Blame Assignment for Higher-Order Contracts with Intersection and Union

Matthias Keil Peter Thiemann
University of Freiburg, Germany
{keilr,thiemann}@informatik.uni-freiburg.de

Abstract

We present an untyped calculus of blame assignment for a higher-order contract system with two new operators: intersection and union. The specification of these operators is based on the corresponding type theoretic constructions. This connection makes intersection and union contracts their inevitable dynamic counterparts with a range of desirable properties and makes them suitable for subsequent integration in a gradual type system.

A denotational specification provides the semantics of a contract in terms of two sets: a set of terms satisfying the contract and a set of contexts respecting the contract. This kind of specification for contracts is novel and interesting in its own right.

A nondeterministic operational semantics serves as the specification for contract monitoring and for proving its correctness. It is complemented by a deterministic semantics that is closer to an implementation and that is connected to the nondeterministic semantics by simulation.

The calculus is the formal basis of *TreatJS*, a language embedded, higher-order contract system implemented for JavaScript.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics; D.4.2 [*Software/Program Verification*]: Programming by contract

Keywords Blame, Higher-Order Contracts, Intersection, Union

1. Introduction

Contracts and contract monitoring [23] are established tools for enforcing certain guarantees at run time. While there are uses and implementations across the whole spectrum of programming languages, contracts are particularly popular for dynamically typed languages that offer few guarantees (beyond memory safety) at run time. In particular, the adoption of contracts in Racket [13] has gained widespread interest.

Starting from simple assertions, contract facilities in programming languages have been extended in line with constructions studied in type theory. This analogy enables the transfer of specifications and desired behavior from statically checked type systems to dynamically checked contract systems. It also facilitates the construction of gradual systems [26, 27] that mediate between statically and dynamically typed components at run time. Previous implementations of contract systems support operators analogous to (dependent) function types [13], product types, sum types [18], as well as universal types [1] and recursive types [29].

Logical operations are another source of inspiration for contract operators. For example, Racket [14, Chapter 8] supports some form of conjunction, disjunction, and negation of contracts. However, they are a best-effort implementation: they come with an operational explanation and the operators cannot be freely combined. Theoretical investigations [7, 29] place similar restrictions on conjunction and disjunction.

We propose to complement the arsenal of contract operators with two further operators from type theory: intersection types and union types. Intersection and union types were conceived for purely theoretical concerns [2], before they were discovered for programming languages [24]. Intersection types may be used to describe overloaded functions [16] and multiple inheritance; union types are dual to intersection types and come up in connection with XML typing [20] as well as in soft type systems [32].

By starting from their type theoretic foundation, we analyze the metatheoretic properties of intersection and union types. This analysis is our basis for postulating requirements for their dynamic contract counterparts. We believe that this approach has a number of advantages when it comes to designing contract operators.

First, many programmers are acquainted with type systems and their operators. If we can provide contracts with matching semantics, then they can build on their type-intuitions when using the corresponding contracts.

Second, matching semantics is important when designing a gradual type system for a language like TypedRacket [28]. In such a system it is crucial that the static meaning of a type operator coincides with the dynamic meaning of its contract counterpart.

Third, when defining operators by derivation from a specification, they obey a range of useful properties by construction. For example, our derived intersection and union operators inherit symmetry, idempotence, and distribution laws with function contracts from the corresponding type constructions.

Contributions This work presents the theory of blame assignment underlying *TreatJS*, a language embedded, higher-order contract system for JavaScript.[1]

- We specify the semantics of a contract in a novel denotational style by a set of terms (subjects) satisfying the contract and a set of contexts respecting the contract.
- We extend higher-order contracts with unrestricted intersection and union contracts; they provide dynamic guarantees analogous to the static guarantees of intersection and union types.

[1] `http://proglang.informatik.uni-freiburg.de/treatjs/`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
© 2015 ACM. 978-1-4503-3669-7/15/08...$15.00
http://dx.doi.org/10.1145/2784731.2784737

- We give a nondeterministic specification of contract monitoring for the full system; the nondeterminism is not essential, but it simplifies the presentation and the proofs.
- We provide a deterministic implementation and establish a simulation relation with the specification.
- We prove contract soundness theorems that are novel in two aspects: they cover subjects and contexts; and they deal with intersection and union.

Overview Section 2 introduces higher-order contracts and our novel notion of context satisfaction and then moves on to motivate requirements for intersection and union contracts from their type-theoretic counterparts. Section 3 explains the denotational semantics of contracts based on an untyped, applied, call-by-value lambda calculus and establishes some fundamental properties of the semantics. Section 4 extends the lambda calculus with nondeterministic contract monitoring, explains its reduction semantics, and specifies the constraint-based computation of blame. Section 5 defines the deterministic monitoring semantics, explains the crucial notion of compatibility, and states the simulation theorem. Section 6 states and explains our contract and blame soundness theorems. Section 7 discusses related work and Section 8 concludes.

A technical report [21] extends this paper by an appendix with further examples and proofs of all theorems.

2. Motivation

We briefly recall the standard contracts and the notion of blame from the literature [13]. Contract satisfaction is usually defined from the point of view of the contract's subject; as a novelty, we introduce the dual concept of *context satisfaction*, which answers the question when a context respects a contract in its hole.

In anticipation of the formal framework defined in Section 3, we let M and N range over lambda terms, V over values, $\mathcal{L}$ over contexts, E over evaluation contexts, and C and D over contracts. We write $M@C$ for asserting contract C to M; at run time, $M@C$ monitors the execution of M and reports violations of C. We also use S and T informally to range over an unspecified language of types that includes the language of simple types.

2.1 Higher-Order Contracts and Context Satisfaction

A flat contract, $\mathit{flat}(M)$, where the expression M denotes a predicate, is satisfied by subject V if the application $M\ V$ does not evaluate to false[2]. A violation raises positive blame. On the other hand, every context respects a flat contract because the contract does not restrict it in any way. Thus, a flat contract never raises negative blame.

A function satisfies a function contract $C \to D$ whenever calling it with an argument that satisfies C implies that the result of the function satisfies D. If an argument satisfying C provokes a result that does not satisfy D, then the function contract raises *positive blame*: the function, the subject, does not satisfy the contract.

Calling the contracted function with an argument that does not satisfy C leads to *negative blame*. Thus, a contract also places an obligation on the context that it may or may not fulfill. We define that a *context respects the contract* $C \to D$ if it only provides arguments satisfying C (as a subject) and puts the result in a context respecting D. Such a context never causes negative blame.

As an example consider the function $\mathit{add} = \lambda x.\lambda y.x + y$ and contract $C = \mathit{Pos} \to (\mathit{Even} \to \mathit{Even})$ where $\mathit{Pos} = \mathit{flat}(\lambda x.x > 0)$ and $\mathit{Even} = \mathit{flat}(\lambda x.x \bmod 2 = 0)$. Applying $\mathit{add}@C$ to 0 yields negative blame: the context $\Box\, 0$ violates the obligation to only provide positive arguments. Applying $(\mathit{add}@C)\ 1$ to 1 also yields negative blame because it puts the outcome of $(\mathit{add}@C)\ 1$ in a context $(\Box\, 1)$ that does not respect the contract $\mathit{Even} \to \mathit{Even}$. Applying $(\mathit{add}@C)\ 1$ to 0 yields positive blame to indicate that add does not satisfy C.

[2] It is also satisfied if $M\ V$ does not terminate.

INTER-I

$$\frac{A \vdash V : S \qquad A \vdash V : T}{A \vdash V : S \cap T}$$

SUB-INTER-L $\quad S \cap T <: S$

SUB-INTER-R $\quad S \cap T <: T$

Figure 1. Intersection types

2.2 Intersection

If a value has both type S and T, then we can also assign it the intersection type $S \cap T$ [5]. Conversely, if a value has type $S \cap T$, then its context may choose to use it as a value of type S or as a value of type T. This intuition materializes directly in the typing and subtyping rules for intersection in Figure 1.[3]

Pierce [25] calls intersection types *the natural type-theoretic analogue of multiple inheritance*, where $S \cap T$ is the name of a class with the properties of both S and T. Intersection types also find uses in modeling finitary overloading as in the following typing for a $+$ operator that stands for addition and string concatenation.

$$+ : (\mathit{Num} \times \mathit{Num} \to \mathit{Num}) \cap (\mathit{Str} \times \mathit{Str} \to \mathit{Str}) \qquad (1)$$

The typing rules for intersection suggest the following requirements for an intersection contract.

IS0 (Idempotence) A value satisfies $C \cap C$ iff it satisfies C.

IS1 (Symmetry) A value satisfies $C \cap D$ iff it satisfies $D \cap C$.

IS2 (Introduction) A value satisfies the intersection contract $C \cap D$ **iff** it satisfies both contracts C **and** D.

For flat contracts, it is easy to check contract satisfaction and it is also straightforward to see that $\mathit{flat}(\lambda x.P) \cap \mathit{flat}(\lambda x.Q) = \mathit{flat}(\lambda x.P \wedge Q)$ is a definition that satisfies the requirements.

For higher-order contracts, monitoring shares the deficiencies of all contract validation methods that are based on testing: Monitoring cannot determine contract satisfaction in general, but it can detect contract failures. Hence, we switch our point of view from contract satisfaction to contract failure manifested in blame allocated by detected contract violations. Switching the point of view turns out to be a matter of negating the requirements.

Recall that positive (negative) blame indicates that a subject (context) does not satisfy (respect) a contract. This observation is key to rephrasing the requirements. We concentrate on the most relevant requirement **IS2**, the others can be treated analogously.

IS2B $\mathcal{L}[M@(C \cap D)]$ raises positive blame iff $\mathcal{L}[M@C]$ raises positive blame or $\mathcal{L}[M@D]$ raises positive blame.

To also capture negative blame in our requirements for an intersection contract, we first need to state when a context satisfies such a contract. The elimination rules SUB-INTER-L/R (via subsumption) are our guidelines. They indicate that the context may choose to consider a value of type $S \cap T$ as either an S or a T. It is, however, critical that this choice is delayed as much as possible (see example at the end of this subsection). The choice must happen in an *elimination context* $\mathcal{F}$, that is, an evaluation context E that directly applies an elimination form to its hole: $\mathcal{F} ::= E[\Box\, V] \mid \ldots$

IC2 An elimination context respects the intersection contract $C \cap D$ **iff** it respects contract C **or** contract D.

[3] The use of V in the introduction rule makes it sound for call-by-value [6].

$$\textsc{Union-E}\quad \frac{A \vdash M : S \cup T \qquad A, x : S \vdash N : R \qquad A, x : T \vdash N : R}{A \vdash \mathit{let}\, x = M\, \mathit{in}\, N : R}$$

$$\textsc{Sub-Union-L}\quad S <: S \cup T$$

$$\textsc{Sub-Union-R}\quad T <: S \cup T$$

Figure 2. Union types

To check this condition effectively, we need to rephrase in terms of contract failures: If there is a term such that the context provokes negative blame, then the context cannot respect the contract.

IC2B $\mathcal{F}[M@(C \cap D)]$ raises negative blame **iff** $\mathcal{F}[M@C]$ **and** $\mathcal{F}[M@D]$ both raise negative blame.

