Static Contract Simplification

Technical Report

Matthias Keil   Peter Thiemann

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

Abstract

Contracts and contract monitoring are a powerful mechanism for specifying properties and guaranteeing them at run time. However, run time monitoring of contracts imposes a significant overhead. The execution time is impacted by the insertion of contract checks as well as by the introduction of proxy objects that perform delayed contract checks on demand.

Static contract simplification attacks this issue using program transformation. It applies compile-time transformations to programs with contracts to reduce the overall run time while preserving the original behavior. Our key technique is to statically propagate contracts through the program and to evaluate and merge contracts where possible. The goal is to obtain residual contracts that are collectively cheaper to check at run time.

We distinguish different levels of preservation of behavior, which impose different limitations on the admissible transformations: Strong blame preservation, where the transformation is a behavioral equivalence, and weak blame preservation, where the transformed program is equivalent up to the particular violation reported. Our transformations never increase the overall number of contract checks.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics; D.2.4 [*Software/Program Verification*]: Programming by contract

Keywords Higher-Order Contracts, Contract Simplification, Hybrid Contract Checking

1. Introduction

Software contracts (Meyer 1988) stipulate invariants for objects as well as pre- and postconditions for functions a programmer regards as essential for the correct execution of a program. Traditionally, contracts are checked at run time by contract monitoring. This approach has become a prominent mechanism to provide strong guarantees for programs in dynamically typed languages while preserving their flexibility and expressiveness.

Using formal and precise specifications in the form of contracts in this way sounds appealing, but it comes with a cost: Contract monitoring degrades the execution time significantly (Takikawa et al. 2016). This cost arises because every contract extends a program with additional code to check the contract assertions that needs to be executed on top of the program. Moreover, human developers may inadvertently add contracts to frequently used functions and objects so that contracts end up on hot paths in a program. In particular, if contracts are naively applied to recursive functions, predicates will check the same values repeatedly, which aggravates the performance degradation.

Existing contract systems for dynamically typed languages (e.g., Racket's contract framework (Flatt et al. 2014, Chapter 7), Disney's JavaScript contract system *contracts.js* (Disney 2013), or TreatJS for JavaScript (Keil and Thiemann 2015b)) each suffer from a considerable performance impact when extending a program with contracts.

In contrast, static contract checking (Xu et al. 2009) avoids any run-time cost if it succeeds in proving and removing a contract at compile time. Otherwise, it defers to plain contract checking. A subsequent extension to hybrid contract checking (Xu 2012) is able to exploit partial proofs and to extract a residual contract that still needs to be checked at run time. Both of these works are phrased as transformations in a typed intermediate language.

A similar motivation is behind static contract verification (Nguyen et al. 2014). This work takes place in the context of an untyped language. It applies symbolic execution as well as techniques from occurrence typing to prove contracts statically and thus eliminate their run-time overhead.

Our work is complementary to these previous efforts. Our main objective is to improve the efficiency of contract monitoring by using static techniques to evaluate as much of a contract as possible, to detect and remove redundant checking of contracts, and to reassemble the fragments that cannot be discharged statically in places where they are least likely to affect performance. Typically, we move contract fragments to a surrounding module boundary and compose them to a new contract.

Thus, our goal is not to detect contract violations at compile time, but rather to optimize contracts based on static information. We also do not rewrite the non-contract code of the underlying program, but perform program transformations restricted to just moving around contracts.

To demonstrate the essence of our approach, consider the following code snippet that contains a function with an intersection contract applied to it.

```
let addOne =
((λ plus (λ z ((plus 1) z)))
 [(λ x (λ y (+ x y)))
  @ ((Number? → (Number? → Number?)) ∩
      (String? → (String? → String?)))])
```

The value *addOne* is a function that takes an argument *z* and returns the result of applying function *plus* to the number *1* and to *z*. We use the inline operator @ to indicate contract assertions that we put in square brackets to improve readability. We further rely on predefined flat contracts *Number?* and *String?* that are type checks for numbers and strings, respectively.

If we assume that the + operator is overloaded as in JavaScript or Java so that it works for number and string values (but possibly fails or delivers meaningless results for other inputs), a programmer may add an intersection contract to *plus*. This contract has the effect that the context of a use of *plus* (here function *addOne*) can choose to use *plus* either with number or string arguments. Monitoring this contract requires six predicate checks at every use of *addOne*.[1]

However, as the second argument of *plus* is already a number, the right side of the intersection can never be fulfilled. So, it would be sufficient to check if the first argument of *plus* and its return value satisfy *Number?*. Moreover, the remaining predicates can be lifted to an interface description on *addOne*, as the following example demonstrates. Only two predicate checks remain.

```
let addOne =
[((λ plus (λ z ((plus 1) z)))
  ((λ x (λ y (+ x y))))) @ (Number? → Number?)]
```

This simplification can be done without knowing whether *addOne* is ever executed: we are not losing any blame due to the transformation. However, it would not be ok to lift argument contracts across function boundaries. Such lifting would introduce checks that may never happen in the original program.

Building on the formalization of Keil and Thiemann (2015a), we present a transformation system for static contract simplification. Our system provides two stages of simplification with different consequences: The *Baseline Optimization* and the *Subset Optimization*.

The *Baseline Optimization* unfolds contracts and evaluates predicates where possible while preserving the blame behavior of a program. The *Subset Optimization* reorganizes contracts and forms new contracts from the contract fragments, but it may reorder contract computations and thus change the blame (i.e., the particular violation that is flagged at run time). Our simplification rules respect three overall guidelines:

Improvement Each transformation step improves the efficiency of contract monitoring: it either *reduces* or *maintains* the total number of predicate checks at run time. Simplification never increases the number of run-time checks.

Strong Blame-Preservation Each transformation step of the baseline simplification fully preserves the blame-behavior of a program. It maintains the oder of contract checks at run time and thus the order in which violations are reported. A transformed program produces the same outcome as the original program.

Weak Blame-Preservation Transformation steps of the subset simplification may change the oder of predicate checks, so that they may also change the order of observed violations. Thus, a transformed program ends in a blame state if and only if the original program would end in a blame state. However, the transformed program may report a different violation.

An optional third level optimizes contracts across module boundaries. Even though it provides the strongest optimization, it may over-approximate contract violations and it would violate all three guideline from above.

Contributions This work makes the following contributions:

- We present the semantics of a two-tier static contract simplification that reorganizes and pre-evaluates contracts at compile time while guaranteeing weak or strong blame preservation.
- We show how to split alternatives into separated observation to deal with intersection and union contracts.
- We give a specification of subcontracting that is used to simplify contracts based on the knowledge of other asserted contracts.

[1] Assuming the semantics of intersection and union contracts from the literature (Keil and Thiemann 2015a).

We created an executable implementation of the semantics using PLT Redex (Felleisen et al. 2009). The implementation is available online at `http://anonymous`.

Overview Section 2 provides a series of examples demonstrating the essence of our contract simplification. Section 3 recalls contracts and contract satisfaction from the literature and introduces an untyped, applied, call-by-value lambda calculus with contract monitoring. Section 4 introduces *Baseline Simplification*, the first tier in our simplification stack. Section 5 shows the second tier, *Subset Simplification*, which also handles alternatives and reduces contracts that are subsumed by other contracts. Section 6 states our blame theorems and Section 7 briefly addresses run time improvements. Section 8 discusses related work and Section 9 concludes.

The appendix extends this paper with auxiliary functions and definitions, further examples, and proofs of all theorems. In particular, it provides the semantics to enable picky-like evaluation semantics and the semantics of a third, approximating, simplification level that provides the stringes simplification.

2. Contract Simplification

This section explains the main ideas of our transformation through a series of examples. We start with a simple example and work up to more complex ones.

2.1 Unrolling Delayed Contracts

The examples consider different contracts on the *addOne* example from the previous section. In a first step we add a simple function contract to *plus*.

```
let addOne =
((λ plus (λ z ((plus 1) z)))
 [(λ x (λ y (+ x y)))
  @ (Number? → (Number? → Number?))])
```

While *Number?* is a flat contract that is checked immediately, function contracts are *delayed* contracts. A delayed contract cannot be checked directly when asserted to a value. Delayed contracts stay attached to a value and are checked when the value is used. Thus, every call to *addOne* triggers three predicate checks.

Instead of asserting a contract that forces a number of delayed contract checks, our *Baseline Simplification* unrolls a delayed contract to all uses of that value. This step pushes the delayed contract inwards to all occurrences of *plus* in *addOne*.

```
let addOne =
((λ plus
  (λ z ([plus @ (Number? → (Number? → Number?))] 1) z)))
  (λ x (λ y (+ x y))))
```

This step performs no simplification, but it yields an equivalent program that is better prepared for unfolding the function contract to its domain and range because the contract is on an expression in application position. The following snippet demonstrates the (complete) unfolding of the contract, which combines a number of primitive unfolding steps.

```
let addOne =
((λ plus
  (λ z [((plus [1 @ Number?]) [z @ Number?]) @ Number?]))
  (λ x (λ y (+ x y))))
```

In the resulting expression, the function contracts are completely decomposed into flat contract components which are spread all over the expression.

Observe that, after unfolding, there are several flat contracts applied to values. Such contracts can be checked statically without changing the outcome. For example, *1* satisfies *Number?*. All satisfied contracts can safely be removed from the program.

```
let addOne =
((λ plus
 (λ z [((plus 1) [z @ Number?]) @ Number?]))
 (λ x (λ y (+ x y))))
```

In a last step, we push the remaining contract fragments outwards where possible. This transformation creates a new function contract from that contract. The special contract $\top$ accepts every value.

```
let addOne =
[((λ plus
  (λ z ((plus 1) [z @ Number?])))
  (λ x (λ y (+ x y)))) @ (⊤ → Number?)]
```

Afterwards, the outermost expression carries a function contract that checks the return value of *addOne* and one contract is left in the body of *addOne* to check the argument. Again, this step does not simplify contracts, but it prepares for further simplification steps.

However, further simplification cannot be done while guaranteeing strong blame preservation. The *Baseline Simplification* is not allowed to behave differently than the dynamics would do at run time.

2.2 Treating Intersection and Union

Our next example considers the full *addOne* example from the introduction, which we repeat for convenience.

```
let addOne =
((λ plus (λ z ((plus 1) z)))
 [(λ x (λ y (+ x y)))
  @ ((Number? → (Number? → Number?)) ∩
       (String? → (String? → String?)))])
```

Intersections of function contracts are *delayed* contracts, they must stay attached to a value until the value is used in an application. Moreover, for an intersection, the context can choose to fulfill the left or the right side of the intersection. Thus, every call to *addOne* triggers six predicate checks.

Again, *Baseline Simplification* unrolls the intersection contract to all uses of that value and unfolds the contract to its domain and range when used in an application. The following snippet demonstrates the unfolding of the intersection contract.

```
let addOne =
((λ plus
 (λ z [[((plus
          [[1 @_2 String?] @_1 Number?])
         [[z @_2 String?] @_1 Number?]) @_2 String?] @_1 Number
       ?]))
 (λ x (λ y (+ x y))))
```

In addition, the static contract monitor maintains additional information in the background (e.g., which contracts belong to the same side of the intersection) to connect each contract with the enclosing operation. In the example we indicate the originating operand of the intersection by $@_1$ and $@_2$.

After unfolding, the *Baseline Simplification* checks flat contracts applied to values. Here, *1* satisfies *Number?* but violates *String?*. While *Number?* can now be removed, information about a failing contract must remain in the source program. In general, it is not possible to report failure statically because we do not know if the code containing the failed contract (in this case, function *addOne*) is ever executed.

However, we do not have to leave the original contract in the expression. Instead, the transformation replaces it by a "false contract" $\bot$, whose sole duty is to remember a contract violation and report it at run time. The next snippet shows the result after processing satisfied and failing contracts and pushing the remaining fragments outwards.

```
let addOne =
[[((λ plus
  (λ z ((plus
        [1 @_2 ⊥])
       [[z @_2 String?] @_1 Number?])))
  (λ x (λ y (+ x y)))) @_1 (⊤ → Number?)] @_2 (⊤ → String?)]
```

Afterwards, the outermost expression carries two function contracts that check the return value of *addOne*; two contracts are left in the body of *addOne* to check the argument.

2.3 Splitting Alternatives in Separated Observations

When remembering the last state in Section 2.2, we still check *String?* even though we know that right side of the intersection is never fulfilled. This is because the *Baseline Simplification* preserves the blame behaviour of a program and is thus not able to deal with alternatives. However, the next *Subset Simplification* abandons the strong blame preservation to further optimize contracts.

Instead of unfolding intersection and union contracts, we now observe both alternatives in separation. After transforming both alternatives, we join the remaining contracts or even discard a whole branch if it definitely leads to a violation.

In our example, we see that *1 @ String?* leads to a context failure of the right operand of the intersection contract. Thus, it is useless to keep the other fragments of the right operand which definitely yields a context violation. The following code snippet shows the result.

```
let addOne =
[((λ plus
  (λ z ((plus [1 @_2 ⊥]) [z @_1 Number?])))
  (λ x (λ y (+ x y)))) @_1 (⊤ → Number?)]
```

It remains to check *z* and *addOne*'s return to be a number. This are two remaining checks per use of *addOne*.

However, we still need to keep the information on the failing alternative. This is required to throw an exception if also the other alternative fails.

2.4 Lifting Contracts

As shown in the example above, contracts on the function's body are reassembled to a new function contract, whereas contracts on function arguments must remain to preserve the order of predicate checks. Continuing the example, our next step constructs a new contract from the remaining contracts on arguments.

```
let addOne =
[[((λ plus
  (λ z ((plus [1 @_2 ⊥]) z)))
  (λ x (λ y (+ x y)))) @_1 (Number? → ⊤)] @_1 (⊤ → Number?)]
```

The *addOne* function now contains another function contract that restricts its domain to number values. As this step puts contracts on arguments in front, it may change the order in which violations arise. This lifting to module boundaries only reorganizes existing contracts, but it helps to reduce contracts that are subsumed by other contracts.

2.5 Condensing Contracts

Continuing the example in Section 2.4, the outermost expression carries two function contracts that check the argument and the return value of *addOne*. Moreover, both contracts arise from the same source contract. These function contracts may be condensed to a single function contract on *addOne*.

```
let addOne =
[((λ plus
  (λ z ((plus [1 @₂ ⊥]) z)))
  (λ x (λ y (+ x y)))) @₁ (Number? → Number?)]
```

This step is only possible if the argument contract, here *Number ?* $\to \top$, is in negative position and if the return contract, $\top \to$ *Number?* is in positive position.

2.6 Contract Subsets

Splitting intersections and unions into separate observations has another important advantage: We know that in every branch every contract must be fulfilled. Thus we can relate contracts to other contracts and use knowledge about other contracts to reduce redundant checks. To demonstrate an example, we add a second contract to our running example.

```
let addOne =
((λ plus [(λ z ((plus 1) z)) @ (Positive? → Positive?)])
 [(λ x (λ y (+ x y)))
  @ ((Number? → (Number? → Number?)) ∩
       (String? → (String? → String?)))])
```

Here, *Positive?* is a flat contract that checks for positive numbers. In addition to the contract on *plus*, *addOne* contains another function contract, requiring *addOne* to be called with positive numbers and to return a positive number. After performing the same transformation steps as before, we obtain the following expressions.

```
let addOne =
[[[((λ plus (λ z ((plus 1) z)))
   (λ x (λ y (+ x y)))) @₁ (Number? → ⊤)] @₁ (⊤ → Number?)]
       @ (Positive? → Positive?)]
||
let addOne =
[((λ plus (λ z ((plus [1 @₂ ⊥]) z)))
 (λ x (λ y (+ x y)))) @ (Positive? → Positive?)]
```

The intersection is split into two parallel observations, as indicated by $\|$. All contracts were pushed to the outermost boundary and form a new module boundary. However, some of the contract checks are redundant.

