B})@^{+\ell'}\mathsf{B}\rangle \longrightarrow^* \langle \Phi \cup \{+\ell \bullet \mathsf{left}^+_\cup\}, K, 42@^{+\ell'}\mathsf{B}\rangle \quad \text{if } +\ell \bullet \mathsf{right}^+_\cup \notin \Phi$$

The validity of contract soundness depends on two conditions. First, the guarding contract must be a top-level contract annotated with a blame label (or root node). Second, top-level violations must

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

Fig. 8. Sound Monitoring Properties

terminate the program with an error to prevent an outside observer interacting with an ill behaved term or continuation. These conditions are not compositional. We cannot use contract soundness to reason about the program $(\lambda x.x)@^p(\mathtt{I} \to \mathtt{I}) \to (\mathtt{I} \to \mathtt{I})$ when the contract is annotated with an arbitrary blame node p. The semantics of blame for wrapping the identity function should be independent of whether the contract is at the top-level, or nested within an intersection or union.

We introduce the additional property *witness soundness* (Theorem 5.4) that allows us to reason about such programs.

Theorem 5.4 (Witness Soundness).

- $M@^{-p \gg \mathrm{dom}_n} A \in [\![A]\!]^+_{p \gg \mathrm{cod}_n}$
- $K \circ \Box\ @^{p \gg \mathrm{cod}_n} B \in [\![B]\!]^-_{-p \gg \mathrm{dom}_n}$

Theorem 5.4 states that a value guarded with a contract of type A from a context (or function argument) always positively satisfies type A from the perspective of a subject (or function body). The dual property applies for contexts and negative satisfaction, arising in cases featuring higher-order functions types. This property encodes the implication naturally associated with function types and their logical interpretation. Witness soundness tells us that wrapping $\lambda x.x$ with any node p never assigns positive blame to the codomain, observing that $(\lambda y.\mathtt{true})@^{-p \gg \mathrm{dom}_0}\mathtt{I} \to \mathtt{I} \in [\![\mathtt{I} \to \mathtt{I}]\!]^+_{p \gg \mathrm{cod}_0}$.

$$
\begin{array}{lll}
 & \langle \cdot, K \circ \Box\, \lambda y.\mathtt{true}, (\lambda x.x)@^p(\mathtt{I} \to \mathtt{I}) \to (\mathtt{I} \to \mathtt{I}) \rangle \\
\longrightarrow^* & \langle [p \mapsto 1], K, ((\lambda x.x)\,((\lambda y.\mathtt{true})@^{-p \gg \mathrm{dom}_0}\mathtt{I} \to \mathtt{I}))@^{p \gg \mathrm{cod}_0}\mathtt{I} \to \mathtt{I} \rangle \\
\longrightarrow^* & \langle [p \mapsto 1], K, ((\lambda y.\mathtt{true})@^{-p \gg \mathrm{dom}_0}\mathtt{I} \to \mathtt{I})@^{p \gg \mathrm{cod}_0}\mathtt{I} \to \mathtt{I} \rangle
\end{array}
$$

Contract soundness is important for verifying that contract monitoring is implemented correctly, but it is not sufficient. A language may satisfy both soundness properties by making any contract check immediately diverge—clearly this is not desirable. We supplement our definition of satisfaction and soundness with a series of *sound monitoring* properties.

5.3 Sound Monitoring

Figure 8 presents the properties for monitoring contract types we expect any system to satisfy. Each type has a positive satisfaction rule for values and a negative satisfaction rule for continuations.

$$\begin{array}{llll} \text{Terms} & M, N & ::= & \cdots \mid V^{\square} \\ \text{Values} & V, W & ::= & \cdots \mid V^{\square} \end{array}$$

$$\frac{\text{for some } \Phi, \Delta, V \quad \langle \Phi, \Delta, K, V^{\square} \rangle \longrightarrow^{*} \langle \Phi', \Delta', K' \circ V^{\square}\, \square, N \rangle}{K \longrightarrow^{*}_{\square} K' \circ \square\, N}$$

Fig. 9. Context Reduction (Extends Syntax and Operational Semantics)

A value V positively satisfies base type ι if value V conforms to type ι. A continuation K negatively satisfies base type ι unconditionally. At an operational level a base type contract never negates its blame node so cannot introduce negative blame. At a semantic level a base type has no contextual requirements to be violated.