Let's evaluate this definition with the overloaded $+$ operator from (1) regarding the type as an intersection contract. If we apply $+$ with the intersection contract to a pair of numbers, then the $\mathit{Str} \times \mathit{Str} \to \mathit{Str}$ part of the contract raises negative blame, but the $\mathit{Num} \times \mathit{Num} \to \mathit{Num}$ part does not. Hence, the intersection must not raise negative blame, either. The same happens, mutatis mutandis, when applying to a pair of strings. Applying to a pair of a number and a string triggers negative blame in both function contracts. Thus, the intersection must also raise negative blame.

As the intended semantics of $+$ satisfies the intersection contract, no use of it would ever give rise to positive blame. However, if we apply a function f with the same intersection contract to a pair of numbers (strings) but the result fails to satisfy Num (Str), then blame is assigned to the subject f.

Generally, the subject of an intersection contract $C \cap D$ must fulfill both contracts C and D. If $C = C_1 \to C_2$ and $D = D_1 \to D_2$ are both function contracts, then any argument has to fulfill $C_1 \cup D_1$. If the argument contracts overlap, then applying the function to an element in their intersection must yield a result that satisfies both, C_2 and D_2. As an example, consider the contract

$$(\mathit{Num} \times \mathit{Num} \to \mathit{Num}) \cap (\mathit{Pos} \times \mathit{Pos} \to \mathit{Pos}) \qquad (2)$$

which describes a function with domain $\mathit{Num} \times \mathit{Num}$ that must map positive arguments to positive results.

Our final example illustrates the need for elimination contexts.

Choose the context $\mathcal{L}$ as: $\quad \mathit{let}\, f = \square\, \mathit{in}\, f\, 42; f\, \texttt{"foo"}$

Clearly, $\mathcal{L}$ respects $(\mathit{Num} \to \mathit{Num}) \cap (\mathit{Str} \to \mathit{Str})$ because it applies f to a number and to a string, but it respects neither $\mathit{Num} \to \mathit{Num}$ nor $\mathit{Str} \to \mathit{Str}$. This example further indicates that the checking of an intersection context attached to a value must happen at the elimination of this value.

2.3 Union

Union types [3] arise naturally in a number of ways: as the dual of intersection types, from logical and semantical considerations, and as generalizations of sum and variant types. Again paraphrasing Pierce [24], union types are related to sum types in the same way as set-theoretic union is related to disjoint union. They are governed by the rules in Figure 2. There is no explicit introduction rule; instead a term of type S or T may be viewed as a term of type $S \cup T$ via subsumption. Pierce's elimination rule Union-E [24] conveys that a context that wants to use a value of union type $S \cup T$ must be prepared to deal with both S and T because the choice between them is taken internally by the value.[4]

[4] Among the elimination rules for union types in the literature we have chosen a simple one that is sound for call-by-value. More general rules exist [10], but they are not needed in this context.

Analogously to intersection types, the requirements for a union contract derive from the typing rules for union types.

US0 (Idempotence) A value satisfies $C \cup C$ iff it satisfies C.

US1 (Symmetry) A value satisfies $C \cup D$ iff it satisfies $D \cup C$.

US2 (Introduction) A value satisfies the union contract $C \cup D$ iff it satisfies contract C or contract D.

It is again easy to see that the union of flat contracts corresponds to the disjunction of their predicates:

$$\mathit{flat}(\lambda x.P) \cup \mathit{flat}(\lambda x.Q) = \mathit{flat}(\lambda x.P \vee Q)$$

For the higher-order case, we rephrase **US2** to blame reporting, again. This time, it is sufficient to restrict to evaluation contexts E.

US2B $E[M@(C \cup D)]$ raises positive blame **iff** $E[M@C]$ **and** $E[M@D]$ both raise positive blame.

The elimination rule Union-E guides the definition of context satisfaction.

UC2 A context respects the union contract $C \cup D$ **iff** it respects contract C **and** contract D.

The rephrasing to blame is by now routine.

UC2B $\mathcal{L}[M@(C \cup D)]$ raises negative blame **iff** $\mathcal{L}[M@C]$ raises negative blame **or** $\mathcal{L}[M@D]$ raises negative blame.

As an example consider the contract

$$(\mathit{Even} \to \mathit{Even}) \cup (\mathit{Pos} \to \mathit{Pos}) \qquad (3)$$

which is either satisfied by a function that always maps an even number to an even number (like $\lambda x. -x$) or by one that always maps a positive number to a positive number (like $\lambda x.x + 1$). It is *not* satisfied by a function that alternates between both return types. For example, the following function h does not satisfy (3).

$$h(x) = \mathit{if}\,(x = 6)\, \mathit{then}\, -6\, \mathit{else}\, 3$$

Because the context has to respect the union contract (3), any argument that does not satisfy $\mathit{Even} \cap \mathit{Pos}$ ought to raise negative blame. A positive even number is needed to elicit positive blame. By inspection, we see that 2 and 6 are representative arguments that exercise all possible behaviors of h. However, $h(2) = 3$ satisfies *Pos* (but fails *Even*) whereas $h(6) = -6$ satisfies *Even* (but fails *Pos*). In this example, no single call in isolation raises positive blame to unveil the insidious behavior of h: at least two tests (e.g., with 2 and 6) are needed elicit positive blame and *monitoring must remember the outcome of previous tests to assign blame properly*.

One might ask why **US2B** is restricted to evaluation contexts. As an example, we construct a dual situation as in the example that exhibited the problem for intersection:

$$\begin{array}{lcl} \mathcal{L} & = & \mathit{let}\, f = (\lambda x.\square)\, \mathit{in}\, (f\, \mathit{true}; f\, \mathit{false}) \\ M & = & \mathit{if}\, x\, \mathit{then}\, 1\, \mathit{else}\, \mathit{true} \end{array}$$

In this case, $\mathcal{L}[M@\mathit{Num}]$ raises positive blame and so does $\mathcal{L}[M@\mathit{Bool}]$. The interesting point is that $\mathcal{L}[M@(\mathit{Num} \cup \mathit{Bool})]$ does *not* raise positive blame, as each invocation of f creates a new union contract which can choose a suitable summand for each value that arises. This behavior conforms to **US2B** because $\mathcal{L}$ is not an evaluation context. If we wrap the choice into a function $h(x) = \mathit{if}\, x\, \mathit{then}\, 1\, \mathit{else}\, \mathit{true}$, then this function satisfies the contract $\mathit{Bool} \to (\mathit{Num} \cup \mathit{Bool})$, but not $(\mathit{Bool} \to \mathit{Num}) \cup (\mathit{Bool} \to \mathit{Bool})$ as explained in the *Even*/*Pos* example.

3. Semantics of Contract Satisfaction

This section defines λ_V^{Con}, an untyped call-by-value lambda calculus with contracts. It first introduces the base calculus and the syntax of

$$
\begin{array}{lcl}
L, M, N & ::= & K \mid x \mid \lambda x.M \mid M\,N \mid O(\vec{M}) \\
K & ::= & \mathit{false} \mid \mathit{true} \mid 0 \mid 1 \mid \ldots \\
C, D & ::= & \mathit{flat}(M) \mid C \to D \mid C \cap D \mid C \cup D \\
V, W & ::= & K \mid \lambda x.M \\
E & ::= & \Box \mid O(\vec{V}\,E\,\vec{M}) \mid E\,M \mid V\,E \\
\mathcal{L}, \mathcal{M}, \mathcal{N} & ::= & \mathcal{A} \mid \mathcal{M}\,N \mid M\,\mathcal{N} \mid O(\vec{M}\mathcal{L}\vec{N}) \\
\mathcal{V} & ::= & \lambda x.\mathcal{L} \\
\mathcal{A} & ::= & \Box \mid \mathcal{V}
\end{array}
$$

$$
\begin{array}{llll}
\text{BETA} & E[(\lambda x.M)\,V] & \longrightarrow E[M\{x := V\}] & \\
\text{OP} & E[O(\vec{V})] & \longrightarrow E[\delta_O(\vec{V})] & \vec{V} \in \mathit{dom}(\delta_O)
\end{array}
$$

Figure 3. Terms, contexts, and reductions of λ_V

$$
\begin{array}{lcll}
[\![\mathit{flat}(M)]\!]^+ & = & \{N \mid M\,N \not\longrightarrow^* \mathit{false}\} & (4) \\
[\![C \to D]\!]^+ & = & \{M \mid \forall N \in [\![C]\!]^+ . M\,N \in [\![D]\!]^+ & (5) \\
 & & \quad \wedge \forall \mathcal{N} \in [\![D]\!]^- . \mathcal{N}[M\,\Box] \in [\![C]\!]^-\} & \\
[\![C \cap D]\!]^+ & = & [\![C]\!]^+ \cap [\![D]\!]^+ & (6) \\
[\![C \cup D]\!]^+ & = & [\![C]\!]^+ \cup [\![D]\!]^+ & (7)
\end{array}
$$

Figure 4. Contract subject satisfaction for λ_V

P-IRRED
$$\frac{\mathcal{M} \not\longrightarrow \qquad \mathcal{M} \notin \{E, E[V\,\mathcal{A}], E[\Box\,N]\}}{\mathcal{M} \in [\![C]\!]^-}$$

P-REDUCE
$$\frac{\mathcal{N} \in [\![C]\!]^- \qquad \mathcal{M} \longrightarrow \mathcal{N}}{\mathcal{M} \in [\![C]\!]^-}$$

P-STUCK
$$\frac{V \neq \lambda x.M}{E[V\,\mathcal{A}] \in [\![C]\!]^-}$$

P-EXPAND
$$\frac{\begin{array}{c}\forall \mathcal{M}, V.\, \lambda x.M = \lambda x.\mathcal{M}[x] \Rightarrow \\ E[\mathcal{M}\{x := \mathcal{A}[V]\}[\mathcal{A}]] \in [\![C]\!]^-\end{array}}{E[(\lambda x.M)\,\mathcal{A}] \in [\![C]\!]^-}$$

P-FLAT
$$E \in [\![\mathit{flat}(M)]\!]^-$$

P-OP
$$E[O(\vec{V}\Box\vec{N})] \in [\![C \to D]\!]^-$$

P-APPLY
$$\frac{N \in [\![C]\!]^+ \qquad E \in [\![D]\!]^-}{E[\Box\,N] \in [\![C \to D]\!]^-}$$

P-UNION
$$\frac{E \in [\![C]\!]^- \qquad E \in [\![D]\!]^-}{E \in [\![C \cup D]\!]^-}$$

P-INTER-L
$$\frac{\mathcal{F} \in [\![C]\!]^-}{\mathcal{F} \in [\![C \cap D]\!]^-}$$

P-INTER-R
$$\frac{\mathcal{F} \in [\![D]\!]^-}{\mathcal{F} \in [\![C \cap D]\!]^-}$$

$$\mathcal{F} ::= E[O(\vec{V}\Box\vec{N})] \mid E[\Box\,N]$$

Figure 5. Contract context satisfaction (coinductive)

contracts, then proceeds to describe contracts and their semantics for the base calculus, and finally gives the semantics of contract assertion and blame propagation.

3.1 The Base Language λ_V

Figure 3 defines syntax and semantics of λ_V, an applied call-by-value lambda calculus, and the syntax of contracts. An expression M is either a first-order constant, a variable, a lambda abstraction, an application, or a primitive operation. Variables, ranged over by x and y, are drawn from a denumerable set. Constants K range over a set of base type values including booleans and numbers.

A contract C is either a flat contract $\mathit{flat}(M)$ defined by a predicate M, a function contract $C \to D$ with domain contract C and range contract D, an intersection between two contracts $C \cap D$, or a union $C \cup D$.