For example, the innermost contract in the first observation requires *addOne* to be called with numbers, whereas the outermost contract restricts the argument already to positive numbers. As positive numbers are a proper subset of numbers, the inner contract will never raise a violation if the outer contract is satisfied. Thus, it can be removed without changing the blame behavior of a program.

The contract *Positive?* $\to$ *Positive?* is more restrictive than any other contract on *addOne*, that is, *Number?* $\to \top$ and $\top \to$ *Number?*. Here, a contract A is more restrictive than a contract B, if the satisfaction of A implies the satisfaction of B.

However, if a more restrictive contract fails, that failure does not imply failure of the less restrictive contract. But as every contract must be fulfilled, the program already stops with a contract violation. Thus, there is no need to check a less restrictive contract on that branch. Unfortunately, removing the less restrictive contract might change the order of observed violations, as the less restrictive contract might report its violation first.

To sum up, in the first observation the outer contract remains as it subsumes any other contract, whereas in the second observation both contracts remain.

Finally, after finishing all transformation steps, the simplifications joins the observation to final expression.

```
let addOne =
[((λ plus
 (λ z ((plus [1 @₂ ⊥]) z)))
 (λ x (λ y (+ x y)))) @ (Positive? → Positive?)]
```

$$
\begin{array}{lcl}
L, M, N & ::= & K \mid x \mid \lambda x.M \mid M\,N \mid op(\vec{M}) \mid \mathit{if}\,L\,M\,N \\
K & ::= & \mathit{false} \mid \mathit{true} \mid 0 \mid 1 \mid \ldots \\
U, V, W & ::= & K \mid \lambda x.M \\
\mathcal{E} & ::= & \square \mid \mathcal{E}\,N \mid V\,\mathcal{E} \mid op(\vec{V}\,\mathcal{E}\,\vec{M}) \mid \mathit{if}\,\mathcal{E}\,M\,N
\end{array}
$$

$$
\begin{array}{llcll}
\textsc{Beta} & \mathcal{E}[\![\lambda x.M\,V]\!] & \longrightarrow & \mathcal{E}[\![M[x \mapsto V]]\!] & \\
\textsc{Op} & \mathcal{E}[\![op(\vec{V})]\!] & \longrightarrow & \mathcal{E}[\![\delta_{op}(\vec{V})]\!] & \vec{V} \in dom(\delta_{op}) \\
\textsc{True} & \mathcal{E}[\![\mathit{if\,true}\,M\,N]\!] & \longrightarrow & \mathcal{E}[\![M]\!] & \\
\textsc{False} & \mathcal{E}[\![\mathit{if\,false}\,M\,N]\!] & \longrightarrow & \mathcal{E}[\![N]\!] &
\end{array}
$$

Figure 1. Syntax and semantics of λ_{J}.

$$
\begin{array}{lcl}
C, D & ::= & I \mid Q \mid C \cup D \mid I \cap C \\
I, J & ::= & \mathit{flat}(M) \\
Q, R & ::= & C \to D \mid x \to A \mid Q \cap R \\
A & ::= & \Lambda x.C
\end{array}
$$

$$
\begin{array}{lcl}
B & ::= & \mathit{+blame} \mid \mathit{-blame} \\
L, M, N & += & M @^{\flat} C \mid\mid V @^{\flat}_{\iota} (\mathit{eval}\,M) \mid B^{\flat} \\
U, V, W & += & \mid\mid V @^{\flat}_{\iota} Q \\
\mathcal{E} & += & \mathcal{E} @^{\flat} C \mid\mid V @^{\flat}_{\iota} (\mathit{eval}\,\mathcal{E})
\end{array}
$$

$$
\begin{array}{lcl}
b & ::= & \flat \mid\mid \iota \\
\kappa & ::= & b \blacktriangleleft W \mid b \blacktriangleleft \iota \mid b \blacktriangleleft \iota \to \iota \mid b \blacktriangleleft \iota \cap \iota \mid b \blacktriangleleft \iota \cup \iota \\
\varsigma & ::= & \cdot \mid \kappa : \varsigma
\end{array}
$$

Figure 2. Syntax extension for λ_{Con}.

3. Contracts and Contract Satisfaction

This section defines λ_{Con}, an untyped call-by-value lambda calculus with contracts which serves as a core calculus for contract monitoring. It first introduces the base calculus λ_{J}, then it proceeds to describe contracts and their semantics for the base calculus. Finally it gives the semantics of contract assertion and blame propagation.

The core calculus is inspired by previous work from the literature (Keil and Thiemann 2015a).

3.1 The Base Language λ_{J}

Figure 1 defines syntax and semantics of λ_{J}, an applied call-by-value lambda calculus. A term M is either a first-order constant, a variable, a lambda abstraction, an application, a primitive operation, or a condition. Variables x, y, and z are drawn from a denumerable set of variables. Constants K range over a set of base type values including booleans, numbers, and strings.

To define evaluation, values U, V, and W range over values and $\mathcal{E}$ over evaluation contexts, which are defined as usual.

The small-step reduction relation $\longrightarrow$ comprises beta-value reduction, built-in partial operations transforming a vector of values into a value, and the reduction of conditionals. 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 $M \longrightarrow N$.

3.2 Contracts and Contracted λ_{J}

Figure 2 defines the syntax of λ_{Con} as an extension of λ_{J}. It first introduces constructs for contract monitoring in general and, second, it adds new terms specific to contract monitoring. Intermediate terms that do not occur in source programs appear after double bars "||".

A contract C is either an immediate contact I, a delayed contract Q, a union between two contract $(C \cup D)$, or an intersection $(I \cap C)$.

Without loss of generality we restrict top-level intersection contracts to intersections between an immediate contract and a rest contract (cf. (Keil and Thiemann 2015a)). This requires that unions and immediate parts are distributivity pulled out of intersections of delayed contracts such that the immediate parts can be check right away when asserted, whereas the delayed parts remain.

Immediate contracts I and J stand for flat contracts define by a predicate M that can be evaluated right away when asserted to a value.

A delayed contracts Q is either a function contract $C \rightarrow D$ with domain contract D and range contract D, a dependent contract $x \rightarrow A$ defined by contract abstraction A, or the finite intersection of function (dependent) contracts. Delayed contracts stay with the value until the value is subject of an application.

Consequently, values are extended with $V @_{\iota}^{\flat} Q$ which represents a value wrapped in a delayed contract that is to be monitored when the value is used. The extended set of values forces us to revisit the built-in operations. We posit that each partial function δ_{op} first erases all contract monitoring from its arguments, then processes the underlying λ_{J}-values, and finally returns a λ_{J}-value.

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

In the intermediate term $V @_{\iota}^{\flat} (\mathit{eval}\, M)$, the term M represents the current evaluation state of the predicate of a flat contract. The $B^{\flat}$ expression signal a contract violation at label $\flat$. Term B is either a *+blame* (positive) or a *-blame* (negative) blame, representing subject or context blame, respectively.

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.

In λ_{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.

3.3 Semantics for λ_{Con}

Figure 3 specifies the small-step reduction semantics of λ_{Con} as a relation $\varsigma, M \longrightarrow \varsigma', N$ on pairs of a constraint list and a term. 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. Appendix A explains the semantics of the constraints.

The rule ASSERT introduces a fresh name for each new instantiation of a monitor in the source program. It is needed for technical reasons. Flat contracts get evaluated immediately. Rule FLAT starts evaluating a flat contract by evaluating the predicate M applied to the unwrapped subject value W. Meta function $\nabla(V)$ (Figure 4) first erases all contract monitors from the subject value V. After predicate evaluation, rule UNIT picks up the result and stores it in a constraint.

The second group of rules UNION and INTERSECTION implements contract checking of top-level union and intersection contracts. Both rules install a new constraint that combine the contract satisfaction of the subcontracts to the satisfaction of the current contract.

The rules D-FUNCTION, D-DEPENDENT, and D-INTERSECTION define the behavior of a contracted value under function application. Rule D-FUNCTION 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.

The D-DEPENDENT rule defines the evaluation of a dependent contract. It substitutes the function's arguments for every occurrence of x in the body of C and proceeds with the assertion of the resulting contract.

Rule D-INTERSECTION embodies the evaluation of an intersection contract under function application. The generated constraint serves to combine the results of the subcontracts. Unlike the union contract, an intersection constraint occurs at each use of the contracted value, which implements the choice of the context (cf. (Keil and Thiemann 2015a)).

Finally, λ_{J} reductions are lifted to λ_{Con} reductions using rule BASE. Built-in operations and conditions "see through" contracts (rules D-OP and D-IF) and proceed on λ_{J}-values without contracts. This also implies that all base type operations only return values that does not contain contracts.

3.4 Introducing Blame

The dynamics in Figure 3 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 associated with 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. Appendix A specifies the constraint satisfaction for λ_{Con}.

Computing a blame assignment boils down to computing an interpretation for a constraint list ς. To determine whether a constraint list ς is a blame state (i.e., whether it should signal a contract violation), we check whether the state ς maps any source-level blame label $\flat$ to *false*.

Definition 1. *ς is a* blame state for blame label $\flat$ *iff*

$$\exists \flat . \varsigma(\flat.\mathit{subject} \wedge \flat.\mathit{context}) \sqsupseteq \mathit{false}.$$

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 $\varsigma, M \Longrightarrow \varsigma', N$ 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 *B*.

$$\frac{\exists \flat . \varsigma(\flat.\mathit{subject}) \sqsupseteq \mathit{false}}{\varsigma, M \Longrightarrow \varsigma, \mathit{+blame}^{\flat}} \qquad \frac{\exists \flat . \varsigma(\flat.\mathit{context}) \sqsupseteq \mathit{false}}{\varsigma, M \Longrightarrow \varsigma, \mathit{-blame}^{\flat}}$$

$$\frac{\varsigma, M \longrightarrow \varsigma', N \qquad \nexists \flat . \varsigma(\flat.\mathit{subject} \wedge \flat.\mathit{context}) \sqsupseteq \mathit{false}}{\varsigma, M \Longrightarrow \varsigma', N}$$

3.5 Lax, Picky, and Indy Semantics

Contract monitoring distinguishes three blame semantics: *Lax*, *Picky*, and *Indy*, initially introduced to handle correctness and completeness of higher-order dependent contracts (Dimoulas et al.

$$
\begin{array}{llllll}
\textsc{Assert} & \varsigma, & \mathcal{E}\llbracket V @^{\flat} C \rrbracket & \longrightarrow & (\flat \blacktriangleleft \iota) : \varsigma, & \mathcal{E}\llbracket V @^{\flat}_{\iota} C \rrbracket & \iota \notin \varsigma \\
\\
\textsc{Flat} & \varsigma, & \mathcal{E}\llbracket V @^{\flat}_{\iota} \mathit{flat}(M) \rrbracket & \longrightarrow & \varsigma, & \mathcal{E}\llbracket V @^{\flat}_{\iota} (\mathit{eval}\,(M\,U)) \rrbracket & U = \nabla(V) \\
\textsc{Unit} & \varsigma, & \mathcal{E}\llbracket V @^{\flat}_{\iota} (\mathit{eval}\,W) \rrbracket & \longrightarrow & (\iota \blacktriangleleft W) : \varsigma, & \mathcal{E}\llbracket V \rrbracket & \\
\\
\textsc{Union} & \varsigma, & \mathcal{E}\llbracket V @^{\flat}_{\iota} (C \cup D) \rrbracket & \longrightarrow & (\iota \blacktriangleleft \iota_1 \cup \iota_2) : \varsigma, & \mathcal{E}\llbracket (V @^{\flat}_{\iota_1} C) @^{\flat}_{\iota_2} D \rrbracket & \iota_1, \iota_2 \notin \varsigma \\
\textsc{Intersection} & \varsigma, & \mathcal{E}\llbracket V @^{\flat}_{\iota} (I \cap C) \rrbracket & \longrightarrow & (\iota \blacktriangleleft \iota_1 \cap \iota_2) : \varsigma, & \mathcal{E}\llbracket (V @^{\flat}_{\iota_1} I) @^{\flat}_{\iota_2} C \rrbracket & \iota_1, \iota_2 \notin \varsigma \\
\\
\textsc{D-Function} & \varsigma, & \mathcal{E}\llbracket (V @^{\flat}_{\iota} (C \rightarrow D))\, W \rrbracket & \longrightarrow & (\iota \blacktriangleleft \iota_1 \rightarrow \iota_2) : \varsigma, & \mathcal{E}\llbracket (V\,(W @^{\flat}_{\iota_1} C)) @^{\flat}_{\iota_2} D \rrbracket & \iota_1, \iota_2 \notin \varsigma \\
\textsc{D-Dependent} & \varsigma, & \mathcal{E}\llbracket (V @^{\flat}_{\iota} (x \rightarrow \Lambda x.C))\, W \rrbracket & \longrightarrow & \varsigma, & \mathcal{E}\llbracket (V\,W) @^{\flat}_{\iota} (C[x \mapsto U]) \rrbracket & U = \nabla(W) \\
\textsc{D-Intersection} & \varsigma, & \mathcal{E}\llbracket (V @^{\flat}_{\iota} (Q \cap R))\, W \rrbracket & \longrightarrow & (\iota \blacktriangleleft \iota_1 \cap \iota_2) : \varsigma, & \mathcal{E}\llbracket ((V @^{\flat}_{\iota_1} Q) @^{\flat}_{\iota_2} R)\, W \rrbracket & \iota_1, \iota_2 \notin \varsigma
\end{array}
$$

$$
\textsc{Base}\quad \frac{M \longrightarrow N}{\varsigma, M \longrightarrow \varsigma, N}
\qquad
\textsc{D-Op}\quad \frac{U = K \mid \lambda x.M}{\varsigma, \mathcal{E}\llbracket op(\vec{U},\, (V @^{\flat}_{\iota} Q),\, \vec{W}) \rrbracket \longrightarrow \varsigma, \mathcal{E}\llbracket op(\vec{U},\, V,\, \vec{W}) \rrbracket}
\qquad
\textsc{D-If}\quad \varsigma, \mathcal{E}\llbracket \mathit{if}\,(V @^{\flat}_{\iota} Q)\, M\, N \rrbracket \longrightarrow \varsigma, \mathcal{E}\llbracket \mathit{if}\, V\, M\, N \rrbracket
$$

Figure 3. Operational semantics of λ_{Con}.

$$
\nabla(V) = \begin{cases} \nabla(W) & V = W @^{\flat}_{\iota} Q \\ V & \text{otherwise} \end{cases}
$$

Figure 4. Unwrap delayed contracts.

$$
\begin{array}{lll}
C, D & \mathrel{+}= & \top \mid \bot \\
L, M, N & \mathrel{+}= & M @^{\flat}_{\iota} C \\
\kappa & \mathrel{+}= & b \blacktriangleleft \neg\iota
\end{array}
$$

Figure 5. Syntax extension of λ_{Con}.

2011). The different styles address the point that a contract might violate another contract.

The *Lax* semantics erase all contract monitors on values before applying a value to a predicate. This is correct but not complete because it guarantees that a well-behaved program gets never blamed for a violation taking place in a predicate. However, it omits to report violations in predicates.

The *Picky* semantics, in contrast, is complete but not correct. As it retains contract monitors on the values it might wrongly blame the program for violating a contract, but it guarantees that every violation is reported.

The *Indy* semantics, an extension of *Picky*, introduces a third party (the contract itself) in addition to the exiting parties context and subject. Passing a value to a predicate reorganizes the monitor such that wrong uses of that value blame the ill-behaved contract.