A value V positively satisfies the any type unconditionally, and a continuation K negatively satisfies the any type type unconditionally.

A value V positively satisfies function type $A \to B$ for any witness p if for all arguments N in $[\![A]\!]^{+}_{p \gg \mathrm{cod}_n}$, then application $V\ N$ is in $[\![B]\!]^{+}_{p \gg \mathrm{cod}_n}$, and for all continuations K in $[\![B]\!]^{-}_{-p \gg \mathrm{dom}_n}$, then the composed continuation $K \circ V\ \square$ is in $[\![A]\!]^{-}_{-p \gg \mathrm{dom}_n}$. Informally, if a value satisfies a function type then application to satisfying arguments yields satisfying results, and the continuation that applies the function value should respect the argument type when composed with a continuation that respects the result type.

Our property for function types is similar to the definition of satisfaction for function types given by Keil and Thiemann [2015a], except we use witness satisfaction rather than general satisfaction. The property requires us to show that $V@^{p}A \to B$ never blames p in any context, so arguments to V cannot be assumed to satisfy A. To use the hypothesis that assumes N satisfies A we must show that wrapping any argument with the domain contract produces a satisfying term—this is contract soundness. If we required $N \in [\![A]\!]^{+}$ then we would only be able to prove the property when p is a blame label as per contract soundness (Theorem 5.3). Instead, we relax the notion of satisfaction to witness satisfaction, taking the witnesses to be the domain and codomain blame nodes. If an argument N is satisfying according to the function result node $p \gg \mathrm{cod}_n$ then the function is obligated to return a satisfying result for the same node; the analogous case applies for continuations and negative satisfaction. Using witness satisfaction allows us to use witness soundness (Theorem 5.4) to show that any argument to V is satisfying because the argument is wrapped as $N@^{-p \gg \mathrm{dom}_n}A$. The use of witness satisfaction is justified because the witnesses are only those that arise naturally in evaluation, and the witness for V ranges over all nodes p, making it equivalent to general satisfaction. If we use general satisfaction for N and K then we are unable to reason about function types within union and intersection, which is unsatisfactory. When we restrict contracts to simple types then we may use general satisfaction provided that blame causes program termination.

A continuation K negatively satisfies function type $A \to B$ if for every argument continuation $K' \circ \square\, N$ that K reduces to then K' is in $[\![B]\!]^{-}$ and N is in $[\![A]\!]^{+}$. Informally, if a continuation satisfies a function type then the continuation only applies the function to satisfying arguments in continuations that satisfy the result type. This property uses *context reduction*, denoted $\longrightarrow^{*}_{\square}$. The definition is given in Figure 9 and extends the syntax and operational semantics of λ_{IU}. The value $V^{\square}$ denotes distinguished value V: the value that continuation K was initially applied to, or the "hole". The behaviour of $V^{\square}$ is identical to V in the operational semantics. Context reduction is used in the satisfaction property to state that for any value V that we apply the continuation to, if the configuration evaluates to an application of V, then that application continuation must satisfy the function type.

$$\text{Branch Types} \quad \circ \quad ::= \quad \cdots \mid \sqcap \qquad \langle K, V@^{p} A \sqcap B\rangle \longrightarrow \langle K, (V@^{p\bullet \mathsf{left}^{+}_{\sqcap}} A)@^{p\bullet \mathsf{right}^{+}_{\sqcap}} B\rangle$$

$$\begin{aligned} \mathit{resolve}(\, p \bullet d^{+}_{\sqcap}[P], \Phi\,) &= \mathit{assign}(\, \mathit{parent}(\, p \bullet d^{+}_{\sqcap}[P]\,), \Phi\,) \\ \mathit{resolve}(\, p \bullet d^{-}_{\sqcap}[P], \Phi\,) &= \mathit{assign}(\, \mathit{parent}(\, p \bullet d^{-}_{\sqcap}[P]\,), \Phi\,) \end{aligned}$$

$$\begin{aligned} V \in [\![A \sqcap B]\!]^{+} \quad &\text{if} \quad V \in [\![A]\!]^{+} \wedge V \in [\![B]\!]^{+} \\ K \in [\![A \sqcap B]\!]^{-} \quad &\text{if} \quad K \in [\![A]\!]^{-} \wedge K \in [\![B]\!]^{-} \end{aligned}$$

Fig. 10. Extending λ_{IU} to support *and* ($\sqcap$)

A value V positively satisfies intersection type $A \cap B$ if V positively satisfies A *and* V positively satisfies B. A continuation K negatively satisfies intersection type $A \cap B$ *if* K negatively satisfies A *or* K negatively satisfies B.