To define evaluation, V and W range over values and E over evaluation contexts, which are standard. The small-step reduction relation $\longrightarrow$ comprises beta-value reduction and built-in partial operations that transform a vector of values into a value. We write $\longrightarrow^*$ for its reflexive, transitive closure and $\not\longrightarrow^*$ for its complement. That is, $M \not\longrightarrow^* N$ if, for all L such that $M \longrightarrow^* L$, it holds that $L \neq N$. We also write $M \not\longrightarrow$ to indicate that there is no N such that $M \longrightarrow N$.

Contexts $\mathcal{L}$ are defined as usual as terms with a hole. We single out value contexts $\mathcal{V}$ that wrap a context in a lambda and answer contexts $\mathcal{A}$ that are value contexts or just a hole. We extend the reduction relation $\longrightarrow$ to contexts by considering the hole as a non-value term. If context reduction terminates, then it has either reached a context value, an evaluation context, or a stuck context.

3.2 Contract Satisfaction

Figures 4 and 5 define contract satisfaction for subjects and contexts, respectively, in terms of the semantics of λ_V. The set $[\![C]\!]^+$ is defined to be the set of *closed terms* (subjects) that *satisfy* the contract C. The set $[\![C]\!]^-$ is the set of *closed contexts* that *respect* C. The definitions are mutually inductive on the structure of C and the rule set in Figure 5 is coinductively defined. We connect this semantics to contract monitoring in Sections 4 and 6.

Equation (4) directly reflects the discussion of flat contracts in Section 2.1. Subject satisfaction for a function contract (5) also follows the previous discussion but with an extra twist. If the argument contract of a function M is itself a function contract, then M must put its argument V, say, in a context that satisfies C (i.e., it must not pass an argument that does not subject-satisfy C), but only under the assumption that the application $M\,V$ itself happens in a context satisfying D. To appreciate the context part of the definition consider that $\lambda x.x \in [\![C \to C]\!]^+$, in particular for $C = \mathit{flat}(\lambda x.x \neq 0) \to \mathit{flat}(\lambda x.x \neq 0)$. However, $\lambda x.x$ cannot guarantee C for its argument if its context does not guarantee C. The term $((\lambda x.x)(\lambda y.1/y))0$ demonstrates such a case.

Equation (6) for satisfaction of intersection corresponds to the requirement **IS2** and Equation (7) for union corresponds to **US2**.

The set of contexts that respect a contract C is defined by induction on the structure of C and then *coinductively* at each level by the rule set in Figure 5. It relies on context reduction.

Generally, a context that never exercises its hole respects all contracts. Rule P-IRRED covers all the cases where context reduction gets stuck before the hole gets involved in a reduction—the exempted cases are covered by other rules. Irreducible contexts include the empty context $\Box$ and contextual values $\mathcal{V}$.

Rule P-REDUCE closes respecting contexts under reduction. Its coinductive interpretation guarantees that contexts that diverge before the hole gets involved respect all contracts. P-STUCK covers the case where an application of a value V to an argument involving a hole cannot reduce because V is not a function.

Rule P-EXPAND treats the case where the hole is involved as part of the argument in a beta-reduction. Before explaining the general rule, it is easier to first consider two special cases of P-EXPAND where x occurs at most once in the body M of the function.

If x does not occur free in M, then $\mathcal{L}[(\lambda x.M)\,\Box] \in [\![C]\!]^-$ because the subject disappears on reduction and cannot be exercised further on. In this case, the premise of P-EXPAND is vacuously true because there is no context $\mathcal{M}$ such that $\lambda x.M = \lambda x.\mathcal{M}[x]$.

If x occurs exactly once in M, then the composed context $E[\mathcal{M}[\mathcal{A}]]$ must be satisfying. In this case, $\mathcal{M}$ does not contain free occurrences of x so that the substitution $\mathcal{M}\{x := \mathcal{A}[V]\} = \mathcal{M}$ has no effect in rule P-EXPAND.

Otherwise, the rule requires that each occurrence of x in $\lambda x.M$ that is bound by the lambda gives rise to a contract respecting context for all values V substituted for the remaining occurrences of x. As an example, consider checking whether the context $E = (\lambda x.N\,x\,x)\,\square$ respects contract $C \rightarrow D$. Thus, we want to ensure that $(\lambda x.N\,x\,x)\,[W \;@\; (C \rightarrow D)]$ does not raise negative blame. Now the latter term reduces to $N\,(W \;@\; (C \rightarrow D))\,(W \;@\; (C \rightarrow D))$ so to ensure that $E \in [\![C \rightarrow D]\!]^-$ it must be the case that both $N\,\square\,(W \;@\; (C \rightarrow D)) \in [\![C \rightarrow D]\!]^-$ and $N\,(W \;@\; (C \rightarrow D))\,\square \in [\![C \rightarrow D]\!]^-$. As $W \,@\, (C \rightarrow D)$ can be an arbitrary value that satisfies $C \rightarrow D$, which cannot be generated from the existing context E, the rule P-EXPAND quantifies over all values V and asks that each $N\,\square\,V \in [\![C \rightarrow D]\!]^-$ and $N\,V\,\square \in [\![C \rightarrow D]\!]^-$.

The remaining rules are inductive and address specific forms of contract. Every evaluation context fulfills a flat contract as expressed by rule P-FLAT. Rule P-OP considers the case where the hole is an argument of a built-in operation. As such an operation never invokes its arguments, this context respects any function contract. Rule P-APPLY is the archetypal context respecting $C \rightarrow D$ that applies the subject to an argument satisfying C and puts it in a context satisfying D.

An evaluation context respects a union contract $C \cup D$ if it respects both C and D according to rule P-UNION. An elimination context $\mathcal{F}$ respects an intersection contract $C \cap D$ if it respects C or D as codified in rules P-INTER-L and P-INTER-R.

This semantics of contract satisfaction and contract respect is not computable, in general. Fortunately, the non-computability is not an issue for the contract monitoring application that we have in mind. First, there are many special cases involving flat contracts, for example, that are decidable. But more importantly, our foremost goal is finding contract violations! Such a violation is a concrete, computable evidence that a subject (context) *does not* satisfy (respect) a contract. Such evidence can be constructed by the contract monitor specified in Section 4, after establishing some basic metatheoretical properties of contract satisfaction.

3.3 Properties of Contract Satisfaction

We start with some easy consequences of the semantics definition. Proposition 3 is needed for contract normalization in Section 4.2. We write $(\!|\mathcal{L}|\!)$ for the set ranged over by metavariable $\mathcal{L}$.

Proposition 1. $[\![\mathit{flat}(M)]\!]^- = (\!|\mathcal{L}|\!)$

Proposition 2. $[\![C \cup D]\!]^- = [\![C]\!]^- \cap [\![D]\!]^-$.

Proposition 3. $[\![C_0 \cap (C \cup D)]\!]^- = [\![(C_0 \cap C) \cup (C_0 \cap D)]\!]^-$.

Due to the untyped setting of our calculus, the semantics of a contract may contain some unexpected expressions. One key observation is that an expression that does not reduce to a value (including expressions that diverge or get stuck) fulfills any contract.

Proposition 4. *Suppose that* $M \not\longrightarrow^* V$. *Then* $M \in [\![C]\!]^+$.

Dually, a context that does not reduce to an evaluation context respects any contract.

Proposition 5. *Suppose that* $\mathcal{L} \not\longrightarrow^* E$. *Then* $\mathcal{L} \in [\![C]\!]^-$.

Furthermore, any subject fulfills a flat contract whose "predicate" expression does not evaluate to a function.

Proposition 6. *If* $N \not\longrightarrow^* \lambda x.M$, *then* $\forall L$: $L \in [\![\mathit{flat}(N)]\!]^+$.

An expression that does not evaluate to a function fulfills any function contract.

$$\begin{array}{lcl}
M, N & += & M \;@^b\; C \;||\; V \;@^\iota\; \mathit{check}(M) \mid \mathit{blame}^\flat \\
I, J & ::= & \mathit{flat}(M) \\
Q, R & ::= & C \rightarrow D \mid Q \cap R \\
U, V, W & += & ||\; V \;@^\iota\; Q \\
b & ::= & \flat \;||\; \iota
\end{array}$$

$$\begin{array}{l}
E \mathrel{+}= E \;@^b\; C \;||\; V \;@^\iota\; \mathit{check}(E) \\
\mathcal{K} ::= \square \mid \mathcal{K} \cap D \mid Q \cap \mathcal{K}
\end{array}$$

$$\begin{array}{l}
\kappa \;::= \flat\blacktriangleleft(\iota) \mid b\blacktriangleleft(W) \mid b\blacktriangleleft(\iota \rightarrow \iota) \mid b\blacktriangleleft(\iota \cup \iota) \mid b\blacktriangleleft(\iota \cap \iota) \\
\varsigma \;::= \cdot \mid \kappa : \varsigma
\end{array}$$

The nondeterministic calculus requires two further extensions.

$$\begin{array}{lcl}
M, N & += & ||\; \langle M \;[\!]^\kappa\; N \rangle \\
E & += & ||\; \langle E \;[\!]^\kappa\; N \rangle \mid \langle M \;[\!]^\kappa\; E \rangle
\end{array}$$

Figure 6. Syntax extension for λ_V^{Con}

Proposition 7. *If* $L \not\longrightarrow^* \lambda x.M$, *then* $L \in [\![C \rightarrow D]\!]^+$.

The semantics of contracts is closed under reduction.

Proposition 8 (Closure under reduction)**.**

1. *If* $M \longrightarrow N$ *and* $M \in [\![C]\!]^+$, *then* $N \in [\![C]\!]^+$.
2. *If* $\mathcal{M} \longrightarrow \mathcal{N}$ *and* $\mathcal{M} \in [\![C]\!]^-$, *then* $\mathcal{N} \in [\![C]\!]^-$.

The semantics satisfies all requirements from Section 2.2 and 2.3.

Theorem 1. *The semantics for subject and context satisfaction fulfill **IS0**, **IS1**, **IS2**, **IC2**, **US0**, **US1**, **US2**, and **UC2**.*

Our intuitions about intersections and unions of flat contracts are supported by straightforward calculation with the semantics.

Theorem 2. *For all* L *and* N*:*

1. $[\![\mathit{flat}(\lambda x.L) \cap \mathit{flat}(\lambda x.N)]\!]^+ = [\![\mathit{flat}(\lambda x.L \wedge N)]\!]^+$
2. $[\![\mathit{flat}(\lambda x.L) \cup \mathit{flat}(\lambda x.N)]\!]^+ = [\![\mathit{flat}(\lambda x.L \vee N)]\!]^+$

3.4 Discussion

At first sight, our semantics may seem very liberal because terms that diverge or get stuck are contained in the satisfaction semantics $[\![C]\!]^+$ of any contract (Proposition 4). But this design just reflects that neither diverging computations nor errors are observable. It would be easy to implement another point of view in our calculus by mapping errors in primitive operations to newly introduced error constants and by making them total and strict in errors.

Several contract systems check that the subject of a function contract is indeed a function, whereas our semantics accepts any non-function as satisfying any function contract (Proposition 7). However, the function contract $C \mapsto D$ that first checks its subject to be a function may be implemented as syntactic sugar with an intersection contract:

$$C \mapsto D := \mathit{flat}(\mathit{isFunction}) \cap (C \rightarrow D)$$

This implementation cleanly separates the first-order part of the contract from its higher-order part up front, which happens under the rug in implemented systems.