Our optimization considers the *Lax* style, as it strongly guarantees correctness and completeness in respect to source program. As reorganizing contracts changes the order of predicate checks optimizing is simpler when using *Lax* semantics.

Appendix D discusses our simplification with respect to *Picky* like evaluation semantics.

4. Baseline Simplification

This section presents the *Baseline Simplification*, the first tier in our simplification stack. The baseline simplification encompass (i) the unfolding of contract assertions and the creation of constraints where possible, (ii) the compile-time evaluation of flat contracts (predicates) on values, and (iii) the unrolling of delayed contracts to all uses of the contracted values. To this end, we introduce a set of canonical (non-transformable) terms T as a subset of λ_{Con} terms M and extend syntax and semantics of λ_{Con} in some respects.

All baseline steps do exactly the same as the dynamics would later on do. Thus, baseline simplification guarantees *Strong Blame-preservation* and *Improvement* for all transformation steps.

4.1 Syntax and Semantics Extension

Figure 5 shows some syntax extension for λ_{Con}.

Contracts now contain $\top$ (true) and $\bot$ (false), shortcuts for predicates evaluating to true or false, respectively. $\top$ and $\bot$ arise during optimization and represent remaining knowledge about the given proposition. Whereas $\top$ can be reduced statically, $\bot$ must remain in the source program to ensure correct blame propagation.

The shortcuts are more efficient because none of them must be checked at run time. When reaching a term with $\top$ or $\bot$ the outcome can immediately be transfered into a constraint and must not be checked like a predicate.

Source terms M now contain partially evaluated contracts on source terms $M @^{\flat}_{\iota} C$ and a new inversion constraint $b \blacktriangleleft \neg\iota$ extends the set of constrains κ. The inversion constraint flips context and subject responsibility and corresponds to the negative side of a function constraint. This constraint only arises during optimization.

Figure 6 specified the reduction of the shortcuts. Rule Reduce/True and Reduce/False creates a constraint for $\top$ and $\bot$. Three special rules, D-Domain, D-Range, and D-False simplify the evaluation of function contracts with $\top$ and $\bot$ on its domain or range portion. If one portion is true only the other portion must be checked. All this function contrast arise only during optimization.

4.2 Canonical Terms and Contexts

Canonical (non-transformable) terms are the output of our transformation. A canonical terms is either a contract-free term or a term with a contract that cannot further transformed at this level.

In general, canonical terms distinguish terms without a contract on the outermost level (indicated by letter S) from terms with a non-transformable contract (indicated by letter T). Canonical terms are a subset of λ_{Con} terms M. Appendix B.1 shows its definition.

A transformation context $\mathcal{F}$, $\mathcal{G}$, and $\mathcal{H}$ (Figure 7) is defined as usual as a term with a hole and a value context $\mathcal{V}$ is a term with an indefinite number of remaining $\bot$ contracts. This terms result only during transformation.

$$
\begin{array}{llllll}
\textsc{Reduce/True} & \varsigma, \ \mathcal{E}[\![V @^\flat_\iota \top]\!] & \longrightarrow & (\iota \blacktriangleleft \mathit{true}) : \varsigma, & \mathcal{E}[\![V]\!] & \\
\textsc{Reduce/False} & \varsigma, \ \mathcal{E}[\![V @^\flat_\iota \bot]\!] & \longrightarrow & (\iota \blacktriangleleft \mathit{false}) : \varsigma, & \mathcal{E}[\![V]\!] & \\
\\
\textsc{D-Domain} & \varsigma, \ \mathcal{E}[\![(V @^\flat_\iota (C \to \top))\, W]\!] & \longrightarrow & (\iota \blacktriangleleft \neg\iota_1) : \varsigma, & \mathcal{E}[\![V\,(W @^\flat_{\iota_1} C)]\!] & \iota_1 \notin \varsigma \\
\textsc{D-Range} & \varsigma, \ \mathcal{E}[\![(V @^\flat_\iota (\top \to D))\, W]\!] & \longrightarrow & (\iota \blacktriangleleft \iota_1) : \varsigma, & \mathcal{E}[\![(V\,W) @^\flat_{\iota_1} D]\!] & \iota_1 \notin \varsigma \\
\textsc{D-False} & \varsigma, \ \mathcal{E}[\![(V @^\flat_\iota (\top \to \bot))\, W]\!] & \longrightarrow & (\iota \blacktriangleleft \mathit{false}) : \varsigma, & \mathcal{E}[\![V\,W]\!] &
\end{array}
$$

Figure 6. Operational semantics extension of λ_{Con}.

$$
\begin{array}{lcl}
\mathcal{F}, \mathcal{G}, \mathcal{H} & ::= & \square \mid \lambda x.\mathcal{F} \mid \mathcal{F}\,M \mid T\,\mathcal{F} \mid op(\vec{T}\,\mathcal{F}\,\vec{M}) \\
& \mid & \mathit{if}\,\mathcal{F}\,M\,N \mid \mathit{if}\,T\,\mathcal{F}\,N \mid \mathit{if}\,T\,T\,\mathcal{F} \mid \mathcal{F} @^\flat_\iota C \\
\\
\mathcal{V} & ::= & \square \mid \mathcal{V} @^\flat_\iota \bot
\end{array}
$$

Figure 7. Transformation contexts.

4.3 Baseline Transformation

The baseline transformation $\varsigma, M \longmapsto_{\mathsf{B}} \varsigma', N'$ is defined on pairs of a constrains list and terms. As before, we write $\longmapsto^*_{\mathsf{B}}$ for its reflexive, transitive closure.

Figure 8 specifies the baseline transformation rules for λ_{Con} terms M. The first rules, Unfold/Assert, Unfold/Union, and Unfold/Intersection, unfold contracts and create constraints equivalent to the dynamics in Subsection 3.5.

Rule Unfold/Op unpacks a delayed contract on a term used in a base type operation. As this contract is never checked, it is true by definition (cf. (Keil and Thiemann 2015a)). Here, terms T_{I} and T_{Q} are canonical terms with a possible immediate or delayed contract.

Rules Unfold/D-Function and Unfold/D-Intersection unfold a delayed contract that appears left in an application corresponding to the dynamics. However, a function contract unfolds the domain portion to the term on the right, which is not necessarily a value.

Rule Unroll unrolls a delayed contract on a term T right in an application. This step removes the contract from T and grafts it to all uses of T. Even though this steps duplicates the number of contract assertions, we do not have more contract checks.

Rule Lower create a new function contract by lowering the contract on the function's body. The new function contract contains $\top$ on it's domain portion as it accepts every argument.

Rules Push/Immediate and Push/False reverse the order of a delayed contract nested in an assertion with an immediate contract or $\bot$. This step is required to evaluate immediate contracts on values and to unfold or unroll function's contracts.

Rule Push/If pushes a contract out of a condition if the contracts appears on both branches of the condition.

Rule Convert/True reduces all occurrences of $\top$ and produces a constraint that maps *true* to the corresponding blame variable ι. Knowledge about satisfied contracts can safely be removed from the source program because they do not compromise the blame assignment.[2]

In contrast to $\top$, $\bot$ (the knowledge about a failing contract) must remain in the source program and cannot be translated into a constraint statically. Doing this would over-approximate contract failures as the constrain might lead to a blame state, without knowing if the ill-behaved sources is ever executed. However, over-approximating violations might be intended, as Appendix G demonstrates.

Finally, rules Verify/True and Verify/True evaluate flat contracts on values that are available at compile time and transform the outcome of the evaluation into $\top$ or $\bot$. Values might be nested in a number of $\bot$ contracts of previously checked predicates.

Lemma 1 (Progress). *For all terms M it holds that either M is a canonical term T or there exists a transformation step $\varsigma, M \longmapsto_{\mathsf{B}} \varsigma', N$.*

4.4 Strong Blame-preservation

To sum up, the Baseline Transformation makes exactly the same transformation steps at compile time, than the dynamics would later on do. Thus, $\longmapsto_{\mathsf{B}}$ preserves the evaluation behavior of our program.

Conjecture 1 (Strong Blame-preservation). *For all transformations $\varsigma_M, M \longmapsto_B \varsigma_N, N$ it hold that either $\varsigma_M, M \Longrightarrow \varsigma'_M, V$ and $\varsigma_N, N \Longrightarrow \varsigma'_N, V$ or that $\varsigma_M, M \Longrightarrow \varsigma''_M, B^\flat$ and $\varsigma_N, N \Longrightarrow \varsigma''_N, B^\flat$.*

5. Subset Simplification

The *Subset Simplification* is the second tier in our transformation stack. Its objective is to reuse knowledge of previously checked contracts to avoid redundant checks whenever possible and to propagate contract violations to the surrounding module boundary.

The transformation first branches every alternative given by intersection and union contracts into an individual observation on which every contract must be fulfilled. Later it reduces contracts that are less restrictive and subsumed by other contracts. Finally, it joins remaining fragments to a new contract where possible.

To this end we introduce a subcontract relation which is closely related do already existing definitions of naive subtyping. Subcontracting claims a contract to be more restrictive than another contract.

5.1 Predicates

To achieve the best possible result it requires to have contracts and predicates as fine grained as possible. Intersection and union contracts enable to build complex contracts from different subcontracts. Distinct properties can be written in distinct predicates (flat contracts) and combined using intersection and union.

For example, a contract that checks for positive even numbers can be written as the intersection of one contract that checks for positive numbers and one contract that checks for even numbers:

$$\mathit{flat}(\lambda x.(x >= 0)) \cap \mathit{flat}(\lambda x.(x\%2 = 0))$$

However, there is a twist. Fine-grained sub-contracts enable contracts be optimized efficiently. But, at evaluation time, they must be handled in separation such that we result in more predicate checks.

For example, consider the following contract that checks for natural numbers:

$$\mathit{flat}(\lambda x.(x >= 0))$$

[2] Due to the specification of contract satisfaction it is also possible to remove $\top$ from the source program without creating a constraint. A non-existing constraint is handled as true because every non-violated contract assertion is satisfied, either because it is fulfilled or never verified.

UNFOLD/ASSERT	$\varsigma,\ \mathcal{F}\llbracket T @^{\flat} C \rrbracket$	$\longmapsto_{\mathsf{B}}$	$(\flat \blacktriangleleft \iota) : \varsigma,$	$\mathcal{F}\llbracket T @^{\flat}_{\iota} C \rrbracket$	$\iota \notin \varsigma$
UNFOLD/UNION	$\varsigma,\ \mathcal{F}\llbracket T @^{\flat}_{\iota} (C \cup D) \rrbracket$	$\longmapsto_{\mathsf{B}}$	$(\iota \blacktriangleleft \iota_1 \cup \iota_2) : \varsigma,$	$\mathcal{F}\llbracket (T @^{\flat}_{\iota_1} C) @^{\flat}_{\iota_2} D \rrbracket$	$\iota_1, \iota_2 \notin \varsigma$
UNFOLD/INTERSECTION	$\varsigma,\ \mathcal{F}\llbracket T @^{\flat}_{\iota} (I \cap C) \rrbracket$	$\longmapsto_{\mathsf{B}}$	$(\iota \blacktriangleleft \iota_1 \cap \iota_2) : \varsigma,$	$\mathcal{F}\llbracket (T @^{\flat}_{\iota_1} I) @^{\flat}_{\iota_2} C \rrbracket$	$\iota_1, \iota_2 \notin \varsigma$
UNFOLD/OP	$\varsigma,\ \mathcal{F}\llbracket op(\vec{T}_{\mathsf{I}},\ (T @^{\flat}_{\iota} Q),\ \vec{T}_{\mathsf{Q}}) \rrbracket$	$\longmapsto_{\mathsf{B}}$	$(\iota \blacktriangleleft \mathit{true}) : \varsigma,$	$\mathcal{F}\llbracket op(\vec{T}_{\mathsf{I}},\ T,\ \vec{T}_{\mathsf{Q}}) \rrbracket$	
UNFOLD/D-FUNCTION	$\varsigma,\ \mathcal{F}\llbracket (T_1 @^{\flat}_{\iota} (C \rightarrow D))\, T_2 \rrbracket$	$\longmapsto_{\mathsf{B}}$	$(\iota \blacktriangleleft \iota_1 \rightarrow \iota_2) : \varsigma,$	$\mathcal{F}\llbracket (T_1\, (T_2 @^{\flat}_{\iota_1} C)) @^{\flat}_{\iota_2} D \rrbracket$	$\iota_1, \iota_2 \notin \varsigma$
UNFOLD/D-INTERSECTION	$\varsigma,\ \mathcal{F}\llbracket (T_1 @^{\flat}_{\iota} (C \cap D))\, T_2 \rrbracket$	$\longmapsto_{\mathsf{B}}$	$(\iota \blacktriangleleft \iota_1 \cap \iota_2) : \varsigma,$	$\mathcal{F}\llbracket ((T_1 @^{\flat}_{\iota_1} C) @^{\flat}_{\iota_2} D)\, T_2) \rrbracket$	$\iota_1, \iota_2 \notin \varsigma$
UNROLL	$\varsigma,\ \mathcal{F}\llbracket \mathcal{V}\llbracket \lambda x.S \rrbracket\, (T @^{\flat}_{\iota} Q) \rrbracket$	$\longmapsto_{\mathsf{B}}$	ς	$\mathcal{F}\llbracket \mathcal{V}\llbracket \lambda x.S[x \mapsto (x @^{\flat}_{\iota} Q)] \rrbracket\, T \rrbracket$	
LOWER	$\varsigma,\ \mathcal{F}\llbracket \lambda x.(T @^{\flat}_{\iota} C) \rrbracket$	$\longmapsto_{\mathsf{B}}$	ς	$\mathcal{F}\llbracket (\lambda x.T) @^{\flat}_{\iota} (\top \rightarrow C) \rrbracket$	
PUSH/IMMEDIATE	$\varsigma,\ \mathcal{F}\llbracket (T @^{\flat_1}_{\iota_1} Q) @^{\flat_2}_{\iota_2} I \rrbracket$	$\longmapsto_{\mathsf{B}}$	ς	$\mathcal{F}\llbracket (T @^{\flat_2}_{\iota_2} I) @^{\flat_1}_{\iota_1} Q \rrbracket$	
PUSH/FALSE	$\varsigma,\ \mathcal{F}\llbracket (T @^{\flat_1}_{\iota_1} Q) @^{\flat_2}_{\iota_2} \bot \rrbracket$	$\longmapsto_{\mathsf{B}}$	ς	$\mathcal{F}\llbracket (T @^{\flat_2}_{\iota_2} \bot) @^{\flat_1}_{\iota_1} Q \rrbracket$	
PUSH/IF	$\varsigma,\ \mathcal{F}\llbracket (\mathit{if}\, T_0\, (T_1 @^{\flat}_{\iota} C)\, (T_2 @^{\flat}_{\iota} C)) \rrbracket$	$\longmapsto_{\mathsf{B}}$	ς	$\mathcal{F}\llbracket (\mathit{if}\, T_0\, T_1\, T_2) @^{\flat}_{\iota} C \rrbracket$	
CONVERT/TRUE	$\varsigma,\ \mathcal{F}\llbracket T @^{\flat}_{\iota} \top \rrbracket$	$\longmapsto_{\mathsf{B}}$	$(\iota \blacktriangleleft \mathit{true}) : \varsigma,$	$\mathcal{F}\llbracket T \rrbracket$	

VERIFY/TRUE

$$\frac{\cdot, M\, V \longrightarrow \varsigma, W \qquad \tau(W) = \mathit{true}}{\varsigma, \mathcal{F}\llbracket \mathcal{V}\llbracket S_{\mathsf{Val}} \rrbracket @^{\flat}_{\iota} \mathit{flat}(M) \rrbracket \longmapsto_{\mathsf{B}} \varsigma, \mathcal{F}\llbracket \mathcal{V}\llbracket S_{\mathsf{Val}} \rrbracket @^{\flat}_{\iota} \top \rrbracket}$$

VERIFY/FALSE

$$\frac{\cdot, M\, V \longrightarrow \varsigma, W \qquad \tau(W) = \mathit{false}}{\varsigma, \mathcal{F}\llbracket \mathcal{V}\llbracket S_{\mathsf{Val}} \rrbracket @^{\flat}_{\iota} \mathit{flat}(M) \rrbracket \longmapsto_{\mathsf{B}} \varsigma, \mathcal{F}\llbracket \mathcal{V}\llbracket S_{\mathsf{Val}} \rrbracket @^{\flat}_{\iota} \bot \rrbracket}$$

Figure 8. Baseline transformation rules.