A value V positively satisfies union type $A \cup B$ if V positively satisfies A *or* V positively satisfies B. A continuation K negatively satisfies union type $A \cup B$ if K negatively satisfies A *and* K negatively satisfies B.

Proposition 5.5. *λ_{IU} obeys all monitoring properties in Figure 8.*

5.4 On Function Contracts

We follow existing work where contracts for function types only monitor application, they do not check that the value is a λ-abstraction. Keil and Thiemann [2015a] propose encoding a traditional function contract $\mathtt{I} \mapsto \mathtt{B}$ using an intersection contract $(\mapsto) \cap \mathtt{I} \to \mathtt{B}$, where $(\mapsto)$ denotes a base type that any abstraction conforms to. Following our satisfaction properties (and theirs) then *all* continuations would satisfy $\mathtt{I} \mapsto \mathtt{B}$! Base types are always negatively satisfied and intersection only requires negative satisfaction of one type. Clearly, intersection is not the right tool for the job.

Intersection should not be used to combine types with disjoint eliminators because there is no context where both branches *could* be assigned negative blame. Instead, we propose using *and* ($\sqcap$). The $\sqcap$ combinator acts like positive intersection and negative union (an intersection contract where both positive and negative blame are covariant). We extend our work to support $\sqcap$ in Figure 10. The operational semantics are extended with only a single rule that splits $\sqcap$. Blame resolution and contract satisfaction are extended with the rules for positive intersection and negative union.

The function encoding with $\sqcap$ still allows us to use intersection to build overloaded functions such as $((\mapsto) \sqcap \mathtt{I} \to \mathtt{B}) \cap ((\mapsto) \sqcap \mathtt{B} \to \mathtt{I})$ because each branch of the intersection is active in the function type eliminator (application).

6 COMPARISON

In this section we compare our work with existing approaches to contract semantics. First, the work by Keil and Thiemann [2015a] on intersection and union contracts. Second, the work by Dimoulas and Felleisen [2011], Dimoulas et al. [2011], and Dimoulas et al. [2012] on contract semantics and correctness for a contract calculus CPCF.

6.1 Intersection and union contracts by Keil and Thiemann [2015a]

Operational Semantics. The operational semantics presented by Keil and Thiemann [2015a] is the first to implement higher-order intersection and union contracts with blame. Our work shows that intersection and union contracts can be implemented uniformly. The salient consequence of

our approach is a simplified treatment of nested intersection and union. Keil and Thiemann [2015a] rely on the distributive law of intersection over union to monitor a contract of type $(A \cup B) \cap C$, leading to the reduction:

$$V@^{p}(A \cup B) \cap C \longrightarrow (V@^{p \bullet \mathsf{left}^{+}_{\cup}} A \cap C)@^{p \bullet \mathsf{right}^{+}_{\cup}} B \cap C$$

Our approach splits the intersection contract without duplicating contract C.

$$V@^{p}(A \cup B) \cap C \longrightarrow (V@^{p \bullet \mathsf{left}^{+}_{\cap}} A \cup B)@^{p \bullet \mathsf{right}^{+}_{\cap}} C$$