4. Contract Monitoring

This section extends the base calculus λ_V to a calculus λ_V^{Con}, which serves as a *nondeterministic specification* for contract monitoring. We deliberately present the nondeterministic version because it is easier to understand and because it enables us to prove the properties of the calculus in Section 6 whereas the proof details become too complex when addressing the deterministic version directly.

$$
\begin{array}{llllll}
\textsc{Activate} & \varsigma, & E[V \mathbin{@^\flat} C] & \longrightarrow & \flat\blacktriangleleft(\iota) : \varsigma, & E[V \mathbin{@^\iota} C] & \iota \notin \varsigma \\
\textsc{Left} & \varsigma, & E[V \mathbin{@^\iota} (\mathcal{K}[I] \cap D)] & \longrightarrow & \iota\blacktriangleleft(\iota_1 \cap \iota_2) : \varsigma, & E[(V \mathbin{@^{\iota_1}} I) \mathbin{@^{\iota_2}} \mathcal{K}[D]] & \iota_1, \iota_2 \notin \varsigma \\
\textsc{Right} & \varsigma, & E[V \mathbin{@^\iota} (Q \cap \mathcal{K}[I])] & \longrightarrow & \iota\blacktriangleleft(\iota_1 \cap \iota_2) : \varsigma, & E[(V \mathbin{@^{\iota_1}} I) \mathbin{@^{\iota_2}} \mathcal{K}[Q]] & \iota_1, \iota_2 \notin \varsigma \\
\textsc{N-Union} & \varsigma, & E[V \mathbin{@^\iota} (\mathcal{K}[C \cup D])] & \longrightarrow & \iota\blacktriangleleft(\iota_1 \cup \iota_2) : \varsigma, & E[\langle V \mathbin{@^{\iota_1}} \mathcal{K}[C] \mathbin{[\!]^{\iota\blacktriangleleft(\iota_1 \cup \iota_2)}} V \mathbin{@^{\iota_2}} \mathcal{K}[D] \rangle] & \iota_1, \iota_2 \notin \varsigma \\
 & & & & & & \\
\textsc{I-Flat} & \varsigma, & E[V \mathbin{@^\iota} \mathit{flat}(M)] & \longrightarrow & \varsigma, & E[V \mathbin{@^\iota} \mathit{check}(M\, V)] & \\
\textsc{I-Unit} & \varsigma, & E[V \mathbin{@^\iota} \mathit{check}(W)] & \longrightarrow & \iota\blacktriangleleft(W) : \varsigma, & E[V] & \\
 & & & & & & \\
\textsc{N-Fun} & \varsigma, & E[(V \mathbin{@^\iota} (C \rightarrow D))\, W] & \longrightarrow & \iota\blacktriangleleft(\iota_1 \rightarrow \iota_2) : \varsigma, & E[(V\, (W \mathbin{@^{\iota_1}} C)) \mathbin{@^{\iota_2}} D] & \iota_1, \iota_2 \notin \varsigma \\
\textsc{N-Inter} & \varsigma, & E[(V \mathbin{@^\iota} (Q \cap R))\, W] & \longrightarrow & \iota\blacktriangleleft(\iota_1 \cap \iota_2) : \varsigma, & E[\langle (V \mathbin{@^{\iota_1}} Q)\, W \mathbin{[\!]^{\iota\blacktriangleleft(\iota_1 \cap \iota_2)}} (V \mathbin{@^{\iota_2}} R)\, W \rangle] & \iota_1, \iota_2 \notin \varsigma
\end{array}
$$

$$
\textsc{D-Con-Op}\quad \frac{U ::= K \mid \lambda x.M}{\varsigma, E[O(\vec{U}(V \mathbin{@^\iota} Q)\vec{W})] \longrightarrow \varsigma, E[O(\vec{U} V W)]}
\qquad\qquad
\textsc{I-Base}\quad \frac{M \longrightarrow N}{\varsigma, M \longrightarrow \varsigma, N}
$$

$$
\textsc{D-Split-App}\quad \varsigma, E[\langle L \mathbin{[\!]^\kappa} M \rangle\, N] \longrightarrow \varsigma, E[\langle L\, N \mathbin{[\!]^\kappa} M\, N \rangle]
\qquad
\textsc{D-App-Split}\quad \varsigma, E[V\, \langle L \mathbin{[\!]^\kappa} M \rangle] \longrightarrow \varsigma, E[\langle V\, L \mathbin{[\!]^\kappa} V\, M \rangle]
$$

$$
\textsc{D-Split-Op}\quad \varsigma, E[O(\vec{U} \langle L \mathbin{[\!]^\kappa} M \rangle \vec{M}] \longrightarrow \varsigma, E[\langle O(\vec{U} L \vec{M}) \mathbin{[\!]^\kappa} O(\vec{U} M \vec{M}) \rangle]
$$

$$
\textsc{D-Split-Check}\quad \varsigma, E[V \mathbin{@^\iota} \mathit{check}(\langle L \mathbin{[\!]^\kappa} M \rangle)] \longrightarrow \varsigma, E[\langle V \mathbin{@^\iota} \mathit{check}(L) \mathbin{[\!]^\kappa} V \mathbin{@^\iota} \mathit{check}(M) \rangle]
$$

$$
\textsc{D-Split-Con}\quad \varsigma, E[\langle L \mathbin{[\!]^\kappa} M \rangle \mathbin{@^b} C] \longrightarrow \varsigma, E[\langle L \mathbin{@^b} C \mathbin{[\!]^\kappa} M \mathbin{@^b} C \rangle]
\qquad
\textsc{I-Split-Collapse}\quad \varsigma, E[\langle M \mathbin{[\!]^\kappa} M \rangle] \longrightarrow \varsigma, E[M]
$$

Figure 7. Dynamics of λ_V^{Con}

4.1 Additional Syntax

Figure 6 defines the syntax of λ_V^{Con} as an extension of λ_V in two steps. The first step introduces constructs for contract monitoring in general. The second step adds the interleaving expression specific to nondeterministic monitoring. Intermediate terms that do not occur in source programs appear after double bars "||".

The only new source term is contract monitoring $M \mathbin{@^b} C$. Its adornment b is drawn from an unspecified denumerable set of blame identifiers, which comprises blame labels $\flat$ that occur in source terms and blame variables ι that are introduced during evaluation.

In the intermediate term $V \mathbin{@^\iota} \mathit{check}(M)$, the term M represents the current evaluation state of the predicate of a flat contract. The $\mathit{blame}^\flat$ expression signals a contract violation at label $\flat$. The two subcontracts of intersection and union contracts are monitored independently using the interleaving expression $\langle M \mathbin{[\!]^\kappa} N \rangle$. The superscript κ is the constraint generated on introduction of the interleaving and indicates whether the interleaving stands for an intersection or for a union.

To specify the dynamics, we refine the syntax of contracts. Contracts I and J stand for immediate, flat contracts that can be evaluated right away. A delayed contract, Q or R, is a finite intersection of function contracts. It stays with a value until it is used. Consequently, values are extended with $V \mathbin{@^\iota} Q$ which represents a value wrapped in a delayed contract that is to be monitored when the value is used in an elimination context (e.g., on function application).

The extended set of values forces us to revisit the built-in operations. We posit that each partial function δ_O first erases all contract monitoring from its arguments, then processes the underlying λ_V-values, and finally returns a λ_V-value.

Evaluation contexts are extended in the obvious way: a contract monitor is only applied to a value and a flat contract is checked before its value is used. To reflect the independence of monitoring subcontracts of intersections and unions, interleavings reduce *non-deterministically*: each evaluation step may choose to reduce the left or right component.

Contract contexts $\mathcal{K}$ are needed for normalizing nested applications of intersection and union contracts. They are explained in Section 4.2.

In λ_V^{Con}, contract monitoring occurs via constraints κ imposed on blame identifiers. There is an indirection constraint and one kind of constraint for each kind of contract: flat, function, intersection, and union. Constraints are collected in a list ς during reduction.

4.2 Reduction

Figure 12 specifies the small-step reduction semantics of λ_V^{Con} as a relation $\varsigma, M \longrightarrow \varsigma', N$ on pairs of a constraint list and an expression. Instead of raising blame exceptions, the rewriting rules for contract enforcement generate constraints in ς: a failing contract must not raise blame immediately, because it may be nested in an intersection or a union. The sequence of elements in the constraint list reflects the temporal order in which the constraints were generated during reduction. The latest, youngest constraints are always on top of the list. Section 4.3 explains the semantics of the constraints and Section 4.3.2 explains the role of the temporal order for the semantics.

The rule Activate introduces a fresh name for each new instantiation of a monitor in the source program. It is needed for technical reasons to establish the simulation relation with the deterministic version of the semantics.

The first group of rules Left, Right, and N-Union implements contract normalization. Normalization has two purposes. Rules Left and Right factorize a contract into an immediate part I and a rest contract. The idea is that a flat contract I that is nested only in intersections (cf. the constraint context $\mathcal{K}$) may be pulled out and checked directly. The logical justification for these rules is associativity of intersection (and union): for instance, $[\![\mathcal{K}[I] \cap$

$D]\!]^+ = [\![I \cap \mathcal{K}[D]]\!]^+$; so their satisfaction semantics stays the same. Both rules also install constraints that combine the contract satisfaction of the subcontracts to the satisfaction of the intersection.

The N-UNION rule embodies the introduction-site choice of the union. If the current contract has the form $\mathcal{K}[C \cup D]$ (i.e., a union nested in a context of intersections), then the union is pulled out by distributivity (Proposition 3) resulting in $\mathcal{K}[C] \cup \mathcal{K}[D]$. Then the expression is split to monitor $\mathcal{K}[C]$ and $\mathcal{K}[D]$ in isolation and a constraint is installed to combine the outcomes of monitoring $\mathcal{K}[C]$ and $\mathcal{K}[D]$ according to **US2** and **UC2**. Thus, for each value with a union contract, the constraint generated by this rule application is the single point that chooses between $\mathcal{K}[C]$ and $\mathcal{K}[D]$.

Flat contracts get evaluated immediately. Rule I-FLAT starts checking a flat contract by evaluating the predicate M applied to the subject value V. After predicate evaluation, rule I-UNIT picks up the result and stores it in a constraint.

The reduction rules N-FUN and N-INTER define the behavior of a contracted value under function application, that is, in a particular elimination context. Rule N-FUN handles a call to a value with a function contract. Different from previous work, the blame computation is handled indirectly by creating new blame variables for the domain and range part; a new constraint is added that transforms the outcome of both portions according to the specification of the function contract. Rule N-INTER duplicates the function application for each conjunct to monitor them concurrently in isolation. The generated constraint serves to combine the results of the subcontracts. Unlike the union contract, splitting for an intersection occurs at each use of the contracted value, which implements the choice of the context.

Built-in operations can "see through" contracts (rule D-CON-OP). An interleaving may be collapsed nondeterministically if its components are equal (IPAIRCOLLAPSE). Finally, reductions of λ_V are lifted to λ_V^{Con} using rule IBASE. This choice implies that an OP reduction only returns a λ_V value that does not contain contracts.

The rules D-SPLIT-APP, D-APP-SPLIT, D-SPLIT-CON, D-SPLIT-OP, and D-SPLIT-CHECK deal with occurrences of interleavings in contexts where they may hinder other reductions. They nondeterministically duplicate the immediately surrounding construct.