Obviously, one can write a similar contract like this:

$$\mathit{flat}(\lambda x.(x > 0)) \cup \mathit{flat}(\lambda x.(x = 0))$$

The presence of intersection and union contracts enables developers to write fine grained contracts, which, in turn, enable us to identify redundant parts and to reduce already checked properties.

To avoid a run time impact cause by fine-grained predicates, we can easily conjunct or disjunct predicates of the same blame label after optimization. This is possible because the intersection and union of flat contracts corresponds to the conjunction and disjunction of predicates (cf. (Keil and Thiemann 2015a)).

5.2 Contract Subsets

When optimizing contracts, we use subcontracting to characterize when a contract is more restrictive than another contract, i.e. it subsumes all of its obligations. The definition of subcontracting is closely related to naive subtyping from the literature (Wadler and Findler 2009).

We write $C \sqsubseteq^* D$ if C is more restrictive than D, i.e. $\mathcal{E}\llbracket M @^{\flat} C \rrbracket \longrightarrow^* V$ implies that $\mathcal{E}\llbracket M @^{\flat} D \rrbracket \longrightarrow^* V$ and $\mathcal{E}\llbracket M @^{\flat} D \rrbracket \longrightarrow^* B^{\flat}$ implies that $\mathcal{E}\llbracket M @^{\flat} C \rrbracket \longrightarrow^* B^{\flat}$.

As a contract specifies the interface between a subject and its enclosing context, subcontracting must be covariant for both parties: context **and** subject.

For example, let *Positive? = flat(*λ*x.x>0)* and *Natural? = flat(*λ *x.x*$\geq$*0)*. Than *Positive?* $\sqsubseteq^*$ *Natural?* as for all x it holds that *x>0* implies *x*$\geq$*0*.

For another example, *Positive?* $\rightarrow$ *Positive?* is a subcontract of *Natural?* $\rightarrow$ *Natural?* as it is more restrictive on both portions. Furthermore, *Positive?* $\rightarrow$ *Natural?* and *Natural?* $\rightarrow$ *Positive?* are further subcontracts of *Positive?* $\rightarrow$ *Positive?*.

In order to specify our subcontracting judgement, we factor subcontracting into two subsidiary relations: subject subcontracting, written $C \sqsubseteq^+ D$, and context subcontracting, written $C \sqsubseteq^- D$. Relation $C \sqsubseteq^+ D$ indicates that the subject portion in C is more restrictive than the subject portion in D, whereas $C \sqsubseteq^- D$ indicates that C restricts the context more than D.

In contrast to Wadler and Findler's definition of positive and negative subtyping (cf. (Wadler and Findler 2009)), which is related to ordinary subtyping, our definition of positive and negative subcontracting is reversal because its is related to the naive subcontracting. We write $C \sqsubseteq^- D$ if C is more restrictive than D, whereas Wadler and Findler write $D \sqsubseteq^- C$.

Figure 9 shows both relations. The two judgement are defined in terms of each other and track swapping of responsibilities on function arguments. Both judgements are reflexive and transitive.

We further write $M \leq N$ if predicate M implies predicate N. For example, *(x>0)* $\leq$ *(x*$\geq$*0)* and *(x=0)* $\leq$ *(x*$\geq$*0)*.

However, our subcontract judgement stops at $M \leq N$. Relation $\leq$ refers to an environment Γ that resolves relations on terms. Γ might be given by a SAT solver or a set of predefined relations.

$$\frac{(M \leq N) \in \Gamma}{M \leq N}$$

It remains to defined subcontracting in terms of context and subject subcontracting.

Definition 2. *C is a* naive subcontract *of D, written $C \sqsubseteq^* D$, iff*

$$C \sqsubseteq^- D \quad \wedge \quad C \sqsubseteq^+ D.$$

We further define an ordinary subcontracting judgement, $C \sqsubseteq D$, which corresponds to the usuals definition of subtyping. Ordinary subcontracting is contravariant for the context portion and covariant for the subject portion.

Definition 3. *C is an* ordinary subcontract *of D, written $C \sqsubseteq D$, iff*

$$D \sqsubseteq^- C \quad \wedge \quad C \sqsubseteq^+ D.$$

In addition to the already mentiont notation, we sometimes use $\sqsubset$, $\sqsubset^*$, $\sqsubset^+$, and $\sqsubset^-$ to indicate a proper subcontract, analogous to already existing definitions on inequality.

Lemma 2. *Relation $\sqsubseteq^*$ is a preorder, i.e. it is reflexive and transitive. For all C, C', and D, we have that:*

- $C \sqsubseteq^* C$

Context Subcontract

$$C \sqsubseteq^- C \qquad I \sqsubseteq^- J \qquad \frac{C_D \sqsubseteq^+ D_D \quad C_R \sqsubseteq^- D_R}{C_D \to C_R \sqsubseteq^- D_D \to D_R}$$

$$\frac{C_D \sqsubseteq^- D_D}{x \to \Lambda x.C \sqsubseteq^- x \to \Lambda x.D} \qquad \frac{C_L \sqsubseteq^- D \quad C_R \sqsubseteq^- D}{C_L \cap C_R \sqsubseteq^- D}$$

$$\frac{C \sqsubseteq^- D_L}{C \sqsubseteq^- D_L \cap D_R} \qquad \frac{C \sqsubseteq^- D_R}{C \sqsubseteq^- D_L \cap D_R} \qquad \frac{C_L \sqsubseteq^- D}{C_L \cup C_R \sqsubseteq^- D}$$

$$\frac{C_R \sqsubseteq^- D}{C_L \cup C_R \sqsubseteq^- D} \qquad \frac{C \sqsubseteq^- D_L \quad C \sqsubseteq^- D_R}{C \sqsubseteq^- D_L \cup D_R}$$

Subject Subcontract

$$C \sqsubseteq^+ C \qquad C \sqsubseteq^+ \top \qquad \bot \sqsubseteq^+ D \qquad \frac{M \leq N}{\mathit{flat}(M) \sqsubseteq^+ \mathit{flat}(N)}$$

$$\frac{C_D \sqsubseteq^- D_D \quad C_R \sqsubset^+ D_R}{C_D \to C_R \sqsubseteq^+ D_D \to D_R} \qquad \frac{C_D \sqsubseteq^+ D_D}{x \to \Lambda x.C \sqsubseteq^+ x \to \Lambda x.D}$$

$$\frac{C_L \sqsubseteq^+ D}{C_L \cap C_R \sqsubseteq^+ D} \qquad \frac{C_R \sqsubseteq^+ D}{C_L \cap C_R \sqsubseteq^+ D}$$

$$\frac{C \sqsubseteq^+ D_L \quad C \sqsubseteq^+ D_R}{C \sqsubseteq^+ D_L \cap D_R} \qquad \frac{C_L \sqsubseteq^+ D \quad C_R \sqsubseteq^+ D}{C_L \cup C_R \sqsubseteq^+ D}$$

$$\frac{C \sqsubseteq^+ D_L}{C \sqsubseteq^+ D_L \cup D_R} \qquad \frac{C \sqsubseteq^+ D_R}{C \sqsubseteq^+ D_L \cup D_R}$$

Figure 9. Context and subject subcontracting.

$$\begin{array}{lcl} \mathcal{T} & ::= & \square \mid (\mathcal{T} \parallel M) \mid (T \parallel \mathcal{T}) \\ \mathcal{A} & ::= & \square \mid \mathcal{A} @^{\flat}_{\iota} C \\ \mathcal{B} & ::= & \square \mid \mathcal{B}\, M \mid T\, \mathcal{B} \mid op(\vec{T}\, \mathcal{B}\, \vec{M}) \mid \mathcal{B} @^{\flat}_{\iota} C \mid \mathit{if}\, \mathcal{B}\, M\, N \end{array}$$

Figure 10. Parallel observations and contexts.

- *if* $C \sqsubseteq^* C'$ *and* $C' \sqsubseteq^* D$ *then* $C \sqsubseteq^* D$

5.3 Canonical Terms, revisited

As for the Baseline Transformation, we first define a set of canonical terms (non-transformable) terms that serve as the output of this level's transformation. The new canonical terms are a subset of the already defined terms in Section 4.2. Appendix B.2 shows its definition.

In addition, Figure 10 shows the definition of a new transformation contexts $\mathcal{T}$ that looks through parallel observation. An assertion context $\mathcal{A}$ is a number of contract assertions and a body context $\mathcal{B}$ is a transformation context without lambda abstractions and without conditions with a hole in a branch.

5.4 Subset Transformation

Figure 11 defines the subset transformation as an extension of the baseline transformation in Figure 8.

A significant change is that the new rules FORK/UNION and FORK/INTERSECTION replace the already exiting rules with the name UNFOLD/UNION and UNFOLD/D-INTERSECTION. Both rules split the current observation and handle both sides of an intersection or union in separation. This step eliminates alternatives such that in each branch every contract must be fulfilled.

Doing this enables us to remove a contracts whose properties are subsumed by another contracts. Without splitting alternatives it would not be possible to remove a contract on the basis of other contracts: The other contracts might be nested in an alternative and thus it might never apply (as there is a weaker alternative).

As for the baseline rules, unions are split immediately when asserted to a value, whereas intersections must remain on the values until the values is used in an application. This is because the context can choose between both alternative on every use of the value.

Rule UNFOLD/INTERSECTION (Figure 8) must not be redefined, as the intersections with flat contracts is equivalent to a conjunction of its constituents (cf. (Keil and Thiemann 2015a)).

Rule LIFT lifts an immediate contract on a variable bound in a lambda abstraction and produces a function contract on that abstraction. However, at this level we are only allowed to lift contracts on variables directly contained in the function body. Lifting from a deeper abstraction would introduce checks that might never happen at run time.

Second, it is only allowed to lift an immediate contract. This is because a lifted delayed contract would remain on the argument value and thus it would effect all uses of that value, which entirely changes the meaning of the program.

Rule LOWER pushes a contract on a function body down and creates a new function contract from that contract. But now we have the restriction that the function's return is not it's argument.

Rule BLAME transforms a function body to a blame term $B^{\flat}$ in case a contract violation happens in that body. As we split alternatives we know that every contract must be fulfilled, and thus violating a contract immediately results in a contract violation for that branch. However, we cannot transform the whole execution to $B^{\flat}$ as we do not know if the body es ever executed (e.g. because its is nested in a condition or the lambda term is never used in an application).

Here, $\eta(\iota, \varsigma)$ computes a blame term $B^{\flat}$ resulting from a predicate violation ($\iota \blacktriangleleft \mathit{false}$) in ι (cf. Appendix C.3).

As before, $\square @^{\flat}_{\iota} \bot$ must remain on the $B^{\flat}$ term to remember the violated contract. However, as the whole body transforms to $B^{\flat}$, $\bot$ may stay as a contract on a function body. In this case, rule LOWER pushes the information of a failing contract down to the enclosing body, which in turn is unrolled when used in an application and rule BLAME transforms the enclosing context to $B^{\flat}$. Appendix H shows an example reduction.

The rules BLAME/IF/TRUE and BLAME/IF/FALSE do the same if a blame appears in branch of a conditions and rule BLAME/GLOBAL produces a global blame term if $\bot$ appears on a top-level term.

Rules SUBSET/LEFT and SUBSET/RIGHT removes a contract from a term that is less restrictive than another contract on that term. As this step removes a contract it might change the order of arising contract violations. The weaker contract might raise a contract violation before the stronger contract is checked. But, as the stronger contract remains, we know that we definitely result in a blame state until the stronger contract is satisfier. And, by definition, this implies that the weaker contract is also satisfied.

Rule PUSH/IF pushes a contract out of an condition if the contract remains ob both branches of the condition.

Rule MERGE merges remaining fragments of the same source contract to a new function. However, this can only be done if both contracts have the same responsibility, which is indicated by precondition $\eta(\iota_1, \varsigma) = \eta(\iota_2, \varsigma)$.

$$
\begin{array}{llllll}
\textsc{Fork/Union} & \varsigma,\ \mathcal{F}\llbracket T@^{\flat}_{\iota}(C \cup D)\rrbracket & \longmapsto'_{\mathsf{S}} & (\iota \blacktriangleleft \iota_1 \cup \iota_2) : \varsigma, & (\mathcal{F}\llbracket T@^{\flat}_{\iota_1} C\rrbracket \parallel \mathcal{F}\llbracket T@^{\flat}_{\iota_2} D\rrbracket) & \iota_1, \iota_2 \notin \varsigma \\
\textsc{Fork/Intersection} & \varsigma,\ \mathcal{F}\llbracket (T_1 @^{\flat}_{\iota}(Q \cap R))\, T_2\rrbracket & \longmapsto'_{\mathsf{S}} & (\iota \blacktriangleleft \iota_1 \cap \iota_2) : \varsigma, & (\mathcal{F}\llbracket (T_1 @^{\flat}_{\iota_1} Q)\, T_2)\rrbracket \parallel \mathcal{F}\llbracket (T_1 @^{\flat}_{\iota_2} R)\, T_2)\rrbracket) & \iota_1, \iota_2 \notin \varsigma
\end{array}
$$