The system by Keil and Thiemann [2015a] supports user-defined contracts while we restrict the language of contracts to a fixed set of types. Combining intersection and union with user-defined contracts poses an additional challenge. A user-defined contract that applies its value, such as C defined as $\lambda f.(f\ \texttt{true}) = \texttt{false}$, will violate the commuting property of intersection and union under a naive implementation. The contract $C \cup (\texttt{I} \to \texttt{I})$ will apply contract C then wrap the resulting value with function contract $\texttt{I} \to \texttt{I}$; the contract $(\texttt{I} \to \texttt{I}) \cup C$ will wrap the value with function contract $\texttt{I} \to \texttt{I}$ then apply contract C to the wrapped value, triggering a negative blame violation on the domain of contract $\texttt{I} \to \texttt{I}$. The behaviour of contracts in separate branches should be independent however a naive implementation can allow them to interfere. Keil and Thiemann [2015a] present a solution that drops contracts appearing in certain illegal contexts. In the example, the wrapper for function contract $\texttt{I} \to \texttt{I}$ will be dropped in the body of contract C, preventing the negative blame violation. We expect that their solution can be adapted to our operational semantics.

Performance. The formal development by Keil and Thiemann [2015a] is used as a basis for the contract implementation *TreatJS* [Keil and Thiemann 2015b]. The latter provides a performance evaluation that analyses the impact of contracts on execution speed. We do not present a performance evaluation based on our work, however we can highlight the key differences between our work and that of Keil and Thiemann [2015a] that may impact performance.

The blame state we use to record contract violations is proportional in size to the constraint set Keil and Thiemann [2015a] maintain, with both data-structures representing the shapes of types used in contracts. To facilitate uniform monitoring our system requires a *context tracker* to record function application, whereas Keil and Thiemann [2015a] do not. The context tracker is proportional in size to the blame state. The primary reward for our approach is that we do not duplicate contracts when monitoring union within intersection because our implementation does not use the distributive law of intersection over union. Both the system of Keil and Thiemann [2015a] and our system do not implement recursive contracts. A naive implementation could cause the blame state to grow without bound. Combining recursive contracts with contracts that require blame state is an interesting technical challenge still to be solved.

Contract Semantics. In addition to an operational semantics, Keil and Thiemann [2015a] also present a "denotational" style contract semantics by defining sets of terms and contexts that satisfy a type using a series of coinductive definitions. The definitions are justified as denotational as they make no mention of contracts and monitoring behaviour, whereas our definition of satisfaction is defined purely in terms of monitoring and blame. They give blame soundness results showing that satisfying terms (contexts) never elicit positive (negative) blame, and their operational semantics satisfy the monitoring properties in Figure 8 as a direct consequence of their definition of contract satisfaction. Keil and Thiemann [2015a] show that their system satisfies *contract soundness* (Theorem 5.3), but they do not present a result equivalent to our notion of *witness soundness* (Theorem 5.4). Stating an equivalent property is difficult because their definition of contract satisfaction ignores blame, while witness soundness is fundamentally about contract satisfaction and blame.

6.2 CPCF

Correctness Criteria. Dimoulas and Felleisen [2011] present CPCF, a call-by-value variant of PCF with higher-order contracts including dependent function contracts, but without intersection and union. CPCF has subsequently been used to refine criteria for blame and monitoring correctness.

Dimoulas et al. [2011] introduce *blame correctness* and compare existing contract monitoring strategies *lax* and *picky*. The *lax* strategy is blame correct but fails to catch certain errors; the *picky* strategy catches more errors but is not blame correct. They introduce a third strategy, *indy*, that discovers the same number of errors as *picky* while retaining blame correctness.

Dimoulas et al. [2012] observe that blame correctness is not enough as it does not distinguish the monitoring strategies *lax* and *indy*. They develop the generalised criterion *complete monitoring*. A contract system is a complete monitor if the embedding of wrapped terms into existing programs is unable to induce stuck states, instead either terminating, diverging, or raising blame. Complete monitoring ensures that all values crossing module boundaries are monitored and they conclude that only the *indy* strategy satisfies complete monitoring. We conjecture that our calculus is a complete monitor under the condition that values embedded with function contracts are also paired with first-order function checks, avoiding stuck states that could arise from attempting to apply a number or boolean. This is because our definition of function contracts only monitor the application of a value, but do not guarantee that the value is a λ-abstraction. The complete monitor property is satisfied primarily because we do not support user-defined or dependent function contracts. To combine intersection and union with dependent function contracts and maintain complete monitoring is an interesting future challenge.