4.3 Constraints

The dynamics in Figure 12 use constraints to create a structure for computing positive and negative blame according to the semantics of subject and context satisfaction, respectively. To this end, each blame identifier b is associated with two truth values, *b.subject* and *b.context*. Intuitively, if *b.subject* is false, then the contract b is not subject-satisfied and may lead to positive blame for b. If *b.context* is false, then there is a context that does not respect contract b and may lead to negative blame for b. But the story is more complicated.

4.3.1 Constraint Satisfaction

A *solution* μ of a constraint list ς is a mapping from blame identifiers to records of elements of $\mathbb{B} = \{\mathtt{t}, \mathtt{f}\}$, such that all constraints are satisfied. We order truth values by $\mathtt{t} \sqsubset \mathtt{f}$ and write $\sqsubseteq$ for the reflexive closure of that ordering. Formally, we specify the mapping by

$$\mu \in ((\!|b|\!) \times \{subject, context\}) \to \mathbb{B}$$

and constraint satisfaction by a relation $\mu \models \varsigma$, which is specified in Figure 9. In the premisses, the rules apply a constraint mapping μ to boolean expressions over constraint variables. This application stands for the obvious homomorphic extension of the mapping.

Every mapping satisfies the empty list of constraints (CS-EMPTY). The cons of a constraint with a constraint list corresponds to the intersection of sets of solutions (CS-CONS). The indirection constraint just forwards its referent (CT-IND).

$$\tau(V) = \begin{cases} \mathtt{f} & V = \mathit{false} \\ \tau(W) & V = W \;@^\iota\; Q \\ \mathtt{t} & \text{otherwise} \end{cases}$$

Figure 8. Mapping values to truth values

CS-EMPTY
$$\mu \models \cdot$$

CS-CONS
$$\frac{\mu \models \kappa \qquad \mu \models \varsigma}{\mu \models \kappa : \varsigma}$$

CT-IND
$$\frac{\mu(\flat.\mathit{subject}) \sqsupseteq \mu(\iota.\mathit{subject}) \qquad \mu(\flat.\mathit{context}) \sqsupseteq \mu(\iota.\mathit{context})}{\mu \models \flat \blacktriangleleft (\iota)}$$

CT-FLAT
$$\frac{\mu(b.\mathit{subject}) \sqsupseteq \tau(V) \qquad \mu(b.\mathit{context}) \sqsupseteq \mathtt{t}}{\mu \models b \blacktriangleleft V}$$

CT-FUNCTION
$$\frac{\begin{array}{c}\mu(b.\mathit{subject}) \sqsupseteq \mu(\iota_1.\mathit{context} \wedge (\iota_1.\mathit{subject} \Rightarrow \iota_2.\mathit{subject})) \\ \mu(b.\mathit{context}) \sqsupseteq \mu(\iota_1.\mathit{subject} \wedge \iota_2.\mathit{context})\end{array}}{\mu \models b \blacktriangleleft \iota_1 \to \iota_2}$$

CT-INTERSECTION
$$\frac{\begin{array}{c}\mu(b.\mathit{subject}) \sqsupseteq \mu(\iota_1.\mathit{subject} \wedge \iota_2.\mathit{subject}) \\ \mu(b.\mathit{context}) \sqsupseteq \mu(\iota_1.\mathit{context} \vee \iota_2.\mathit{context})\end{array}}{\mu \models b \blacktriangleleft \iota_1 \cap \iota_2}$$

CT-UNION
$$\frac{\begin{array}{c}\mu(b.\mathit{subject}) \sqsupseteq \mu(\iota_1.\mathit{subject} \vee \iota_2.\mathit{subject}) \\ \mu(b.\mathit{context}) \sqsupseteq \mu(\iota_1.\mathit{context} \wedge \iota_2.\mathit{context})\end{array}}{\mu \models b \blacktriangleleft \iota_1 \cup \iota_2}$$

Figure 9. Constraint satisfaction

In rule CT-FLAT, W is the outcome of the predicate of a flat contract. The rule sets subject satisfaction to $\mathtt{f}$ if $W = \mathit{false}$ and otherwise to $\mathtt{t}$, where the function $\tau(\cdot) : (\!|V|\!) \to \mathbb{B}$ translates values to truth values by stripping delayed contracts (see Figure 8). A flat contract never blames its context so that *b.context* is always true.

The rule CT-FUNCTION determines the blame assignment for a function contract b from the blame assignment for the argument and result contracts, which are available through ι_1 and ι_2. Let's first consider the subject part. A function f satisfies contract b if it satisfies its obligations towards its argument ι_1*.context* **and** if the argument satisfies its contract then the result satisfies its contract, too. The first part arises if f is a higher-order function, which may pass illegal arguments to its function-arguments. The second part is partial correctness of the function with respect to its contract.

A function's context (caller) satisfies the contract if it passes an argument that satisfies contract ι_1*.subject* **and** uses the result according to its contract ι_2*.context*. The second part becomes non-trivial with functions that return functions.

The rule CT-INTERSECTION determines the blame assignment for an intersection contract at b from its constituents at ι_1 and ι_2. A subject satisfies an intersection contract if it satisfies both constituent contracts: $\iota_1.\mathit{subject} \wedge \iota_2.\mathit{subject}$ (cf. **IS2**). A context, however, has the choice to fulfill one of the constituent contracts: $\iota_1.\mathit{context} \vee \iota_2.\mathit{context}$ (cf. **IC2**).

Dually, the rule CT-UNION determines the blame assignment for a union contract at b from its constituents at ι_1 and ι_2 according to **US2** and **UC2**. A subject satisfies a union contract if

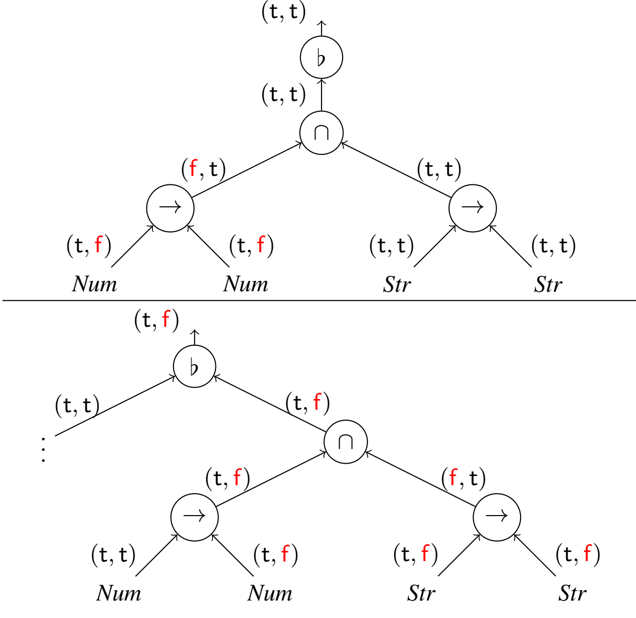

Figure 10. Blame calculation for *addOne* $= \lambda x.(x + \text{"}1\text{"})$ with contract $(Num \to Num) \cap (Str \to Str)$. The top picture shows the constraint graph after applying *addOne* to the string *"1"* (first call). The bottom picture shows the extended graph after applying *addOne* to the number *1* (second call). Each node is a constraint. Each edge is a reference to a blame variable. The labeling next to the arrow shows the record (*context*, *subject*) assigned by the solution of the underlying constraint list. The root ($\flat$) collects the outcome of all delayed contract assertions.

$\cdot$
$[]$
$((\lambda f.f)\ @^0\ ((P \to P) \to N))\ (\lambda x. -x)\ 42$
$\xrightarrow{(1)} 0\blacktriangleleft(1 \to 2) : \cdot$
$[0 \mapsto (\mathsf{t},\mathsf{t}), 1 \mapsto (\mathsf{t},\mathsf{t}), 2 \mapsto (\mathsf{t},\mathsf{t})]$
$(((\lambda f.f)\ ((\lambda x. -x)\ @^1\ (P \to P)))\ @^2\ N)\ 42$
$\xrightarrow{(2)} 0\blacktriangleleft(1 \to 2) : \cdot$
$[0 \mapsto (\mathsf{t},\mathsf{t}), 1 \mapsto (\mathsf{t},\mathsf{t}), 2 \mapsto (\mathsf{t},\mathsf{t})]$
$(((\lambda x. -x)\ @^1\ (P \to P))\ @^2\ N)\ 42$
$\xrightarrow{(3)} 2\blacktriangleleft(\mathit{ff}) : 0\blacktriangleleft(1 \to 2) : \cdot$
$[0 \mapsto (\mathsf{t},\mathsf{f}), 1 \mapsto (\mathsf{t},\mathsf{t}), 2 \mapsto (\mathsf{t},\mathsf{f})]$
$((\lambda x. -x)\ @^1\ (P \to P))\ 42$
$\xrightarrow{(4)} 1\blacktriangleleft(3 \to 4) : 2\blacktriangleleft(\mathit{ff}) : 0\blacktriangleleft(1 \to 2) : \cdot$
$[0 \mapsto (\mathsf{t},\mathsf{f}), 1 \mapsto (\mathsf{t},\mathsf{t}), 2 \mapsto (\mathsf{t},\mathsf{f}), 3 \mapsto (\mathsf{t},\mathsf{t}), 4 \mapsto (\mathsf{t},\mathsf{t})]$
$((\lambda x. -x)\ (42\ @^3\ P))\ @^4\ P$
$\xrightarrow{(5)} 3\blacktriangleleft(\mathit{tt}) : 1\blacktriangleleft(3 \to 4) : 2\blacktriangleleft(\mathit{ff}) : 0\blacktriangleleft(1 \to 2) : \cdot$
$[0 \mapsto (\mathsf{t},\mathsf{f}), 1 \mapsto (\mathsf{t},\mathsf{t}), 2 \mapsto (\mathsf{t},\mathsf{f}), 3 \mapsto (\mathsf{t},\mathsf{t}), 4 \mapsto (\mathsf{t},\mathsf{t})]$
$((\lambda x. -x)\ 42)\ @^4\ P$
$\xrightarrow{(6)} 3\blacktriangleleft(\mathit{tt}) : 1\blacktriangleleft(3 \to 4) : 2\blacktriangleleft(\mathit{ff}) : 0\blacktriangleleft(1 \to 2) : \cdot$
$[0 \mapsto (\mathsf{t},\mathsf{f}), 1 \mapsto (\mathsf{t},\mathsf{t}), 2 \mapsto (\mathsf{t},\mathsf{f}), 3 \mapsto (\mathsf{t},\mathsf{t}), 4 \mapsto (\mathsf{t},\mathsf{t})]$
$(-42)\ @^4\ P$
$\xrightarrow{(7)} 4\blacktriangleleft(\mathit{ff}) : 3\blacktriangleleft(\mathit{tt}) : 1\blacktriangleleft(3 \to 4) : 2\blacktriangleleft(\mathit{ff}) : 0\blacktriangleleft(1 \to 2) : \cdot$
$[0 \mapsto (\mathsf{f},\mathsf{t}), 1 \mapsto (\mathsf{t},\mathsf{f}), 2 \mapsto (\mathsf{t},\mathsf{f}), 3 \mapsto (\mathsf{t},\mathsf{t}), 4 \mapsto (\mathsf{t},\mathsf{f})]$
-42

Figure 11. Example reduction sequence. Each item comprises the constraint list, its solution (in gray), and the term

it satisfies one of the constituent contracts: $\iota_1.\mathit{subject} \vee \iota_2.\mathit{subject}$. A context, however, needs to fulfill both constituent contracts: $\iota_1.\mathit{context} \wedge \iota_2.\mathit{context}$, because it does not know which contract is satisfied by the subject.