$$
\begin{array}{llllll}
\textsc{Lift} & \varsigma,\ \mathcal{F}\llbracket \lambda x.\mathcal{B}\llbracket x@^{\flat}_{\iota} I\rrbracket\rrbracket & \longmapsto'_{\mathsf{S}} & (\iota \blacktriangleleft \neg\iota_1) : \varsigma, & \mathcal{F}\llbracket (\lambda x.\mathcal{B}\llbracket x\rrbracket)@^{\flat}_{\iota_1}(I \to \top)\rrbracket & \iota_1 \notin \varsigma \\
\textsc{Lower} & \varsigma,\ \mathcal{F}\llbracket \lambda x.(T@^{\flat}_{\iota} C)\rrbracket & \longmapsto'_{\mathsf{S}} & \varsigma & \mathcal{F}\llbracket (\lambda x.T)@^{\flat}_{\iota}(\top \to C)\rrbracket & T \neq x \\
\\
\textsc{Blame} & \varsigma,\ \mathcal{F}\llbracket \lambda x.\mathcal{B}\llbracket T@^{\flat}_{\iota}\bot\rrbracket\rrbracket & \longmapsto'_{\mathsf{S}} & \varsigma, & \mathcal{F}\llbracket \lambda x.(B^{\flat}@^{\flat}_{\iota}\bot\rrbracket) & B^{\flat} = \eta(\iota, \varsigma) \\
\textsc{Blame/If/True} & \varsigma,\ \mathcal{F}\llbracket \mathit{if}\, T_0\, \mathcal{B}\llbracket T_1 @^{\flat}_{\iota}\bot\rrbracket\, N\rrbracket & \longmapsto'_{\mathsf{S}} & \varsigma, & \mathcal{F}\llbracket \mathit{if}\, T_0\, (B^{\flat}@^{\flat}_{\iota}\bot)\, N\rrbracket & B^{\flat} = \eta(\iota, \varsigma) \\
\textsc{Blame/If/False} & \varsigma,\ \mathcal{F}\llbracket \mathit{if}\, T_0\, T_1\, \mathcal{B}\llbracket T_2 @^{\flat}_{\iota}\bot\rrbracket\rrbracket & \longmapsto'_{\mathsf{S}} & \varsigma, & \mathcal{F}\llbracket \mathit{if}\, T_0\, T_1\, (B^{\flat}@^{\flat}_{\iota}\bot)\rrbracket & B^{\flat} = \eta(\iota, \varsigma) \\
\textsc{Blame/Global} & \varsigma,\ \mathcal{B}\llbracket T@^{\flat}_{\iota}\bot\rrbracket & \longmapsto'_{\mathsf{S}} & \varsigma, & B^{\flat} & B^{\flat} = \eta(\iota, \varsigma) \\
\\
\textsc{Subset/Left} & \varsigma,\ \mathcal{F}\llbracket \mathcal{A}\llbracket T@^{\flat}_{\iota_1} C\rrbracket @^{\flat}_{\iota_2} D\rrbracket & \longmapsto'_{\mathsf{S}} & \varsigma, & \mathcal{F}\llbracket \mathcal{A}\llbracket T@^{\flat}_{\iota_1} C\rrbracket\rrbracket & C \sqsubseteq^{*} D \\
\textsc{Subset/Right} & \varsigma,\ \mathcal{F}\llbracket \mathcal{A}\llbracket T@^{\flat}_{\iota_1} C\rrbracket @^{\flat}_{\iota_2} D\rrbracket & \longmapsto'_{\mathsf{S}} & \varsigma, & \mathcal{F}\llbracket \mathcal{A}\llbracket T@^{\flat}_{\iota_2} D\rrbracket\rrbracket & C \sqsupset^{*} D \\
\\
\textsc{Push/If} & \varsigma,\ \mathcal{F}\llbracket (\mathit{if}\, T_0\, \mathcal{A}_1\llbracket T_1 @^{\flat}_{\iota} C\rrbracket\, \mathcal{A}_2\llbracket T_2 @^{\flat}_{\iota} C\rrbracket)\rrbracket & \longmapsto'_{\mathsf{S}} & \varsigma & \mathcal{F}\llbracket (\mathit{if}\, T_0\, \mathcal{A}_1\llbracket T_1\rrbracket\, \mathcal{A}_2\llbracket T_2\rrbracket)@^{\flat}_{\iota} C\rrbracket &
\end{array}
$$

$$
\textsc{Merge}\ \frac{Q = C \to \top \qquad R = \top \to D \qquad \eta(\iota_1, \varsigma) = \eta(\iota_2, \varsigma) \qquad \iota_3 \notin \varsigma}{\varsigma, \mathcal{F}\llbracket \mathcal{A}\llbracket T@^{\flat}_{\iota_1} Q\rrbracket @^{\flat}_{\iota_2} R\rrbracket \longmapsto'_{\mathsf{S}} \iota_1 \blacktriangleleft \iota_3 : \iota_2 \blacktriangleleft \iota_3 : \varsigma\, \mathcal{F}\llbracket \mathcal{A}\llbracket T@^{\flat}_{\iota_3} C \to D\rrbracket\rrbracket}
\qquad
\textsc{Baseline}\ \frac{\varsigma,\ M \longmapsto_{\mathsf{B}} \varsigma',\ N}{\varsigma,\ M \longmapsto'_{\mathsf{S}} \varsigma',\ N}
\qquad
\textsc{Trace}\ \frac{\varsigma,\ M \longmapsto'_{\mathsf{S}} \varsigma',\ N}{\varsigma,\ \mathcal{T}\llbracket M\rrbracket \longmapsto_{\mathsf{S}} \varsigma',\ \mathcal{T}\llbracket N\rrbracket}
$$

Figure 11. Subset transformation.

$$
\begin{array}{lcl}
\mathcal{M}, \mathcal{N} & ::= & \square \mid \lambda x.\mathcal{M} \mid \mathcal{M}\, N \mid M\, \mathcal{N} \mid op(\vec{M}\, \mathcal{M}\, \vec{N}) \\
 & \mid & \mathit{if}\, \mathcal{M}\, M\, N \mid \mathit{if}\, M\, \mathcal{M}\, N \mid \mathit{if}\, M\, N\, \mathcal{M} \mid \mathcal{L}@^{\flat}_{\iota} C \\
\mathcal{L}, & ::= & \lambda x.\mathcal{M} \mid \mathcal{M}\, N \mid M\, \mathcal{N} \mid op(\vec{M}\, \mathcal{M}\, \vec{N}) \\
 & \mid & \mathit{if}\, \mathcal{M}\, M\, N \mid \mathit{if}\, M\, \mathcal{M}\, N \mid \mathit{if}\, M\, N\, \mathcal{M} \mid \mathcal{L}@^{\flat}_{\iota} C
\end{array}
$$

Figure 12. Final terms and contexts.

Finally, the rule BASELINE lifts baseline transformations to subset transformations, and rule TRACE considers the transformation in a parallel observation.

Lemma 3 (Progress)**.** *For all terms M it holds that either M is a canonical term T or there exists a transformation step $\varsigma, M \longmapsto_{S} \varsigma', N$.*

Apart from the subset transformation which uses native subcontracting to reduces contracts subsumed by other contract, we also use ordinary subcontracting to simplifiy contracts contained in an intersection or union contract. Appendix F demonstrates this.

5.5 Join Traces

The subset transformation splits intersection and union into separated observation, i.e. its outcome is a set of parallel observations. Thus, after transforming all observations to canonical terms, we need to join the remaining fragments to a valid source program.

Joining terms must be one of the last step as it is not possible to apply further transformations afterwards. Subsequent transformations would mix up contracts from different alternatives and thus they may change the meaning of a contract.

To this end, Figure 12 defines a context $\mathcal{M}$, which is defined as usual as a term with a hole. The only exception is that contexts $\mathcal{M}$ did not have holes in a contract assertion.

As our transformation did not change the basic syntax of a program (it only propagates contracts), terms remain equivalent (excepting contract assertions) until they transform to a blame term. It follows that the only difference between parallel terms are contract assertions and blame terms.

We call terms that differ only in contract assertions and blame terms as *structurally equivalent*. We write $M \simeq N$ if term M is structurally equivalent to term N and $\mathcal{G} \simeq \mathcal{H}$ for the structural equivalence of contexts.

Definition 4. *Two terms are structural equal, written $M \simeq N$, if they only differ in contract assertions and blame terms.*

Definition 5. *Two contexts are structural equal, written $\mathcal{M} \simeq \mathcal{N}$, if they only differ in contract assertions and blame terms.*

Figure 13 shows the structural equivalence of terms and contexts. The equivalence relations looks through contract assertions and compares only λ_{J} terms.

Lemma 4. *Relation $\simeq$ on terms M and context $\mathcal{M}$ is an equivalence relation.*

To join terms in parallel observations we only need to walk through all terms and to synchronise contract assertions and blame terms. Figure 14 presents the synchronisation of contracts and terms, indicated by relation $T \longmapsto_{\mathsf{J}} M$ on terms.

First, rule JOIN triggers the synchronization of two canonical terms nested in a parallel observation and rule MATCH dissolves a parallel observation if both terms are identical.

The rules SYNCHRONIZE/LEFT and SYNCHRONIZE/RIGHT handle the case that one side results in a blame state. In this case the blame term is omitted and the operation proceeds with other term. Remember, it is not required to maintain the blame term because $\square@^{\flat}_{\iota}\bot$ still remains in the source program.

Finally, rule SYNCHRONIZE/CONTRACT looks for different contract assertions on a term S and merges the contract assertions. Precondition $\mathcal{G} \equiv \mathcal{H}$ requires that the enclosing contexts are structurally equivalent. Only the contract assertions must be different.

Here, $\mathcal{A}_r \sqcup \mathcal{A}_l$ computes the union of both contexts and synchronizing proceeds with the new context in place.

The easiest way to merge contexts is to place one context in the hole of the other context. Even through this did not change the blame behaviour of a program it might duplicate contract checks, and thus it violates our ground rule not to introduce more checks.

Term Equivalence

$$K \simeq K \qquad x \simeq x \qquad \frac{M \simeq N}{\lambda x.M \simeq \lambda x.N}$$

$$\frac{M \simeq N}{\lambda x.M \simeq \lambda x.N} \qquad \frac{M_0 \simeq N_0 \quad M_1 \simeq N_1}{M_0\, M_1 \simeq N_0\, N_1}$$

$$\frac{\vec{M} \simeq \vec{N}}{op(\vec{M}) \simeq op(\vec{N})} \qquad \frac{\vec{M}_0 \simeq \vec{N}_0 \quad \vec{M}_1 \simeq \vec{N}_1 \quad \vec{M}_2 \simeq \vec{N}_2}{\mathit{if}\, M_0\, M_1\, M_2 \simeq \mathit{if}\, N_0\, N_1\, N_2}$$

$$\frac{M \simeq N}{M @^{\flat}_{\iota} C \simeq N} \qquad \frac{M \simeq N}{M \simeq N @^{\flat}_{\iota} C} \qquad M \simeq B^{\flat} \qquad B^{\flat} \simeq N$$

Context Equivalence

$$\square \simeq \square \qquad \frac{\mathcal{M} \simeq \mathcal{N}}{\mathcal{M} @^{\flat}_{\iota} C \simeq \mathcal{N}} \qquad \frac{\mathcal{M} \simeq \mathcal{N}}{\mathcal{M} \simeq \mathcal{N} @^{\flat}_{\iota} C}$$

$$\frac{\mathcal{M} \simeq \mathcal{N}}{\lambda x.\mathcal{M} \simeq \lambda x.\mathcal{N}} \qquad \frac{\mathcal{M} \simeq \mathcal{N} \quad M \simeq N}{\mathcal{M}\, M \simeq \mathcal{N}\, N}$$

$$\frac{\mathcal{M} \simeq \mathcal{N} \quad M \simeq N}{M\, \mathcal{M} \simeq N\, \mathcal{M}}$$

$$\frac{\mathcal{M} \simeq \mathcal{N} \quad \vec{M}_0 \simeq \vec{N}_0 \quad \vec{M}_i \simeq \vec{N}_i}{op(\vec{M}_0\, \mathcal{M}\, \vec{M}_i) \simeq op(\vec{N}_0\, \mathcal{N}\, \vec{N}_i)}$$

$$\frac{\mathcal{M} \simeq \mathcal{N} \quad M_1 \simeq N_1 \quad M_2 \simeq N_2}{\mathit{if}\, \mathcal{M}\, M_1\, M_2 \simeq \mathit{if}\, \mathcal{N}\, N_1\, N_2}$$

$$\frac{\mathcal{M} \simeq \mathcal{N} \quad M_0 \simeq N_0 \quad M_2 \simeq N_2}{\mathit{if}\, M_0\, \mathcal{M}\, M_2 \simeq \mathit{if}\, N_1\, \mathcal{N}\, N_2}$$

$$\frac{\mathcal{M} \simeq \mathcal{N} \quad M_0 \simeq N_0 \quad M_1 \simeq N_1}{\mathit{if}\, M_0\, M_1\, \mathcal{M} \simeq \mathit{if}\, N_0\, N_1\, \mathcal{N}}$$

Figure 13. Term and context equivalence.

JOIN

$$\frac{(M \parallel N) \longmapsto'_{\mathsf{J}} (M' \parallel N')}{\mathcal{T}[\![(M \parallel N)]\!] \longmapsto_{\mathsf{J}} \mathcal{T}[\![(M' \parallel N')]\!]}$$

MATCH

$$(M \parallel M) \longmapsto'_{\mathsf{J}} M$$

SYNCHRONIZE/LEFT

$$\frac{\mathcal{M} \simeq \mathcal{N}}{(\mathcal{M}[\![\mathcal{A}_l[\![S]\!]]\!] \parallel \mathcal{N}[\![\mathcal{A}_r[\![B^{\flat}]\!]]\!]) \longmapsto'_{\mathsf{J}} (\mathcal{M}[\![\mathcal{A}_l[\![S]\!]]\!] \parallel \mathcal{N}[\![\mathcal{A}_r[\![S]\!]]\!])}$$

SYNCHRONIZE/RIGHT

$$\frac{\mathcal{M} \simeq \mathcal{N}}{(\mathcal{M}[\![\mathcal{A}_l[\![B^{\flat}]\!]]\!] \parallel \mathcal{N}[\![\mathcal{A}_r[\![S]\!]]\!]) \longmapsto'_{\mathsf{J}} (\mathcal{M}[\![\mathcal{A}_l[\![S]\!]]\!] \parallel \mathcal{N}[\![\mathcal{A}_r[\![S]\!]]\!])}$$

SYNCHRONIZE/CONTRACT

$$\frac{\mathcal{M} \simeq \mathcal{N} \quad \mathcal{A}_l \neq \mathcal{A}_r \quad \mathcal{A} = \mathcal{A}_r \sqcup \mathcal{A}_l}{(\mathcal{M}[\![\mathcal{A}_l[\![S]\!]]\!] \parallel \mathcal{N}[\![\mathcal{A}_r[\![S]\!]]\!]) \longmapsto'_{\mathsf{J}} (\mathcal{M}[\![\mathcal{A}[\![S]\!]]\!] \parallel \mathcal{N}[\![\mathcal{A}[\![S]\!]]\!])}$$

Figure 14. Join parallel observations.

CONDENSE

$$\varsigma,\, \mathcal{M}[\![\mathcal{A}[\![M @^{\flat_0}_{\iota_0} C]\!] @^{\flat_1}_{\iota_1} C]\!] \longmapsto_{\mathsf{C}} \iota_1 \blacktriangleleft \iota_0 : \varsigma,\, \mathcal{M}[\![\mathcal{A}[\![M @^{\flat_0}_{\iota_0} C]\!]]\!]$$

Figure 15. Condense transformation.

Thus, merging assertion contexts only places contract assertions in the hole that are not already contained in the context. Formally, we specify the join of two assertion contexts $\mathcal{A}$ and $\mathcal{A}'$ by

$$\mathcal{A} \sqcup \mathcal{A}' = \mathcal{A}[\![\mathcal{A}' \setminus \mathcal{A}]\!]$$

Here, $\mathcal{A}' \setminus \mathcal{A}$ computes a new context with assertions from $\mathcal{A}'$ that are not already contained in $\mathcal{A}$. Appendix C.4 shows its computation.

5.6 Condense Remaining Contracts

Finally, one last step remains to be done: After synchronizing contracts it might happen that we have identical contracts on the same term. As they belong to different alternatives we cannot remove one of them, but we can condense them a single assertion.

Figure 15 shows the transformation. If a term has two assertions of the same contract, Rule CONDENSE removes one of them and creates a new constraint that redirect the result from the other assertion to the blame variable of the remove assertion.

5.7 Bringing all this together

To model transformation with parallel observations, we define a new transformation $\Longmapsto$ on terms M. It first applies the standard transformation $\longmapsto_{\mathsf{S}}$ until it reaches a canonical term T. Second, it applies $\longmapsto_{\mathsf{J}}$ to join terms and $\longmapsto_{\mathsf{C}}$ to condense the remaining contracts.

$$\frac{\cdot, M \longmapsto^*_{\mathsf{S}} \varsigma, T \quad T \longmapsto^*_{\mathsf{J}} L \quad L \longmapsto^*_{\mathsf{C}} N}{\cdot, M \Longmapsto^* \varsigma, N}$$

5.8 Weak Blame-preservation

While reorganizing contracts, the Subset Transformation did no strictly preserve the blame behaviour of a program. An ill-behaved program by lead to another contract violation first, whereas well-behaved programs still result in the same output.