Contract Semantics. We are not the first to define contract satisfaction using monitoring behaviour; Dimoulas and Felleisen [2011] define contract satisfaction using observational equivalence. A value satisfies a contract if monitoring the value using the contract is observationally equivalent to monitoring the value using only the negative (or client) obligations of the contract. A context satisfies a contract if monitoring the context using the contract is observationally equivalent to monitoring the context using only the positive (or server) obligations of the contract. Dimoulas and Felleisen [2011] do not present a series of sound monitoring properties, though CPCF does satisfy the properties in Figure 8 for simple types. If we restrict contracts to simple types then our definition of contract satisfaction provides a similar equivalence.

Definition 6.1 (Positive and Negative Obligations).

$$(A \to B)^+ = A^- \to B^+ \qquad \iota^+ = \iota \qquad \mathsf{any}^+ = \mathsf{any}$$
$$(A \to B)^- = A^+ \to B^- \qquad \iota^- = \mathsf{any} \qquad \mathsf{any}^- = \mathsf{any}$$

Define operations A^+ and B^- on types that specify the positive (server) and negative (client) obligations of a type. Obligations for function types are contravariant in the domain and covariant in the codomain. Positive obligations for base types are the identity. Negative obligations for base types are any as base types have no contextual requirements. Obligations for any are the identity.

When V positively satisfies A then monitoring V with A and A^- are bisimilar. When K negatively satisfies B then guarding K with B and B^+ are bisimilar.

Theorem 6.2 (Satisfaction and Trust). *For bisimulation $\approx$ then:*

- $V \in [\![A]\!]^+$ *iff* $\langle K, V@^p A\rangle \approx \langle K, V@^p A^-\rangle$ *for all* K *and fresh* p.
- $K \in [\![B]\!]^-$ *iff* $\langle K, V@^p B\rangle \approx \langle K, V@^p B^+\rangle$ *for all* V *and fresh* p.

Our approach to contract satisfaction and the approach of Dimoulas and Felleisen [2011] coincide for simple types, but consider when the types are extended to include intersection and union.

The value $\lambda x.x$ should satisfy the intersection type $(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B})$. Following the obligation-oriented semantics of Dimoulas and Felleisen [2011] suggests that the client obligations of the intersection type are the domain contracts in each branch. Erasing the client obligations of the contract in any satisfying context should not add blame to the value. For example:

$$\langle Id \circ \square\, 42, \lambda x.x@^{+\ell}(\mathtt{I} \to \mathtt{I}) \cap (\mathtt{B} \to \mathtt{B})\rangle \tag{5a}$$

$$\langle Id \circ \square\, 42, \lambda x.x@^{+\ell}(\mathtt{any} \to \mathtt{I}) \cap (\mathtt{any} \to \mathtt{B})\rangle \tag{5b}$$

If Program (5a) does not assign blame to $+\ell$ then Program (5b) with client obligations erased should also not blame $+\ell$. However, observe that no terminating function may satisfy the type $(\mathtt{any} \to \mathtt{I}) \cap (\mathtt{any} \to \mathtt{B})$ as no function can return a value that satisfies both $\mathtt{I}$ and $\mathtt{B}$: erasing client obligations will cause $+\ell$ to be assigned blame! There is no immediate solution to the problem of combining obligation-oriented approaches to contract satisfaction with intersection and union.

7 RELATED WORK

Contracts and Blame. Meyer [1988] proposed the use of software contracts for monitoring function pre and post conditions at run-time, later implemented in Eiffel [Meyer 1992]. The description of contracts alluded to what we now refer to as positive and negative obligations (and consequently blame): preconditions bind the client and post-conditions bind the class.

Findler and Felleisen [2002] gave us higher-order function contracts along with positive and negative blame assignment. Their work paved the way for gradual typing and combining typed and untyped code [Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006], with an extensive range of research involving contracts, gradual typing, and blame soon following. Contracts have been extended to include polymorphism [Guha et al. 2007], and affine types [Tov and Pucella 2010]. Gradual typing has been extended to include references [Siek et al. 2015b], polymorphism [Ahmed et al. 2017; Igarashi et al. 2017a], and session types [Igarashi et al. 2017b]. Wadler and Findler [2009] introduce *blame calculus*, adding blame to gradually typed languages with explicit casts. Siek et al. [2015a] refined the definition of gradually typed languages with the *gradual guarantee*.