Figure 10 illustrates blame calculation with constraints using the function *addOne* $= \lambda x.(x + \text{"}1\text{"})$ with the contract $(Num \to Num) \cap (Str \to Str)$. After applying *addOne* to *"1"* the contract $(Num \to Num)$ fails and blames the context, whereas the second contract $(Str \to Str)$ succeeds. Because the context of an intersection may choose which side to fulfill, the intersection is satisfied.

A second call which applies *addOne* to *1* leads to blame: $Num \to Num$ fails, blaming the subject, because the result is a string; $Str \to Str$ fails and blames the context. In this case, the intersection contract blames the subject because it has to satisfy both contracts.

4.3.2 Solving Constraints

Computing a blame assignment boils down to computing a solution for a constraint list ς. To this end, we define its *dependency graph* $DG(\varsigma)$. Its nodes are blame identifiers and there is an edge from ι_k to b if there is a constraint with b on the left side and ι_k on the right side: $b\blacktriangleleft(\dots \iota_k \dots) \in \varsigma$. An easy induction on reduction sequences (Figure 12) shows that $DG(\varsigma)$ is always a forest.

Proposition 9. *If* $\emptyset, M \longrightarrow^* \varsigma, N$*, then* $DG(\varsigma)$ *is a forest rooted in the blame labels* $\flat$*.*

A least solution $LSol(\varsigma) \in (\llparenthesis b \rrparenthesis \times \{subject, context\}) \to \mathbb{B}$ can be computed for any constraint list ς arising during reduction by evaluating the constraints for the blame variables in some topological order consistent with $DG(\varsigma)$. That is, $LSol(\varsigma) \models \varsigma$ and $LSol(\varsigma) \sqsubseteq \mu$ for all $\mu \models \varsigma$. Recall that, due to the ordering on $\mathbb{B}$, the function $LSol(\cdot)$ with $LSol(\cdot)(b, x) = \mathsf{t}$ is the least element of $(\llparenthesis b \rrparenthesis \times \{subject, context\}) \to \mathbb{B}$.

To establish our technical results, we would like to argue that constraint solutions grow monotonically when the constraint list is extended. Unfortunately, the least solution $LSol(\kappa : \varsigma)$ for an extended constraint list is not always comparable to the solution $LSol(\varsigma)$ for the original list.

For an example, consider the reduction sequence in Figure 11 where P and N are shorthands for the contracts *Pos* and *Num* and *tt* and *ff* stand for *true* and *false*. We write the subject part of the respective least solutions as a mapping from blame variable to truth value. In the last reduction step (7), the least solution of the constraint list changes non-monotonically: $0.\mathit{subject}$ changes from f to t. If we included the context blame, we would see that at the same step $0.\mathit{context}$ changes from t to f: Blame is transferred from the subject to the context due to the non-monotonicity of implication.

Let's analyze this baffling situation. When enforcing a function contract, execution first finds a violation of the range contract—giving rise to subject blame—and then a violation of the domain contract—giving rise to context blame. One may argue that a contract monitor would already raise blame in step (3) when $0.\mathit{subject}$ flips to f. However, this contract may appear nested in a union contract so that blaming would be delayed and reduction would continue as in the above reduction.

Our choice, which we make formal in Section 6, is to follow the lead of implemented systems (without union and intersection) that always report the first contract violation. We capture this preference for the first violation by defining a monotone constraint semantics, which first cleans the constraint list from later constraints that violate monotonicity and then takes the least solution.

Definition 1. *The* monotone constraint semantics *is* $[\![\varsigma]\!] = \mathit{LSol}(\mathit{Clean}(\varsigma))$ *where* $\mathit{Clean}() : (\!|\varsigma|\!) \to (\!|\varsigma|\!)$ *is defined by*

$$\begin{array}{lcl} \mathit{Clean}(\cdot) & = & \cdot \\ \mathit{Clean}(\kappa : \varsigma) & = & \begin{cases} \kappa : \varsigma' & \mathit{LSol}(\varsigma') \sqsubseteq \mathit{LSol}(\kappa : \varsigma') \\ \varsigma' & \mathit{otherwise} \end{cases} \\ & & \mathit{where}\ \varsigma' = \mathit{Clean}(\varsigma) \end{array}$$

Proposition 10. *For all* ς *and* κ, $[\![\varsigma]\!] \sqsubseteq [\![\kappa : \varsigma]\!]$.

The implementation is straightforward: each constraint $b◀(\dots)$ is only allowed to "fire" once and sets either $b.\mathit{subject}$ or $b.\mathit{context}$ to f. Afterwards the constraint becomes inactive.

4.3.3 Introducing Blame

To determine whether a constraint list ς is a blame state (i.e., whether it should signal a contract violation), we check whether the semantics $[\![\varsigma]\!]$ maps any source-level blame label $\flat$ to false.

Definition 2. ς *is a* blame state for blame label $\flat$ *iff*

$$[\![\varsigma]\!](\flat.\mathit{subject} \wedge \flat.\mathit{context}) \sqsupseteq \mathsf{f}.$$

ς *is a* blame state *if there exists a blame label* $\flat$ *such that* ς *is a blame state for this label.*

To model reduction with blame, we define a new reduction relation $\varsigma, M \longmapsto \varsigma', M'$ on configurations. It behaves like $\longrightarrow$ unless ς is a blame state. In a blame state, it stops signaling the violation. There are no reductions with *blame*.

$$\frac{\varsigma, M \longrightarrow \varsigma', N \qquad \varsigma \text{ is not a blame state}}{\varsigma, M \longmapsto \varsigma', N} \qquad \frac{\varsigma \text{ is blame state for } \flat}{\varsigma, M \longmapsto \varsigma, \mathit{blame}^{\flat}}$$

4.3.4 Lifting Definitions

The present section tacitly lifts various semantic notions and results from λ_V to the extended calculus λ_V^{Con} with monitoring. In this subsection, we make this lifting precise.

A number of definitions and results in Section 3 refer to reduction and context reduction in λ_V. These definitions (semantics of contracts and the results in Section 3.3) are lifted to λ_V^{Con} by taking $M \longrightarrow N$ as a shorthand for $\forall\varsigma.\exists\varsigma'.\varsigma, M \longmapsto \varsigma', N$ (top-level reduction with *blame*). The same lifting applies to $\mathcal{M} \longrightarrow \mathcal{N}$.

The coinductive definition of $[\![C]\!]^-$ in Figure 5 is extended with additional rules for the extra syntactic constructs. Most of the new rules just cater to reductions with a hole in place of the value. Context reduction needs to be extended, which is a straightforward.

5. Deterministic Monitoring

The calculus $\lambda_{Vd}^{\mathit{Con}}$ provides a deterministic reduction semantics for contract monitoring. Its syntax is identical to λ_V^{Con}, but without the interleaving expression (i.e., Figure 6 top only). Figure 12 specifies its one-step reduction relation on expressions. It is surrounded by a top level reduction $\varsigma, M \longmapsto \varsigma', N$ that reduces M only if ς is not a blame state according to Definition 2. Its definition is analogous to the one in Section 4.3.3

Just like nondeterministic reduction, contract monitoring normalizes contracts before it starts their enforcement. This part of the rule set is identical to the nondeterministic rules, except for the rule D-UNION which implements union contracts by enforcing the contracts in some order instead of interleaving their execution.

The rules I-FLAT and I-UNIT that deal with flat contracts are identical to the nondeterministic version.

The remaining rules implement monitoring of delayed contracts. As in λ_V^{Con}, the assertion of a delayed contract assumes that the value is a function and wraps it so that the contract is checked when the function is applied. The rules D-FUN, D-INTER, and DROP act when such a wrapped value is applied to an argument W. Compared to λ_V^{Con}, these rules need to take into account the new notion of *compatibility*. Roughly, two flat contract executions are compatible if they belong to the same component of a nested union/intersection.

In λ_V^{Con}, execution is compartmentalized by interleave expressions that mimic the nesting of currently active union and intersection contracts. As we saw in the D-UNION rule, $\lambda_{Vd}^{\mathit{Con}}$ intermingles the execution of contracts from all compartments. Compatibility of a contract with its evaluation context is defined such that contracts from different compartments are never mixed up. We come back to compatibility after explaining the rules.

The rule D-FUN handles the call of a contracted function. If differs from the λ_V^{Con}-rule N-FUN only in the side condition of compatibility. Rule D-INTER sequentially applies both contracts in terms of a new constraint, but only if the intersection is compatible with the context. Rule DROP drops a delayed contract that is not compatible with the current evaluation context.

5.1 Compatibility

To illustrate the need for compatibility, we consider the contract

$$\begin{array}{l} C = ((P^7 \to^5 P^8) \to^1 \mathit{FP}^6) \cap^0 ((N^9 \to^3 N^a) \to^2 \mathit{FN}^4) \quad (8) \\ \text{where} \quad P = \mathit{flat}(\lambda x.x > 0) \quad \mathit{FP} = \mathit{flat}(\lambda f.f\,1 > 0) \\ \qquad\qquad N = \mathit{flat}(\lambda x.x < 0) \quad \mathit{FN} = \mathit{flat}(\lambda f.f\,(-1) < 0) \end{array}$$

Semantically, it is clear that

$$\lambda f.f \in [\![C]\!]^+ = [\![(P \to P) \to \mathit{FP}]\!]^+ \cap [\![(N \to N) \to \mathit{FN}]\!]^+.$$

But if we reduce the configuration

$$\cdot, ((\lambda f.f) @^{\flat} C)\,(\lambda x.x)\,42$$

ignoring the compatibility side conditions on D-FUN and D-INTER (and omitting rule DROP) then, after a few steps,[5] we arrive at a configuration that blames the subject wrongly:[6]

$$\begin{array}{r} 3◀(9 \to a) : 7◀(\mathit{true}) : 5◀(7 \to 8) : 1◀(5 \to 6) : \dots \\ 2◀(3 \to 4) : 0◀(1 \cap 2) : \flat◀(0) : \cdot \\ \dots @^6\ \mathit{check}((((\lambda x.x)\,(1 @^9 N)) @^a N) @^8 P > 0) \dots \end{array} \quad (9)$$

Blame is triggered by the next step that reduces $(1 @^9 N)$ to 1 and adds the constraint $9◀(\mathit{false})$. It is caused by the evaluation of a flat contract *FP* on an argument that is wrapped in the function contracts $P \to P$ and $N \to N$. The problem is that the contract $N \to N$ does not belong to the same operand of the intersection as *FP* and thus $N \to N$ must not be enforced in the body of *FP*.

We can determine this mismatch by recognizing that the superscript of $\dots @^6\ \mathit{check}(\dots)$, belongs to the left component of $0◀(1 \cap 2)$ whereas the superscript of $1 @^9 N$ belongs to the right component. We avoid such mismatches altogether by ignoring delayed contracts that originate from a different compartment of an enclosing union or intersection contract. Thus, the D-FUN and D-INTER rules must verify that all enclosing $\mathit{check}(\dots)$ expressions in the evaluation context are *compatible* with the contract. The companion rule DROP drops incompatible delayed contracts.

To define compatibility, we first identify the contract component to which a blame variable belongs. To do so we compute the unique path from a source-level blame label to the blame variable in the dependency graph of a constraint list (a forest by Lemma 9). Each step of a path is drawn from the set $\mathit{Step} = \{\to, \cap, \cup, \downarrow\} \times \{1, 2\} \times (\!|\iota|\!)$ and denote the left/right (1/2) subcontract of a function, intersection, or union contract along with the constraint variable at that position. The symbol $\downarrow$ stands for an indirection constraint and its single subcontract is always at position 1.