Conjecture 2 (Weak Blame-preservation)**.** *For all $\varsigma_M, M \Longmapsto^* \varsigma_N, N$ it hold that either $\varsigma_M, M \Longmapsto \varsigma'_M, V$ and $\varsigma_N, N \Longmapsto \varsigma'_N, V$ or $\varsigma_M, M \Longmapsto \varsigma''_M, B^{\flat}_M$ and $\varsigma_N, N \Longmapsto \varsigma''_N, B^{\flat'}_N$.*

6. Technical Results

In addition to weak and string blame preservation, our contract simplification grantees not to introduce more predicate checks at run time. Even throut we reorganize and duplicate contract assertions, the total number of predicate checks at run time remains the same or decrease on every transformation step.

Definition 6 (Size)**.** *The size $|L|$ of a closed term L is the number of predicates checks during reduction $\Longrightarrow$ of L.*

Theorem 1 (Improvement)**.** *For each term M with $\cdot, M \Longmapsto \varsigma, N$ it holds that $|N| \leq |M|$.*

Finally, to prove soundness of our static contract simplification, we need to show that our transformation terminates.

Theorem 2 (Termination)**.** *For each term M, there exists a canonical term T such that $\varsigma_M, M \Longmapsto^* \varsigma_T, T$.*

Benchmark	Normal *time (sec)*	Baseline *time (sec)*	Subset *time (sec)*
Example1	39.4	27.0 (-31.27 %)	27.0 (-31.37 %)
Example2	87.1	58.4 (-33.00 %)	46.1 (-47.12 %)
Example3	66.5	54.4 (-18.17 %)	26.5 (-60.11 %)
Example4	114.5	85.1 (-25.63 %)	44.6 (-61.01 %)
Example5	148.3	108.0 (-27.18 %)	60.0 (-59.55 %)
Example6	295.7	200.0 (-32.36 %)	118.6 (-59.90 %)

Figure 16. Results from running the TreatJS contract system. Column **Normal** gives the baseline execution time of the unmodified program, whereas column **Baseline** and column **Subset** contain the execution time and the improvement (in percent) after applying the baseline or subset simplification, respectively.

7. Practical Evaluation

To give an insight into the run time improvements of our system we apply the simplifications to different versions of the *addOne* example from Section 2. Our testing procedure uses the *addOne* function to increase a counter on each iteration in a while loop.

Each example program addresses a certain property and contains a different number of contracts. For example, *Example2* corresponds the *addOne* function in Section 2.1 and *Example4* corresponds to the function in Section 2.6. Appendix J shows the JavaScript implementation of all benchmark programs.

To run the examples we use the TreatJS (Keil and Thiemann 2015b) contract system for JavaScript and the SpiderMonkey[3] JavaScript engine. Lacking an implementation (beyond the PLT Redex model) we applied the simplification steps manually.

Figure 16 shows the execution time required for a loop with *100000* iterations. The numbers indicate that the *Baseline Simplification* improves the run time by approximately 28%, whereas the *Subset Simplification* makes an improvement up to 62%. Appendix J shows the full table of results.

In addition, there is another aspects not already addressed by Keil and Thiemann (2015b): Adding contracts may also prevent a program from being optimized. The SpiderMonkey engine, for example, uses two different optimizing compilers to speed up operation. When activating SpiderMonkey's optimizing JIT, the run time of the normal program without contracts decreases by 20x, whereas the run time of the program with contracts only decreases by 2x. This lack of optimization opportunities for the JIT is one important reason for the big run time deterioration they reported.

8. Related Work

Higher-Order Contracts Software contracts were introduced with Meyer's *Design by Contract*™ methodology (Meyer 1988) which stipulates the specification of Hoare-like pre- and postconditions for all components of a program and introduces the idea of monitoring these contracts while the program executes.

Findler and Felleisen (2002) extend contracts and contract monitoring to higher-order functional languages. Their work has attracted a plethora of follow-up works that ranges from semantic investigations (Blume and McAllester 2006; Findler and Blume 2006) over studies on blame assignment (Dimoulas et al. 2011; Wadler and Findler 2009) to extensions in various directions: intersection and union contracts (Keil and Thiemann 2015a), polymorphic contracts (Ahmed et al. 2011; Belo et al. 2011), behavioral and temporal contracts (Dimoulas et al. 2012; Disney et al. 2011), etc.

[3] `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`

Contract Validation Contracts may be validated statically or dynamically. However, most frameworks perform run-time monitoring as proposed in Meyer's work.

Dynamic contract checking, as for example performed by Findler and Felleisen (2002); Keil and Thiemann (2015b), enable programmers to specify properties of a component without restricting the flexibility of the underlying dynamic programming language. However, run time monitoring imposes a significant overhead as it extends the original program with contract checks.

Purely static frameworks (e.g., ESC/Java (Flanagan et al. 2002)) transform specifications and programs into verification conditions to be verified by a theorem prover. Others (Xu et al. 2009; Tobin-Hochstadt and Van Horn 2012) rely on program transformation and symbolic execution to prove adherence to contracts.

Static contract checking avoids additional run-time checks, but existing approaches are incomplete and limited to check a restricted set of properties. They rely on theorem proving engines to discharge contracts that are written in the host language: the translation to logic may not always succeed.

Contract Simplification Nguyen et al. (2014) present a static contract checker that has evolved from Tobin-Hochstadt and Van Horn's work on symbolic execution (Tobin-Hochstadt and Horn 2012). Their approach is to verify contracts by executing programs on *unknown* abstract values that are refined by contracts as they are applied. If a function meets its obligation, the corresponding contract gets removed. However, their approach only applies to the positive side of a function contract.

Propagating contracts is closely related to symbolic executions. Both refine the value of a term based on contracts, whether by moving the contract through the term or by using an abstract value.

Compared to our work, Nguyen et al. (2014) addresses the opposite direction. Where we unroll a contract to its enclosing context and decompose a contract into its components, they verify the function's obligations based on the given domain specification. However, in case the given contract cannot be verified, they must retain the whole contract at its original position and they are not able to simplify the domain portion of a function contract.

Furthermore, their symbolic verification is not able to handle true alternatives in the style of intersection and union contracts.

We claim that both approaches are complementary to one another. Our contract simplification would benefit from a preceding verification that simplifies the function's obligations before unrolling a contract to the enclosing context.

Xu (2012) also combines static and dynamic contract checking. Her approach translates contracts into verification conditions that get verified statically. Whereas satisfied conditions will be removed, there may be conditions that cannot be proved: they remain in the source program in the form of dynamic checks.

9. Conclusion

The goal of static contract simplification is to reduce the overhead of run-time contract checking. To this end, we decompose contracts and propagates their components through the program. In many cases we are able to discharge parts of contracts statically. The remaining components are simplified and reassembled to a new and cheaper contract at a module boundary.

As a novel aspect, we can simplify contracts that cannot be verified entirely at compile time. The remaining contract fragments stay in the program and get lifted to the enclosing module boundary.

A case study with microbenchmarks shows some promise. Our simplification decreases the total number of predicate checks at run time and thus reduces the run-time impact caused by contracts and contract monitoring. The study also shows that the degree of improvement depends on the granularity of contracts. Fine-grained contracts enable more improvement.

References

A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In Ball and Sagiv (2011), pages 201–214.

T. Ball and M. Sagiv, editors. *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, 2011. ACM.

J. F. Belo, M. Greenberg, A. Igarashi, and B. C. Pierce. Polymorphic contracts. In G. Barthe, editor, *Proceedings of the 20th European Symposium on Programming*, volume 6602 of *Lecture Notes in Computer Science*, pages 18–37, Saarbrücken, Germany, Mar. 2011. Springer-Verlag. ISBN 978-3-642-19717-8.

M. Blume and D. McAllester. Sound and complete models of contracts. *Journal of Functional Programming*, 16:375–414, July 2006. ISSN 0956-7968. doi: 10.1017/S0956796806005971.

C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: No more scapegoating. In Ball and Sagiv (2011), pages 215–226.

C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Complete monitors for behavioral contracts. In *European Symposium on Programming*, volume 7211 of *Lecture Notes in Computer Science*, Tallinn, Estland, Apr. 2012. Springer-Verlag.

T. Disney. contracts.js. `https://github.com/disnet/contracts.js`, Apr. 2013.

T. Disney, C. Flanagan, and J. McCarthy. Temporal higher-order contracts. In O. Danvy, editor, *Proceedings International Conference on Functional Programming 2011*, pages 176–188, Tokyo, Japan, Sept. 2011. ACM Press, New York. ISBN 978-1-4503-0865-6.

M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

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-Verlag.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In S. Peyton-Jones, editor, *Proceedings International Conference on Functional Programming 2002*, pages 48–59, Pittsburgh, PA, USA, Oct. 2002. ACM Press, New York.

C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 2002. ACM Press. ISBN 1-58113-463-0. doi: http://doi.acm.org/10.1145/512529.512558.

M. Flatt, R. B. Findler, and PLT. The racket guide. `http://docs.racket-lang.org/guide/index.html`, Feb. 2014. Version 6.0.

M. Keil and P. Thiemann. Blame assignment for higher-order contracts with intersection and union. In K. Fisher and J. H. Reppy, editors, *ICFP 2015*, pages 375–386, Vancouver, BC, Canada, Sept. 2015a. ACM.

M. Keil and P. Thiemann. Treatjs: Higher-order contracts for JavaScript. In J. Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPICS*, pages 28–51, Prague, Czech Repulic, July 2015b. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-86-6.

G. T. Leavens and M. B. Dwyer, editors. *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, 2012. ACM.

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

P. C. Nguyen, S. Tobin-Hochstadt, and D. V. Horn. Soft contract verification. In J. Jeuring and M. M. T. Chakravarty, editors, *ICFP'14*, pages 139–152, Gothenburg, Sweden, Sept. 2014. ACM. ISBN 978-1-4503-2873-9. URL `http://dl.acm.org/citation.cfm?id=2628136`.

$$\tau(V) = \begin{cases} \mathit{false} & V = \mathit{false} \\ \tau(W) & V = W @^{\flat}_{\iota} Q \\ \mathit{true} & \text{otherwise} \end{cases}$$

Figure 17. Mapping values to truth values.

A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? In R. Bodík and R. Majumdar, editors, *POPL 2016*, pages 456–468, St. Petersburg, FL, USA, Jan. 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837630. URL `http://doi.acm.org/10.1145/2837614.2837630`.

S. Tobin-Hochstadt and D. V. Horn. Higher-order symbolic execution via contracts. In Leavens and Dwyer (2012), pages 537–554.

S. Tobin-Hochstadt and D. Van Horn. Higher-order symbolic execution via contracts. In Leavens and Dwyer (2012), pages 537–554.

P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In G. Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2009. ISBN 978-3-642-00589-3. doi: 10.1007/978-3-642-00590-9. URL `http://dx.doi.org/10.1007/978-3-642-00590-9`.

D. N. Xu. Hybrid contract checking via symbolic simplification. In O. Kiselyov and S. Thompson, editors, *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, January 23-24, 2012*, pages 107–116. ACM, 2012. ISBN 978-1-4503-1118-2.

D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In B. Pierce, editor, *Proceedings 36th Annual ACM Symposium on Principles of Programming Languages*, pages 41–52, Savannah, GA, USA, Jan. 2009. ACM Press. ISBN 978-1-60558-379-2.

10. Constraint Satisfaction

The dynamics in Figure 3 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, $b.\mathit{subject}$ and $b.\mathit{context}$. Intuitively, if $b.\mathit{subject}$ is false, then the contract b is not subject-satisfied and may lead to positive blame for b. If $b.\mathit{context}$ is false, then there is a context that does not respect contract b and may lead to negative blame for b.

An *interpretation* μ of a constraint list ς is a mapping from blame identifiers to records of elements of $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$, such that all constraints are satisfied. We order truth values by $\mathit{true} \sqsubset \mathit{false}$ and write $\sqsubseteq$ for the reflexive closure of that ordering. This ordering reflects gathering of informations with each execution step.

Formally, we specify the mapping by

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

where $(\!|b|\!)$ ranges over metavariable b and constraint satisfaction by a relation $\mu \models \varsigma$, which is specified in Figure 2.

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 concatenation of a constraint with a constraint list corresponds to the intersection of sets of solutions (CS-STATE). The indirection constraint just forwards its referent (CT-INDIRECTION).

In rule CT-FLAT, W is the outcome of the predicate of a flat contract. The rule sets subject satisfaction to *false* if $W = \mathit{false}$ and otherwise to *true*, where the function $\tau(\cdot) : (\!|V|\!) \to \mathbb{B}$ translates

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

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

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

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

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

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

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

Figure 18. Constraint satisfaction.

values to truth values by stripping delayed contracts (see Figure 1). A flat contract never blames its context so that $b.\mathit{context}$ 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 satisfies contract b if it satisfies its obligations towards its argument $\iota_1.\mathit{context}$ **and** if the argument satisfies its contract then the result satisfies its contract, too. The first part arises if the function 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 $\iota_1.\mathit{subject}$ **and** uses the result according to its contract $\iota_2.\mathit{context}$. 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: $\iota_1.\mathit{subject} \wedge \iota_2.\mathit{subject}$. A context, however, has the choice to fulfill one of the constituent contracts: $\iota_1.\mathit{context} \vee \iota_2.\mathit{context}$.

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

Canonical Terms

$$
\begin{array}{lcl}
S_{\mathsf{Val}} & ::= & K \mid \lambda x.S \\
S_{\mathsf{NonVal}} & ::= & x \mid B^{\flat} \\
 & \mid & T_{\mathsf{I}}\, T \mid T_{\mathsf{Const}}\, T \mid T_{\mathsf{Abs}}\, T_{\mathsf{I}} \mid T_{\mathsf{Abs}}\, T_{\mathsf{Val}} \\
 & \mid & op(\overrightarrow{T_{\mathsf{NonQ}}}) \\
 & \mid & \mathit{if}\, T_0\, T_1\, T_2 \quad \textbf{where not} \quad T_1 = T_1' @^{\flat}_{\iota} C \textbf{ and} \\
 & & T_2 = T_2' @^{\flat}_{\iota} C \\
 & & \\
S & ::= & S_{\mathsf{Val}} \mid S_{\mathsf{NonVal}} \\
 & & \\
T_{\mathsf{Const}} & ::= & K \mid T_{\mathsf{Const}} @^{\flat}_{\iota} \bot \\
T_{\mathsf{Abs}} & ::= & \lambda x.S \mid T_{\mathsf{Abs}} @^{\flat}_{\iota} \bot \\
T_{\mathsf{Val}} & ::= & S_{\mathsf{Val}} \mid T_{\mathsf{Val}} @^{\flat}_{\iota} \bot \\
 & & \\
T_{\mathsf{I}} & ::= & S_{\mathsf{NonVal}} \mid T_{\mathsf{I}} @^{\flat}_{\iota} I \mid T_{\mathsf{I}} @^{\flat}_{\iota} \bot \\
T_{\mathsf{Q}} & ::= & T_{\mathsf{Val}} \mid T_{\mathsf{I}} \mid T_{\mathsf{Q}} @^{\flat}_{\iota} Q \\
T_{\mathsf{NonQ}} & ::= & T_{\mathsf{Val}} \mid T_{\mathsf{I}} \\
 & & \\
T & ::= & T_{\mathsf{Q}}
\end{array}
$$

Non-Canonical Terms

$$
\begin{array}{lcl}
O, P & ::= & \lambda x.O \mid O\, N \mid M\, O \mid op(\vec{M}\, O\, \vec{N}) \\
 & \mid & \mathit{if}\, O\, M\, N \mid \mathit{if}\, L\, O\, N \mid \mathit{if}\, L\, M\, O \mid O @^{\flat}_{\iota} C \\
 & \mid & (\mathcal{V}[\![\lambda x.M]\!])\, (N @^{\flat}_{\iota} Q) \mid (M @^{\flat}_{\iota} Q)\, N \\
 & \mid & \mathcal{V}[\![V]\!] @^{\flat}_{\iota} I \mid \lambda x.(M @^{\flat}_{\iota} C) \\
 & \mid & op(\vec{L}\, (M @^{\flat}_{\iota} Q)\, \vec{N}) \mid \mathit{if}\, L\, (M @^{\flat}_{\iota} C)\, (N @^{\flat}_{\iota} C) \\
 & \mid & M @^{\flat}_{\iota} \top \mid M @^{\flat} C \mid M @^{\flat}_{\iota} C \cup D \mid M @^{\flat}_{\iota} I \cap C \\
 & \mid & (M @^{\flat}_{\iota_q} Q) @^{\flat}_{\iota_i} I \mid (M @^{\flat}_{\iota_q} Q) @^{\flat}_{\iota_i} \bot
\end{array}
$$

Figure 19. Canonical and non-canonical terms of the *Baseline Transformation*.