Wadler [2015] compares languages that use contracts and those that use casts. Particular focus is given to blame tracking for function types: *"To complement or not to complement?"*. They argue for the former. We believe the presentation of our system benefits significantly by using a single blame identifier with complement instead of using two and swapping them. Positive and negative blame identifiers have a pleasing symmetry with positive an negative satisfaction for types.

Gradual Typing with Intersection and Union. Castagna and Lanvin [2017] extend the gradually typed lambda calculus with intersection and union types. Their system uses a set-theoretic interpretation of intersection and union, and employs abstract interpretation in the style of Garcia et al. [2016] to give a semantics to gradual types. Dynamic checks are added through type-directed cast insertion, like most sound gradual type systems. They show that clever use of intersection types that combine the gradual type (?) can be used to reduce the number of type casts a user is required to write by hand. The calculus does not consider blame (using cast errors instead) and their choice of operational semantics prevents the statement of a useful blame theorem.

Toro and Tanter [2017] develop the idea of *gradual union types*, combining the advantages of tagged and untagged unions. A gradual union permits the flexibility of dynamic typing without being totally permissive: a gradual union type will statically reject some programs while the gradual type will not. Similarly to Castagna and Lanvin [2017], they use abstract interpretation to derive the semantics of their language, which also omits blame.

Siek and Tobin-Hochstadt [2016] extend the gradually typed lambda calculus with untagged union and equi-recursive types, allowing them handle common idioms for encoding data structures

in dynamically typed languages. Their union types only contain unique type constructors therefore unions of function types are not permitted. This restriction greatly simplifies blame tracking as it means union checking only requires first-order checks, without any blame state.

Contract Semantics. Blume and McAllester [2006] give a sound and complete model for contracts defined in the style of Findler and Felleisen [2002]. The semantics of a contract are defined to be a set of values, similarly to Keil and Thiemann [2015a], and their definition resembles our monitoring properties. Soundness is defined using a *central lemma* that exhibits a duality between positive and negative, comparable to other work, however their semantics does not distinguish satisfying contexts. Consequently, their soundness result must define a set of *safe* programs where possible blame errors can be replaced with divergence, without changing behaviour. Findler et al. [2004] and Findler and Blume [2006] study contracts as pairs of projections. They find that using projections to model contracts yields a more efficient implementation. Extending a projection model to support intersection and union poses an interesting challenge as the projection functions would have to be impure functions due to blame state. The model developed by Findler and Blume [2006] reveals that there are two kinds of any contract: one that is the most permissive, and one that has no obligations. Our implementation of any satisfies the latter interpretation.

Xu et al. [2009] develop a static verification framework for Haskell using higher-order contracts and symbolic execution. They give a declarative definition of satisfaction that shares similarities with ours, however they do not give a concrete definition to satisfaction and do not consider negative satisfaction. They highlight the *telescopic property*: when composing two wrappers of the same type, the negative label of the first wrapper and the positive label of the second wrapper are redundant. A directly equivalent property cannot be stated in our framework as we complement a single label rather than swap a pair of labels, however the property can be captured in spirit. For any term $M@^{+\ell}A@^{+\ell'}A$, then the blame nodes $-\ell$ and $+\ell'$ (if they arise during reduction through wrapping) can never be blamed—this follows immediately from contract soundness.

Swords et al. [2018] study a variety of contract enforcement strategies including eager and lazy, synchronous and parallel. They present a unified framework capable of describing the aforementioned strategies by observing that contracts are patterns of communication, with monitored programs communicating with contract monitors via channels. Their framework is highly compositional as it allows contracts to be implemented using multiple strategies.

8 CONCLUSION

Contracts are a lightweight approach for using types to describe and enforce program invariants dynamically. Intersection and union describe common idioms from dynamically typed programming, making them desirable contract operators. We extend the untyped lambda calculus with contracts for higher-order intersection and union types, giving uniform treatment to both. Previous work relies on monitoring that has multiple specialised rules for intersection; we give a single rule that immediately decomposes any intersection. Previous work relies on monitoring that extracts unions from within intersections; we give a single rule that immediately decomposes any union.