[5] The full reduction sequence may be inspected in the supplement.

[6] Blame variables are chosen to match the superscripts in Equation (8).

$$
\begin{array}{llllll}
\textsc{Beta} & \varsigma, & E[(\lambda x.M)\,V] & \longrightarrow & \varsigma, & E[M[x \mapsto V]] & \\
\textsc{D-Con-Op} & \varsigma, & E[O(\vec{U}(V \mathbin{@^{b}} Q)\vec{W})] & \longrightarrow & \varsigma, & E[O(\vec{U} V \vec{W})] & U ::= K \mid \lambda x.M \\
\textsc{Op} & \varsigma, & E[O(\vec{V})] & \longrightarrow & \varsigma, & E[\delta_O(\vec{V})] & V ::= K \mid \lambda x.M, \\
 & & & & & & \vec{V} \in \mathit{dom}(\delta_O) \\
\textsc{Activate} & \varsigma, & E[V \mathbin{@^{\flat}} C] & \longrightarrow & \flat \blacktriangleleft (\iota) : \varsigma, & E[V \mathbin{@^{\iota}} C] & \iota \notin \varsigma \\
\textsc{Left} & \varsigma, & E[V \mathbin{@^{\iota}} (\mathcal{K}[I] \cap D)] & \longrightarrow & \iota \blacktriangleleft (\iota_1 \cap \iota_2) : \varsigma, & E[(V \mathbin{@^{\iota_1}} I) \mathbin{@^{\iota_2}} \mathcal{K}[D]] & \iota_1, \iota_2 \notin \varsigma \\
\textsc{Right} & \varsigma, & E[V \mathbin{@^{\iota}} (Q \cap \mathcal{K}[I])] & \longrightarrow & \iota \blacktriangleleft (\iota_1 \cap \iota_2) : \varsigma, & E[(V \mathbin{@^{\iota_1}} I) \mathbin{@^{\iota_2}} \mathcal{K}[Q]] & \iota_1, \iota_2 \notin \varsigma \\
\textsc{D-Union} & \varsigma, & E[V \mathbin{@^{\iota}} (\mathcal{K}[C \cup D])] & \longrightarrow & \iota \blacktriangleleft (\iota_1 \cup \iota_2) : \varsigma, & E[(V \mathbin{@^{\iota_1}} \mathcal{K}[C]) \mathbin{@^{\iota_2}} \mathcal{K}[D]] & \iota_1, \iota_2 \notin \varsigma \\
\textsc{I-Flat} & \varsigma, & E[V \mathbin{@^{\iota}} \mathit{flat}(M)] & \longrightarrow & \varsigma, & E[V \mathbin{@^{\iota}} \mathit{check}(M\,V)] & \\
\textsc{I-Unit} & \varsigma, & E[V \mathbin{@^{\iota}} \mathit{check}(W)] & \longrightarrow & \iota \blacktriangleleft (W) : \varsigma, & E[V] & \\
\textsc{D-Fun} & \varsigma, & E[(V \mathbin{@^{\iota}} (C \rightarrow D))\,W] & \longrightarrow & \iota \blacktriangleleft (\iota_1 \rightarrow \iota_2) : \varsigma, & E[(V\,(W \mathbin{@^{\iota_1}} C) \mathbin{@^{\iota_2}} D] & \iota_1, \iota_2 \notin \varsigma, \mathit{comp}_{\varsigma}(E, \iota) \\
\textsc{D-Inter} & \varsigma, & E[(V \mathbin{@^{\iota}} (Q \cap R))\,W] & \longrightarrow & \iota \blacktriangleleft (\iota_1 \cap \iota_2) : \varsigma, & E[((V \mathbin{@^{\iota_1}} Q) \mathbin{@^{\iota_2}} R)\,W] & \iota_1, \iota_2 \notin \varsigma, \mathit{comp}_{\varsigma}(E, \iota) \\
\textsc{Drop} & \varsigma, & E[(V \mathbin{@^{\iota}} Q)\,W] & \longrightarrow & \varsigma, & E[V\,W] & \neg \mathit{comp}_{\varsigma}(E, \iota)
\end{array}
$$

Figure 12. Operational semantics of λ_{Vd}^{Con}; reductions in gray are identical to λ_{V}^{Con} reductions

$$
\mathit{comp}(\varepsilon, \rho) \qquad \mathit{comp}(\pi, \varepsilon) \qquad \frac{\iota_1 \neq \iota_2}{\mathit{comp}((\downarrow, 1, \iota_1).\pi, (\downarrow, 1, \iota_2).\rho)}
$$

$$
\frac{\iota_1 \neq \iota_2 \qquad i_1, i_2 \in \{1, 2\}}{\mathit{comp}((\rightarrow, i_1, \iota_1).\pi, (\rightarrow, i_2, \iota_2).\rho)}
$$

$$
\frac{\mathit{comp}(\pi, \rho)}{\mathit{comp}((\Diamond, i, \iota).\pi, (\Diamond, i, \iota).\rho)}
$$

Figure 13. Compatibility of paths

$$
\mathit{comp}_{\varsigma}(\square, b) \qquad \frac{\mathit{comp}_{\varsigma}(E, b)}{\mathit{comp}_{\varsigma}(O(\vec{V}\,E\,\vec{M}), b)} \qquad \frac{\mathit{comp}_{\varsigma}(E, b)}{\mathit{comp}_{\varsigma}(E\,M, b)}
$$

$$
\frac{\mathit{comp}_{\varsigma}(E, b)}{\mathit{comp}_{\varsigma}(V\,E, b)} \qquad \frac{\mathit{comp}_{\varsigma}(E, b)}{\mathit{comp}_{\varsigma}(E \mathbin{@^{b_0}} C, b)}
$$

$$
\frac{\mathit{comp}_{\varsigma}(E, b) \qquad \mathit{comp}_{\varsigma}(b, b_0)}{\mathit{comp}_{\varsigma}(V \mathbin{@^{b_0}} \mathit{check}(E), b)}
$$

Figure 14. Compatibility with an evaluation context

Definition 3. *Define $\mathit{Path}(\varsigma, b) \subseteq \mathit{Step}^{+}$ by induction on the length of the unique path in $\mathit{DG}(\varsigma)$ from b to a root $\flat$.*

$$
\mathit{Path}(\varsigma, b) = \begin{cases} (\downarrow, 1, \flat) & b = \flat \\ \mathit{Path}(\varsigma, b_0).(\downarrow, 1, \iota) & b_0 \blacktriangleleft (\iota) \in \varsigma, b = \iota \\ \mathit{Path}(\varsigma, b_0).(\Diamond, i, b) & b_0 \blacktriangleleft (\iota_1 \Diamond \iota_2) \in \varsigma, b = \iota_i, \\ & i \in \{1, 2\}, \Diamond \in \{\rightarrow, \cap, \cup, \downarrow\} \end{cases}
$$

Let ς^{*} be the final state of the reduction sequence in (9) and consider the paths that belong to the blame variables 6 on the *check*() and 9 on the flat contract that triggers the failure.

$$
\begin{array}{ll}
\mathit{Path}(\varsigma^{*}, 6) = & (\downarrow, 1, \flat).(\downarrow, 1, 0).(\cap, 1, 1).(\rightarrow, 2, 6) \\
\mathit{Path}(\varsigma^{*}, 9) = & (\downarrow, 1, \flat).(\downarrow, 1, 0).(\cap, 2, 2).(\rightarrow, 1, 3).(\rightarrow, 1, 9)
\end{array}
$$

The paths clearly indicate that the two blame variables belong to different operands of the same intersection and thus are incompatible. It remains to formally define compatibility.

Definition 4. *Two paths $\pi, \rho \in \mathit{Step}^{*}$ are* compatible *if $\mathit{comp}(\pi, \rho)$ is derivable from the set of inductive rules in Figure 13.*

Two blame identifiers are compatible with respect to a constraint list ς if the corresponding paths are compatible:

$$
\mathit{comp}_{\varsigma}(b_1, b_2) = \mathit{comp}(\mathit{Path}(\varsigma, b_1), \mathit{Path}(\varsigma, b_2)).
$$

An evaluation context is compatible with a blame identifier, $\mathit{comp}_{\varsigma}(E, b)$, if b is compatible with all blame identifiers of the check expressions traversed in E as defined by the rules in Figure 14.

Two paths are compatible if one is a prefix of the other or if they have a common prefix and then proceed with different indirections or with different subcontracts of a function contract. The rationale is that different indirections are created for different instantiations of the same contract: different instantiations are independent of one another so that these instantiations may interact arbitrarily. Similarly, the domain and the range part of a function contract are independent and their (sub-) contracts may interact arbitrarily.

Compatibility with an evaluation context must consider all pending contract checks because compatibility is not transitive.

Proposition 11. *Compatibility of blame identifiers is reflexive and symmetric, but not transitive.*

5.2 Simulation

The nondeterministic semantics of λ_{V}^{Con} and the deterministic semantics of λ_{Vd}^{Con} are related by a simulation relation. Whenever the deterministic semantics makes a step, then this evaluation step can be simulated in the nondeterministic semantics in zero or more steps. For lack of space, we defer the technical details of the simulation relation $\varsigma \vdash M \succeq_B M'$ to the supplement. The relation is indexed by a constraint list ς and a set B of blame variables that indicate the path to the current compartment. An expression is value-inactive if no value subexpression contains a split expression. To distinguish the reduction relations, we prime all deterministic reductions.

Theorem 3. *Suppose that M is a value-inactive, closed expression, $\varsigma \vdash M \succeq_{\emptyset} M'$, and $\varsigma, M' \longrightarrow' \varsigma', N'$. Then there exist N and ς' such that $\varsigma' \vdash N \succeq_{\emptyset} N'$ and $\varsigma, M \longrightarrow^{*} \varsigma', N$.*

6. Technical Results

In the literature, contract soundness typically states that applying a contract to an expression forces this expression to behave according to the contract. We augment this theorem with a dual context part that states that contexts ending in contract monitoring respect the contract that is monitored.

Theorem 4 (Contract soundness for expressions)**.**

$$M \mathbin{@^{b}} C \in [\![C]\!]^{+}.$$

Theorem 5 (Contract soundness for contexts)**.**

$$\mathcal{L}[\square \mathbin{@^{b}} C] \in [\![C]\!]^{-}.$$

However, such a soundness statement can be satisfied by a trivial interpretation of contract assertion that does not terminate or that throws some exception. Hence, we set out to prove a stronger result. If we assert a contract to an expression that is known to satisfy the contract, then no context should be able to elicit blame.

We need some notation to state this result. We write $\mathit{BLab}(X)$ for the set of blame labels occurring in syntactic phrases X like expressions, contracts, and contexts. We write $\mathit{dom}(\varsigma)$ for the set of blame identifiers that occur on the left side of a constraint in ς: $\mathit{dom}(\varsigma) = \{b \mid b◀(\dots) \in \varsigma\}$. This set is the largest set of blame identifiers b that may be defined in the least solution $\mathit{LSol}(\varsigma)$. That is, for $x \in \{\mathit{subject}, \mathit{context}\}$, if $\mathit{LSol}(\varsigma)(b, x) \sqsupseteq \mathsf{f}$, then $b \in \mathit{dom}(\varsigma)$.