11. Canonical Terms

Canonical (non-transformable) terms are the output of our transformation. Each level has its own output grammar consisting of λ_{Con} term M that cannot be further transformed at this level.

Canonical terms distinguish contract-free term (indicated by latter S) from terms with a contract (indicated by latter T).

11.1 Canonical Term of the Baseline Transformation

This sections considers the output terms of the *Baseline Transformation* (Section 4.3). Figure 3 defines the syntax of canonical terms S and T as a subset of λ_{Con} term M.

A terms without a contract S is either a value term S_{Val}, including first-order constants and lambda abstractions, or a non-value term S_{NonVal}, which is either a variable, a blame term, an application, a primitive operation, or a condition.

Applications are divided into different combinations. This excludes all reducible combinations and only the given terms remain. For example, a function contract on the left side of an application is reducible, whereas a function contract in the right side is only reducible if the left side is a lambda term.

A canonical term T is a term T_{Q}, which is either a term T_{Val} or a term T_{I} with an indefinite number of delayed contracts on the outermost position. Terms T_{Val} are all values with remaining $\bot$ contracts. Terms T_{I} are all terms with immediate contracts or $\bot$ contracts on the outermost position. As all immediate contracts on values will be reduced, immediate contracts only remain on non-value terms S_{NonVal}.

Canonical Terms

$$
\begin{array}{lcl}
S_{\mathsf{Val}} & ::= & K \mid \lambda x.S \quad \textbf{where} \quad \lambda x.S \neq \lambda x.\mathcal{B}[\![x @^{\flat}_{\iota} I]\!] \\
S_{\mathsf{NonVal}} & ::= & x \mid B^{\flat} \\
 & \mid & T_{\mathsf{I}}\, T_{\mathsf{Q}} \mid K\, T_{\mathsf{Q}} \mid S_{\mathsf{Val}}\, T_{\mathsf{I}} \mid S_{\mathsf{Val}}\, S \\
 & \mid & op(\overrightarrow{T_{\mathsf{NonQ}}}) \\
 & \mid & \mathit{if}\, T_{\mathsf{Q}}\, \mathcal{A}[\![T_{\mathsf{Q}}']\!]\, \mathcal{A}[\![T_{\mathsf{Q}}'']\!] \quad \textbf{where not} \quad T_{\mathsf{Q}}' = T' @^{\flat}_{\iota} C \textbf{ and} \\
 & & T_{\mathsf{Q}}'' = T'' @^{\flat}_{\iota} C \\
 & & \\
T_{\mathsf{I}} & ::= & S_{\mathsf{NonVal}} \mid S_{\mathsf{NonVal}} @^{\flat}_{\iota} I \\
 & \mid & \mathcal{A}[\![T_{\mathsf{I}} @^{\flat_0}_{\iota_0} I]\!] @^{\flat_1}_{\iota_1} J \quad \textbf{where} \quad I \not\sqsubseteq^* J, J \not\sqsubseteq^* I \\
 & & \\
T_{\mathsf{Q}} & ::= & S_{\mathsf{Val}} \mid T_{\mathsf{I}} \mid S_{\mathsf{Val}} @^{\flat}_{\iota} Q \mid T_{\mathsf{I}} @^{\flat}_{\iota} Q \\
 & \mid & \mathcal{A}[\![T_{\mathsf{Q}} @^{\flat_0}_{\iota_0} Q]\!] @^{\flat_1}_{\iota_1} R \quad \textbf{where} \quad Q \not\sqsubseteq^* R, Q \not\sqsubseteq^* R \\
 & & \textbf{and not } Q \neq C \to \top, R \neq \top \to D, \flat_0 = \flat_1 \\
 & & \\
T & ::= & T_{\mathsf{Q}} \mid (T_r \parallel T_l) \mid B^{\flat} @^{\flat}_{\iota} \bot
\end{array}
$$

Non-Canonical Terms

$$
\begin{array}{lcl}
O, P & += & \mathcal{A}[\![M @^{\flat_0}_{\iota_0} C]\!] @^{\flat_1}_{\iota_1} D \quad \textbf{where} \quad C \not\sqsubseteq^* D, D \not\sqsubseteq^* C \\
 & \mid & \mathcal{A}[\![M @^{\flat}_{\iota_0} C]\!] @^{\flat}_{\iota_1} D \quad \textbf{where} \quad C \not\sqsubseteq D, D \not\sqsubseteq C \\
 & \mid & \lambda x.\mathcal{B}[\![x @^{\flat}_{\iota} I]\!] \\
 & \mid & \mathit{if}\, L\, (M @^{\flat}_{\iota} C)\, (N @^{\flat}_{\iota} C) \\
 & \mid & M @^{\flat}_{\iota} \bot
\end{array}
$$

Figure 20. Canonical and non-canonical terms of the *Subset Transformation*.

In addition to the canonical terms, Figure 3 shows the syntax of transformable (non-canonical) terms O, which is the complement of the canonical terms T.

A non-canonical term O is either a term that contains a non-canonical term or one of the patterns addressed by our transformation rules in Section 4.3.

Lemma 5 (Terms)**.** *For all terms M it holds that M is either a canonical term T or a transformable (non-canonical) term O.*

11.2 Canonical Term of the Subset Transformation

Figure 4 shows the definition of canonical term T and source terms S as a restriction of the already defined canonical terms. For readability, we reuse the same symbols as in Section **??**. The repeatedly defined terms replace the already existing definition, whereas every other definition remains valid.

A lambda abstractions $\lambda x.S$ now did not contain immediate contracts on its argument x and terms only contain contracts that are not subsumed by another contract on that term. Terms T now contains a blame terms with a remaining $\bot$ contract.

Lemma 6 (Terms)**.** *For all terms M it holds that M is either a canonical term T or a transformable (non-canonical) term O.*

$$\mathcal{K} ::= \square \mid \kappa : \mathcal{K}$$

Figure 21. Constraint context.

$$\textit{root-of}(b,\,\varsigma) = \begin{cases} \flat, & b = \flat \\ \textit{root-of}(b',\,\varsigma), & b = \iota, \mathcal{K}[\![(b' \blacktriangleleft \iota) : \varsigma]\!] \\ \textit{root-of}(b',\,\varsigma), & b = \iota_0, \mathcal{K}[\![(b' \blacktriangleleft \iota_0 \rightarrow \iota_1) : \varsigma]\!] \\ \textit{root-of}(b',\,\varsigma), & b = \iota_1, \mathcal{K}[\![(b' \blacktriangleleft \iota_0 \rightarrow \iota_1) : \varsigma]\!] \\ \textit{root-of}(b',\,\varsigma), & b = \iota_0, \mathcal{K}[\![(b' \blacktriangleleft \iota_0 \cap \iota_1) : \varsigma]\!] \\ \textit{root-of}(b',\,\varsigma), & b = \iota_1, \mathcal{K}[\![(b' \blacktriangleleft \iota_0 \cap \iota_1 : \varsigma]\!] \\ \textit{root-of}(b',\,\varsigma), & b = \iota_0, \mathcal{K}[\![(b' \blacktriangleleft \iota_0 \cup \iota_1) : \varsigma]\!] \\ \textit{root-of}(b',\,\varsigma), & b = \iota_1, \mathcal{K}[\![(b' \blacktriangleleft \iota_0 \cup \iota_1) : \varsigma]\!] \\ \textit{root-of}(b',\,\varsigma), & b = \iota, \mathcal{K}[\![(b' \blacktriangleleft \neg\iota) : \varsigma]\!] \\ b, & \textit{otherwise} \end{cases}$$

Figure 22. Function *root-of*.

$$\neg B = \begin{cases} \textit{-blame}, & B = \textit{+blame} \\ \textit{+blame}, & B = \textit{-blame} \end{cases}$$

Figure 23. Blame inversion.

$$\textit{sign-of}(b,\,\varsigma) = \begin{cases} \textit{+blame}, & b = \flat \\ \textit{sign-of}(b',\,\varsigma), & b = \iota, \mathcal{K}[\![(b' \blacktriangleleft \iota) : \varsigma]\!] \\ \neg\textit{sign-of}(b',\,\varsigma), & b = \iota_0, \mathcal{K}[\![(b' \blacktriangleleft \iota_0 \rightarrow \iota_1) : \varsigma]\!] \\ \textit{sign-of}(b',\,\varsigma), & b = \iota_1, \mathcal{K}[\![(b' \blacktriangleleft \iota_0 \rightarrow \iota_1) : \varsigma]\!] \\ \textit{sign-of}(b',\,\varsigma), & b = \iota_0, \mathcal{K}[\![(b' \blacktriangleleft \iota_0 \cap \iota_1) : \varsigma]\!] \\ \textit{sign-of}(b',\,\varsigma), & b = \iota_1, \mathcal{K}[\![(b' \blacktriangleleft \iota_0 \cap \iota_1 : \varsigma]\!] \\ \textit{sign-of}(b',\,\varsigma), & b = \iota_0, \mathcal{K}[\![(b' \blacktriangleleft \iota_0 \cup \iota_1) : \varsigma]\!] \\ \textit{sign-of}(b',\,\varsigma), & b = \iota_1, \mathcal{K}[\![(b' \blacktriangleleft \iota_0 \cup \iota_1) : \varsigma]\!] \\ \neg\textit{sign-of}(b',\,\varsigma), & b = \iota, \mathcal{K}[\![(b' \blacktriangleleft \neg\iota) : \varsigma]\!] \end{cases}$$

Figure 24. Function *sign-of*.

12. Auxiliary Functions and Definitions

This section shows some auxiliary functions and definitions used in the paper.

12.1 Function *root-of*$(b,\,\varsigma)$

Each top-level contract assertion is rooted in one blame label $\flat$, i.e. for every blame identifier b there exists a path from b to a blame label $\flat$.

To computer the root label $\flat$ for an identifier b we need to walk backwards through all constraints κ on a path until we reach the top level blame label $\flat$.

To do so, Figure 5 introduces a constraint context $\mathcal{K}$, which is either a hole $\square$ or constraint κ extending a constraint context $\mathcal{K}$.

Figure 6 finally shows the computation of function *root-of*$(b,\,\varsigma)$.

12.2 Sign of a Blame Identifier

Following the definition in Section 3, constraints κ implement a tree rooted in a blame label $\flat$. Within the tree every blame identifier b can report positive or negative blame, depending on its constraint when mapping one of its truth values (ι*.subject* or ι*.context*) to false.

Thus, we define the *sign* of a blame identifier b as the responsibility to satisfy the subject part in this identifier b. We represent signs by blame term B that would result from a subject blame in b.

Definition 7. *B is the sign of a blame identifier b in ς if $(b \blacktriangleleft$ false$) : \varsigma$ leads to $B^\flat$ on a single path for any $\flat$.*

In this case, every alternative must be treated as neutral, i.e. they do not stop or change the arising blame. Thus we consider only a single path from b to its root $\flat$.

To compute the sign of a blame identifier b, Figure 7 first defines an inversion of blame terms B that flips the responsibility. Figure 8 than shows the computation of the sign of blame identifier b. It is defined by induction and tracks swapping of responsibilities on function and inversion constraints.

12.3 Computation of Blame Terms

To computes a blame term $B^\flat$ resulting from a predicate violation ($\iota \blacktriangleleft$ *false*) in ι we need to compute the sign of ι in ς and to annotate the resulting blame term with the root blame identifier $\flat$ of ι in ς.

$$\frac{B = \textit{sign-of}(b,\,\varsigma) \qquad \flat = \textit{root-of}(b,\,\varsigma)}{\eta(\iota,\,\varsigma) = B^\flat}$$

12.4 Compute joined Assertion Context

We define the join of two assertion contexts $\mathcal{A}$ and $\mathcal{A}'$ by

$$\mathcal{A} \sqcup \mathcal{A}' = \mathcal{A}[\![\mathcal{A}' \setminus \mathcal{A}]\!]$$

Term $\mathcal{A}' \setminus \mathcal{A}$ computes a new context with assertions from $\mathcal{A}'$ that are not already contained in $\mathcal{A}$. Its computation is defined as:

$$\mathcal{A} \setminus \mathcal{A}' = \begin{cases} \mathcal{A}'' \setminus \mathcal{A}', & \mathcal{A} = \mathcal{A}''@^\flat_\iota C \wedge \square@^\flat_\iota C \in \mathcal{A}' \\ (\mathcal{A}'' \setminus \mathcal{A}')@^\flat_\iota C, & \mathcal{A} = \mathcal{A}''@^\flat_\iota C \wedge \square@^\flat_\iota C \notin \mathcal{A}' \\ \mathcal{A}, & \textit{otherwise} \end{cases}$$

We write $\square@^\flat_\iota C \in \mathcal{A}$ to indicate that the assertion of C under blame label $\flat$ and blame variable ι is contained in $\mathcal{A}$, and $\square@^\flat_\iota C \notin \mathcal{A}$ for its negation. Contract entailment is defined by:

$$\square@^\flat_\iota C \in \mathcal{A}@^\flat_\iota C \qquad\qquad \frac{\square@^\flat_\iota C \in \mathcal{A}}{\square@^\flat_\iota C \in \mathcal{A}@^\flat_\iota D}$$

PUSH/IMMEDIATE

$$\varsigma,\ \mathcal{F}[\![(T @^{\flat}_{\iota_1} Q) @^{\flat}_{\iota_2} I]\!] \longmapsto_{\mathsf{B}} \varsigma\, \mathcal{F}[\![(T @^{\flat}_{\iota_2} (I \hookleftarrow (\square @^{\flat}_{\iota_1} Q))) @^{\flat}_{\iota_1} Q]\!]$$

Figure 25. Changes for Picky Semantics.

EXTEND/PREDICATE

$$\frac{M = \lambda x.N}{\varsigma,\ \mathcal{F}[\![\mathit{flat}(M) \hookleftarrow (\square @^{\flat}_{\iota} Q)]\!] \longmapsto_{\mathsf{B}} \varsigma,\ \mathcal{F}[\![\lambda x.N[x \mapsto (x @^{\flat}_{\iota} Q)]]\!]}$$

EXTEND/NON-PREDICATE

$$\frac{M \neq \lambda x.N}{\varsigma,\ \mathcal{F}[\![\mathit{flat}(M) \hookleftarrow (\square @^{\flat}_{\iota} Q)]\!] \longmapsto_{\mathsf{B}} \varsigma,\ \mathcal{F}[\![\mathit{flat}(M)]\!]}$$

Figure 26. Unroll delayed contract.