We give a new approach to describing and verifying contract semantics in the untyped lambda calculus. We adopt existing approaches that describe values as positively satisfying types and continuations as negatively satisfying types, however we describe satisfaction in terms of blame behaviour. Additionally, we also define a new type of satisfaction: witness satisfaction. We show that our monitoring semantics are correct by describing a series of monitoring properties any system should satisfy. Our use of witness satisfaction allows us to verify the monitoring semantics for all contracts that occur at run-time, not only top-level contracts.

REFERENCES

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free: Parametricity, With and Without Types. In *ACM International Conference on Functional Programming (ICFP)*.

Franco Barbanera and Mariangiola Dezani-Ciancaglini. 1991. Intersection and Union Types. In *Theoretical Aspects of Computer Software*, Takayasu Ito and Albert R. Meyer (Eds.). Springer Berlin Heidelberg.

Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/978-3-662-44202-9_11

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

Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. In *ACM International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/3110285

Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and Precise Type Checking for JavaScript. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/3133872

Mario Coppo and Mariangiola Dezani-Ciancaglini. 1978. A new type assignment for λ-terms. 19 (1978), 139–156.

Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. In *ACM International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/351240.351259

Christos Dimoulas and Matthias Felleisen. 2011. On contract satisfaction in a higher-order world. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 5 (2011), 16.

Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. 2011. Correct Blame for Contracts: No More Scapegoating. In *ACM Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1926385.1926410

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

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*.

Robert Bruce Findler and Matthias Blume. 2006. Contracts as Pairs of Projections. In *International Symposium on Functional and Logic Programming (FLOPS)*, Masami Hagiya and Philip Wadler (Eds.).

Robert Bruce Findler, Matthias Blume, and Matthias Felleisen. 2004. An investigation of contracts as projections. *University of Chicago Technical Report, TR-2004-02* (2004).

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-order Functions. In *ACM International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/583852.581484

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *ACM Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2837614.2837670

Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. 2007. Relationally-parametric Polymorphic Contracts. In *Dynamic Languages Symposium (DLS)*. https://doi.org/10.1145/1297081.1297089

Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. 2017b. Gradual Session Types. In *ACM International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/3110282

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017a. On Polymorphic Gradual Typing. In *ACM International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/3110284

Matthias Keil and Peter Thiemann. 2015a. Blame assignment for higher-order contracts with intersection and union. In *ACM International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2858949.2784737

Matthias Keil and Peter Thiemann. 2015b. TreatJS: Higher-Order Contracts for JavaScripts. In *European Conference on Object-Oriented Programming (ECOOP) (Leibniz International Proceedings in Informatics (LIPIcs))*. 28–51. https://doi.org/10.4230/LIPIcs.ECOOP.2015.28

Bertrand Meyer. 1988. *Object-Oriented Software Construction*. Prentice-Hall, Inc.

Bertrand Meyer. 1992. *Eiffel: The Language*. Prentice-Hall, Inc.

Benjamin C. Pierce. 1993. Intersection Types and Bounded Polymorphism. In *Typed Lambda Calculi and Applications*. Springer Berlin Heidelberg.

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

Jeremy G. Siek and Sam Tobin-Hochstadt. 2016. *The Recursive Union of Some Gradual Types*. Springer International Publishing. https://doi.org/10.1007/978-3-319-30936-1_21

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015a. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015b. Monotonic References for Efficient Gradual Typing. In *European Symposium on Programming (ESOP)*.

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

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *Dynamic Languages Symposium (DLS)*. https://doi.org/10.1145/1176617.1176755

Matías Toro and Éric Tanter. 2017. A Gradual Interpretation of Union Types. In *Static Analysis*.

Jesse A. Tov and Riccardo Pucella. 2010. Stateful Contracts for Affine Types. In *European Symposium on Programming (ESOP)*.

Philip Wadler. 2015. A Complement to Blame. In *SNAPL*.

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *European Symposium on Programming (ESOP)*. 1–16.

Dana N. Xu, Simon Peyton Jones, and Koen Claessen. 2009. Static Contract Checking for Haskell. In *ACM Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1480881.1480889