Theorem 6 (Subject blame soundness)**.** *Suppose that* $M \in [\![C]\!]^{+}$. *For all* ς, E *with* $b \notin \mathit{dom}(\varsigma) \cup \mathit{BLab}(M, C, E)$, ς', *and* N *such that* $\varsigma, E[M \mathbin{@^{b}} C] \longmapsto^{*} \varsigma', N$, *it holds* $[\![\varsigma']\!](b, \mathit{subject}) \sqsubseteq \mathsf{t}$.

Theorem 7 (Context blame soundness)**.** *Suppose that* $\mathcal{L} \in [\![C]\!]^{-}$. *For all* ς, M *with* $b \notin \mathit{dom}(\varsigma) \cup \mathit{BLab}(M, C, E)$, ς', *and* N *such that* $\varsigma, \mathcal{L}[M \mathbin{@^{b}} C] \longmapsto^{*} \varsigma', N$, *it holds* $[\![\varsigma']\!](b, \mathit{context}) \sqsubseteq \mathsf{t}$.

We are interested in the contraposition of these two theorems: If reduction reaches a blame state for a subject, then there is a value violating the corresponding contract; and dually, if reduction reaches a blame state for a context, then there is indeed a context violating its contract.

In the companion technical report [21] we show that λ_V^{Con} is a conservative extension of the original blame calculus [13]. If we restrict the contract language of λ_V^{Con} to flat contracts and function contracts, then we can map programs in this restricted language to Findler and Felleisen's calculus and establish a bisimulation between executions in the two calculi. The actual statement of the theorem is somewhat technical.

7. Related Work

Higher-Order Contracts Software contracts evolved from Floyd and Hoare's work on using pre- and postconditions for program specification and verification [15, 19]. Meyer's *Design by Contract*™ methodology [22] stipulates the specification of contracts for all components of a program and introduces the idea of monitoring these contracts while the program is running.

Findler and Felleisen [13] were the first to construct contracts and contract monitors for higher-order functional languages. Their work has attracted a plethora of follow-up works that range from deliberations on blame assignment [8, 31] to extensions in various directions: contracts for polymorphic types [1, 17], for affine types [30], and for temporal conditions [9].

Semantics of Contracts Blume and McAllester [4] construct a semantics of contracts and show that it is sound and complete with respect to Findler-Felleisen style contract monitoring. Their definition of the set of expressions that satisfy a contract is superficially similar to ours, but there are some subtle differences that lead to considerable technical complexity in their work. The key difference is that their semantics does not have a counterpart to our notion of a context respecting a contract. This omission forces them to introduce a notion of safe expressions to exclude expressions that do not respect their context. They do consider dependent function contracts, the study of which we defer to future work.

Findler and Blume [11] model the semantics of higher-order, non-dependent contracts using pairs of error projections. The semantics is shown sound with respect to the Blume-McAllester model and completeness holds for non-empty contracts. Unfortunately, a projection-based semantics cannot be extended to dependent function contracts [12].

Dimoulas and Felleisen [7] investigate different styles of contract monitoring, ranging from tight monitoring to shy monitoring. Their base language is CPCF, a simply typed lambda calculus with higher-order (dependent) contracts and monitoring. Instead of providing a denotational semantics of contracts (as we do), they base their work on contextual equivalence and contextual simulation. While a set of contract-abiding terms could be derived from their definitions, the thus defined semantics would be defined in terms of monitoring, whereas our semantics is defined without recourse to monitoring. On the other hand, this choice enables the authors to relate different styles of contract monitoring and to clarify blame assignment by splitting contracts in a server (subject) and client (context) part, which compose back to the original contract. They do not investigate further operators on contracts.

Combinations of Contracts Racket's contract system [14, Chapter 8.1] supports the operators `and/c` and `or/c` on contracts. They are designed to extend their obvious action on flat contracts as conjunction and disjunction in a practically useful way to higher-order contracts. However, they are significantly different from intersection and union, so our proposal may be a useful complement.

The contract `(and/c` $C \dots$`)` "... tests any value by applying the contracts in order, from left to right." Thus, a contract like `(and/c (-> number? number?) (-> string? string?))` always raises context blame because no argument can be a number and a string at the same time. In contrast, the intersection contract $(\mathit{Num} \to \mathit{Num}) \cap (\mathit{Str} \to \mathit{Str})$ enables a context to choose between a number and a string argument.

The documentation of `(or/c` $C \dots$`)` is quite involved with many operational details. Essentially, the flat contracts among $C \dots$ are checked in order. If one of them succeeds, then the disjunction succeeds. Otherwise, the first-order parts of the remaining contracts are checked in order. The disjunction fails unless exactly one contract remains: in that case, the checked value is wrapped in the remaining contract.

Compared to intersection or union, `or/c` does not handle arbitrary combinations of flat and function contracts. It is not possible to construct the `or/c` of two function contracts of the same arity because such functions cannot be told apart by a first-order check.

Racket's `case->` operator [14, Chapter 8.2] essentially provides an arity-indexed function contract. Hence, the arity of each sub-contract must be different and, when asserted, a first-order arity check suffices to select one of the sub-contracts. Then the function gets wrapped as usual. This functionality is very specific to Racket and it is not clear whether it could be modeled with intersection and/or union.

In summary, Racket's contract system supports operators inspired by disjunction and conjunction. Their specification is operational and their properties are designed to fit the needs of a practical programmer. In contrast, our proposal for intersection and union contracts has a denotational specification grounded in type theory and our operators inherit the properties of the type-theoretic constructs.

The rewriting-based approach to check higher-order contracts symbolically [29] also supports contract operators in the spirit or `and/c` and `or/c`. This approach is designed to fit in with Racket's contract implementation and has similar restrictions.

8. Conclusion

Our calculus of blame assignment for higher-oder contracts with intersection and union contracts has a number of novel aspects. First, the specification for intersection and union contracts is strongly inspired by their type-theoretic counterparts. This connection tightly integrates statically and dynamically typed worlds which may be beneficial for future integration in a gradual type system.

Second, our development is based on a novel denotational semantics of contracts. It distinguishes a set of terms, subjects, that satisfy a contract and a set of contexts that respect the contract. Our monitoring soundness result proves that terms from the former set can never lead to subject (positive) blame whereas contexts from the latter set can never lead to context (negative) blame.

Acknowledgments

This work benefited from discussion with participants of the Dagstuhl Seminar "Scripting Languages and Frameworks: Analysis and Verification" in 2014: Christos Dimoulas, Matthias Felleisen, Cormac Flanagan, Fritz Henglein, and Sam Tobin-Hochstadt. In particular, Christos provided a flood of examples and untiring enthusiasm to discuss the semantics of contracts. Thanks are also due to Robby Findler, Phil Wadler, and the anonymous PLDI 2015 reviewers for their thoughtful remarks and (counter-) examples.

References

[1] A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In T. Ball and M. Sagiv, editors, *Proc. 38th ACM Symp. POPL*, pages 201–214, Austin, TX, USA, Jan. 2011. ACM Press.

[2] F. Barbanera and M. Dezani-Ciancaglini. Intersection and union types. In T. Ito and A. Meyer, editors, *Proc. Theoretical Aspects of Computer Software*, volume 526 of *LNCS*, Sendai, Japan, 1991. Springer.

[3] F. Barbanera, M. Dezani-Ciancaglini, and U. de' Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

[4] M. Blume and D. McAllester. Sound and complete models of contracts. *J. Funct. Program.*, 16:375–414, July 2006.

[5] M. Coppo and M. Dezani-Ciancaglini. A new type-assignment for λ-terms. *Archiv. Math. Logik*, 19(139-156), 1978.

[6] R. Davies and F. Pfenning. Intersection types and computational effects. In P. Wadler, editor, *Proc. ICFP 2000*, pages 198–208, Montreal, Canada, Sept. 2000. ACM Press, New York.

[7] C. Dimoulas and M. Felleisen. On contract satisfaction in a higher-order world. *ACM TOPLAS*, 33(5):16, 2011.

[8] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: No more scapegoating. In T. Ball and M. Sagiv, editors, *Proc. 38th ACM Symp. POPL*, pages 215–226, Austin, TX, USA, Jan. 2011. ACM Press.

[9] T. Disney, C. Flanagan, and J. McCarthy. Temporal higher-order contracts. In O. Danvy, editor, *Proc. ICFP 2011*, pages 176–188, Tokyo, Japan, Sept. 2011. ACM Press, New York.

[10] J. Dunfield and F. Pfenning. Type assignment for intersections and unions in call-by-value languages. In A. D. Gordon, editor, *FOSSACS 2003*, volume 2620 of *LNCS*, pages 250–266. Springer, 2003.

[11] R. B. Findler and M. Blume. Contracts as pairs of projections. In P. Wadler and M. Hagiya, editors, *Proceedings of the 8th International Symposium on Functional and Logic Programming FLOPS 2006*, pages 226–241, Fuji Susono, Japan, Apr. 2006. Springer.

[12] R. B. Findler, M. Blume, and M. Felleisen. An investigation of contracts as projections. Technical Report TR-2004-02, University of Chicago, Computer Science Department, 2004.

[13] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In S. Peyton-Jones, editor, *Proc. ICFP 2002*, pages 48–59, Pittsburgh, PA, USA, Oct. 2002. ACM Press, New York.

[14] M. Flatt and PLT. *The Racket Reference*, v.6.1.1 edition, 2015. `http://docs.racket-lang.org/reference/index.html`.

[15] R. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, pages 19–32, 1967.

[16] M. Furr, J. An, J. S. Foster, and M. W. Hicks. Static type inference for Ruby. In S. Y. Shin and S. Ossowski, editors, *SAC*, pages 1859–1866, Honolulu, Hawaii, USA, Mar. 2009. ACM.

[17] A. Guha, J. Matthews, R. B. Findler, and S. Krishnamurthi. Relationally-parametric polymorphic contracts. In P. Costanza and R. Hirschfeld, editors, *DLS*, pages 29–40. ACM, 2007.

[18] R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In P. Wadler and M. Hagiya, editors, *Proceedings of the 8th International Symposium on Functional and Logic Programming FLOPS 2006*, pages 208–225, Fuji Susono, Japan, Apr. 2006. Springer.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12:576–580, 1969.

[20] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM TOPLAS*, 27(1), 2004.

[21] M. Keil and P. Thiemann. Blame assignment for higher-order contracts with intersection and union. Technical report, Institute for Computer Science, University of Freiburg, 2015.

[22] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[23] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Upper Saddle River, NJ, USA, 2nd edition, 1997.

[24] B. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, Feb. 1991.

[25] B. C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, Dec. 1991.

[26] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Sept. 2006.

[27] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium, DLS 2006*, pages 964–974, Portland, Oregon, USA, 2006. ACM.

[28] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In P. Wadler, editor, *Proc. 35th ACM Symp. POPL*, pages 395–406, San Francisco, CA, USA, Jan. 2008. ACM Press.

[29] S. Tobin-Hochstadt and D. Van Horn. Higher-order symbolic execution via contracts. In G. T. Leavens and M. B. Dwyer, editors, *OOPSLA*, pages 537–554. ACM, 2012.

[30] J. A. Tov and R. Pucella. Stateful contracts for affine types. In A. D. Gordon, editor, *ESOP 2010*, volume 6012 of *LNCS*, pages 550–569. Springer, 2010.

[31] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proc. 18th ESOP*, volume 5502 of *LNCS*, pages 1–16, York, UK, Mar. 2009. Springer.

[32] A. K. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM TOPLAS*, 19(1):87–152, Jan. 1997.