13. Picky Semantics

The semantics in Section 4 guarantee *Strong Blame-preservation* with respect to Lax evaluation semantics (cf. Section 3.5). Lax unpacks contracts from values that flow to a predicate, i.e. a contract cannot violate another contract.

The *Baseline Transformation* makes this intuitively when processing the rule PUSH/IMMEDIATE. It pushes a delayed contract out of an assertion with an immediate contract, such that the predicate will not see the delayed contract. This step is entirely required for following transformation steps.

When considering Picky (or even Indy) evaluation semantics, a contract must remain on a value when the value is used in a predicate. Consequently, we cannot pushes a delayed contract out of an assertion with an immediate contract without changing the blame behaviour of the underlying program.

To enables Picky (or Indy) semantics, rule PUSH/IMMEDIATE (Figure 8) must be replayed by another rule that unrolls a delayed contract in an immediate contract when pushing the delayed contract. Figure 9 shows the new rule.

Here, term $T @^{\flat}_{\iota} (I \hookleftarrow (\square @^{\flat}_{\iota'} Q))$ is a new intermediate term that unrolls a delayed contract in a predicate (cf. rule UNROLL). Figure 10 shows its semantics.

Rule EXTEND/PREDICATE unrolls a delayed contract to all uses of the value in a predicate and rule EXTEND/PREDICATE proceeds on the unchanged predicate in case that the predicate is not a lambda abstraction.

Indeed, it would also be possible to replace predicate M by $M @^{\flat}_{\iota} Q \rightarrow \top$ as it enforces Q when the predicate is used in an application. However, unrolling the contract enables to apply further simplification steps in the function's body.

14. Proofs of Theorems

14.1 Proof of Lemma 1

Proof of Lemma 1. Immediate from the definition of canonical and non-canonical terms (Section B.1). □

14.2 Proof of Lemma 4

Proof of Lemma 4. $\simeq$ is an equivalence relation if and only if it is reflexive, symmetric, and transitive. Proof by induction over $\simeq$. □

14.3 Proof of Conjecture 1

14.4 Congruence of States and Terms

Definition 8. *Two configurations (ς_M, M) and (ς_N, N) are structural equivalent under renaming α, written $\varsigma_M, M \cong_\alpha \varsigma_N, N$ with $\alpha = \emptyset \mid \alpha[\iota \mapsto \iota']$ iff $\varsigma_M, M \cong_{\alpha'} (\varsigma_N, N)[\iota \mapsto \iota']$ with $\alpha = \alpha'[\iota \mapsto \iota']$.*

Lemma 7. *For all configurations (ς_M, M) and (ς_N, N) with $\varsigma_M, M \cong \varsigma_N, N$ it holds that $\varsigma_M, M \longrightarrow \varsigma'_M, M'$ and $\varsigma_N, N \longrightarrow \varsigma'_N, N'$ with $\varsigma'_M, M' \cong \varsigma'_N, N'$.*

Proof of Theorem 3. Proof by induction over ς and M. □

14.5 Congruence of States and Terms

Definition 9. *State ς' is an effect-free extension of state ς, written $\varsigma \preccurlyeq \varsigma'$, iff $\varsigma, M \longrightarrow \varsigma'', M'$ and $\varsigma', M \longrightarrow \varsigma''', M'$ with $\varsigma'' \preccurlyeq \varsigma'''$.*

Lemma 8. *For terms M, states ς, and transformations $\varsigma, M \Longmapsto \varsigma', N$ it hold that $\varsigma \preccurlyeq \varsigma'$.*

Proof of Theorem 3. Immediate by the definition of $\longrightarrow$ and $\Longmapsto$. Every new constrains is composed of fresh blame variable ι. Proof by induction over $\Longmapsto$. □

Proof Sketch for Conjecture 1. Let reduction output $O ::= V \mid B^\flat$. For all terms M and states ς_M with $\varsigma_M, M \longmapsto_{\mathsf{B}} \varsigma_N, N$ it hold that $\varsigma_M, M \Longrightarrow \varsigma'_M, O$ and $\varsigma_N, N \Longrightarrow \varsigma'_N, O$ with $\varsigma'_M \cong_\alpha \varsigma'_N$.

As our transformation only propagates contract assertions and did not modify the underlying program, we know that if $\varsigma, M \longrightarrow^* \varsigma', V$ and $\varsigma, N \longrightarrow^* \varsigma'', W$ then $V = W$.

To prove preservation, we show that the transformed term results in the same reduction sequence than the original term and that the both, state and terms, are equivalent under variable renaming I.

Proof by induction over $\longmapsto_{\mathsf{B}}$.

Rule UNFOLD/ASSERT: $M = T@^\flat C$, $N = T@^\flat_\iota C$, and $\varsigma_N = (\flat \blacktriangleleft \iota') : \varsigma_M$.

(1) From M and $\Longrightarrow$ it follows that $\varsigma_M, T@^\flat C \longrightarrow^* \varsigma'_M, V@^\flat C$ and by rule ASSERT it follows that $\varsigma'_M, V@^\flat C \longrightarrow (\flat \blacktriangleleft \iota) : \varsigma'_M, V@^\flat_\iota C$.

(2) From lemma 3 it follows the $\varsigma_N, T@^\flat_{\iota'} C \longrightarrow^* \varsigma'_N, V@^\flat_{\iota'} C$ with $\varsigma'_M \preccurlyeq \varsigma'_N$.

Claim holds because $(\flat \blacktriangleleft \iota) : \varsigma'_M, V@^\flat_\iota C \cong_\alpha \varsigma'_N, V@^\flat_{\iota'} C$ with $\alpha = \emptyset[\iota \mapsto \iota']$.

Rule UNFOLD/UNION: Equivalent to case UNFOLD/ASSERT.

Rule UNFOLD/INTERSECTION: Equivalent to case UNFOLD/ASSERT.

Rule UNFOLD/D-FUNCTION: $M = (T_1@^\flat_\iota(C \to D))\, T_2$, $N = (T_1\,(T_2@^\flat_{\iota_1}C))@^\flat_{\iota_2}D$, and $\varsigma_N = (\iota \blacktriangleleft \iota'_1 \to \iota'_2) : \varsigma_M$.

(1) From M and $\Longrightarrow$ it follows that $\varsigma_M, (T_1@^\flat_\iota(C \to D)), T_2 \longrightarrow^* \varsigma'_M, (V_1@^\flat_\iota(C \to D)), T_2$ and $\varsigma'_M, (V_1@^\flat_\iota(C \to D)), T_2 \longrightarrow^* \varsigma''_M, (V_1@^\flat_\iota(C \to D)), V_2$. By rule D-FUNCTION it follows that $\varsigma''_M, (V_1@^\flat_\iota(C \to D)), V_2 \longrightarrow (\iota \blacktriangleleft \iota_1 \to \iota_2) : \varsigma''_M, (V_1\,(V_2@^\flat_{\iota_1}C))@^\flat_{\iota_2}D$.

(2) From lemma 3, N and $\Longrightarrow$ it follows that $\varsigma_N, (T_1\,(T_2@^\flat_{\iota'_1}C))@^\flat_{\iota'_2}D \longrightarrow^* \varsigma'_N, (V_1\,(T_2@^\flat_{\iota'_1}C))@^\flat_{\iota'_2}D$ and $\varsigma'_N, (V_1\,(T_2@^\flat_{\iota'_1}C))@^\flat_{\iota'_2}D \longrightarrow^* \varsigma''_N, (V_1\,(V_2@^\flat_{\iota'_1}C))@^\flat_{\iota'_2}D$ with $\varsigma''_M \cong_\alpha \varsigma''_N$.

Claim holds because $(\iota \blacktriangleleft \iota_1 \to \iota_2) : \varsigma''_M, (V_1\,(V_2@^\flat_{\iota_1}C))@^\flat_{\iota_2}D \cong_\alpha \varsigma''_N, (V_1\,(V_2@^\flat_{\iota'_1}C))@^\flat_{\iota'_2}D$ with $\alpha = \emptyset[\iota_1 \mapsto \iota'_2][\iota_2 \mapsto \iota'_2]$.

Rule UNFOLD/D-INTERSECTION: Equivalent to case UNFOLD/D-FUNCTION.

Rule UNROLL: Immediate from rule BETA.

Rule LOWER: Immediate from rule D-FUNCTION.

Rule PUSH/IMMEDIATE: Immediate from rule FLAT.

Rule PUSH/FALSE: Immediate from rule FLAT.

Rule PUSH/IF: Immediate from rules TRUE and FALSE.

Rule CONVERT/TRUE: Immediate from rule UNIT.

□

14.6 Proof of Lemma 2

Proof of Lemma 2. $\sqsubseteq^*$ is a preorder if and only if it is reflexive and transitive. Proof by induction over $\sqsubseteq^*$. □

14.7 Proof of Lemma 3

Proof of Lemma 3. Immediate from the definition of canonical and non-canonical terms (Section B.2). □

14.8 Proof of Conjecture 2

Proof Sketch for Conjecture 2. Equivalent to proof of Conjecture 1. However, certain rules change the order of contract checks. So wen need to consider two cases: one where the early contract is satisfied, and one where it is violated. □

14.9 Proof of Theorem 1

Proof of Theorem 1. Proof by induction over $\Longmapsto$. It is immediate from the definition that all rules, expect rule UNROLL, either decompose, eliminate, or graft a contract. It remains to shows that:

1. no rules duplicates contract checks, and
2. no rule pushes contract checks from one function body (or branch in a condition) to the enclosing body.

Rule UNROLL . This rule grafts the assertion of a delayed contract to all uses of the contracted value. From $\longrightarrow$, we have that for all $\lambda x.M\,(N@^\flat_\iota Q)$ first $N \longrightarrow^* V$ and by rule BETA it follows that $\lambda x.M\,(V@^\flat_\iota Q)$ reduces to $\lambda x.M[x \mapsto (V@^\flat_\iota Q)]$. By rule UNROLL we have that $\lambda x.M\,(N@^\flat_\iota Q)$ translates to $\lambda x.M[x \mapsto (x@^\flat_\iota Q)]\, N$. From $\longrightarrow$, we now have that $N \longrightarrow^* V$ and by rule BETA if follows that $\lambda x.M[x \mapsto (x@^\flat_\iota Q)][x \mapsto V]$, which is equivalent to the result of the original reduction.

Rule LIFT . This rule lifts an immediate contract nested in a function body to a new function contract. However, this contract never applies if the function is not executed. Thus, it preserves the number contract checks.

Rule LOWER . This rule lowers a contract to the function boundary. But, this contract never applies if the function is not executed. It also preserves the number contract checks.

Rule PUSH/IF . This rule pushes an immediate contract out of a condition, if the contract definitely applies to both branches. Thus, it preserves the number contract checks.

Otherwise Every other rule either decomposes or eliminates a contract, or reverses the order of contract checks. Thus they reduces or preserves the total number contract checks at run time.

□

Proof of Theorem 2. The transformation terminates with a canonical term because (1) there are no cycles and (2) there is no transformation step that introduces new contract check.

1. All rules, except rule UNROLL, decompose or eliminate a contract, or grafts a contract to an enclosing term. Only rule UNROLL duplicates a contract to all uses of the contracted value. However, as the transformation did not lift delayed contracts, it is not a cycle.
2. Immediate from Theorem 1.

□

SIMPLIFY/UNION/1

$$\frac{C \sqsubseteq D}{\varsigma,\, \mathcal{F}[\![V @^{\flat} (C \cap D)]\!] \longmapsto_{\mathsf{B}} \varsigma,\, \mathcal{F}[\![V @^{\flat} D]\!]}$$

SIMPLIFY/UNION/2

$$\frac{D \sqsubseteq C}{\varsigma,\, \mathcal{F}[\![V @^{\flat} (C \cap D)]\!] \longmapsto_{\mathsf{B}} \varsigma,\, \mathcal{F}[\![V @^{\flat} C]\!]}$$

SIMPLIFY/INTERSECTION/1

$$\frac{C \sqsubseteq D}{\varsigma,\, \mathcal{F}[\![V @^{\flat} (C \cap D)]\!] \longmapsto_{\mathsf{B}} \varsigma,\, \mathcal{F}[\![V @^{\flat} C]\!]}$$

SIMPLIFY/INTERSECTION/2

$$\frac{D \sqsubseteq C}{\varsigma,\, \mathcal{F}[\![V @^{\flat} (C \cap D)]\!] \longmapsto_{\mathsf{B}} \varsigma,\, \mathcal{F}[\![V @^{\flat} D]\!]}$$

Figure 27. Contract simplification.

15. Simplify Intersection and Union

Apart from the subset optimization, which reduces a contract assertion based on more restrictive assertion, we can also use the subcontract relation to simplify contracts before unfolding.

For example, the intersection contract *Positive?* $\rightarrow$ *Natural?* $\cap$ *Natural?* $\rightarrow$ *Positive?* can be simplified to a single function contract *Positive?* $\rightarrow$ *Positive?*. This is because, the context can choose to call the function with a natural or positive number. However, in case the input is a natural number, but not a positive number, than the range must be a positive number. And, in case the input is a positive number, than it is also a natural number (as positive numbers are a proper subset of natural numbers) and thus the output needs to satisfy both range contract, i.e. it must be a positive number.

So, we can simplify the intersection because *Natural?* $\rightarrow$ *Positive?* is an ordinary subset of *Positive?* $\rightarrow$ *Natural?* (cf. Section 5.2). Its domain is less or equal restrictive and its range is more or equal restrictive.

In general, we can simplify intersection and union contracts if one operand is an ordinary subset of the other operand. Figure 11 shows some simplification rules that omits one branch of an alternative if that alternative is subsumed by the other branch.

To keep the system deterministic, we must simplify contracts at assertion time and thus we skip rule UNFOLD/ASSERT in this cases.

$$
\begin{array}{lcl}
S_{\mathsf{Val}} & ::= & K \mid \lambda x.S \quad \textbf{where} \quad \lambda x.S \neq \lambda x.\mathcal{B}[\![x@^{\flat}_{\iota}C]\!] \\
S_{\mathsf{NonVal}} & ::= & x \mid B^{\flat} \\
& \mid & T_{\mathsf{I}}\,T_{\mathsf{Q}} \mid K\,T_{\mathsf{Q}} \mid S_{\mathsf{Val}}\,T_{\mathsf{I}} \mid S_{\mathsf{Val}}\,S \\
& \mid & op(\vec{T_{\mathsf{I}}}) \mid op(\vec{S_{\mathsf{Val}}}) \\
& \mid & \mathit{if}\,T_{\mathsf{Q}}\,\mathcal{A}[\![T'_{\mathsf{Q}}]\!]\,\mathcal{A}[\![T''_{\mathsf{Q}}]\!] \quad \textbf{where not} \quad T'_{\mathsf{Q}} = T'@^{\flat}_{\iota}C \textbf{ and} \\
& & T''_{\mathsf{Q}} = T''@^{\flat}_{\iota}C \\
\\
T_{\mathsf{I}} & ::= & S_{\mathsf{NonVal}} \mid S_{\mathsf{NonVal}}@^{\flat}_{\iota}I \\
& \mid & \mathcal{A}[\![T_{\mathsf{I}}@^{\flat_0}_{\iota_0}I]\!]@^{\flat_1}_{\iota_1}J \quad \textbf{where} \quad I \not\sqsubseteq^* J, J \not\sqsubseteq^* I \\
\\
T_{\mathsf{Q}} & ::= & S_{\mathsf{Val}} \mid T_{\mathsf{I}} \mid S_{\mathsf{Val}}@^{\flat}_{\iota}Q \mid T_{\mathsf{I}}@^{\flat}_{\iota}Q \\
& \mid & \mathcal{A}[\![T_{\mathsf{Q}}@^{\flat_0}_{\iota_0}Q]\!]@^{\flat_1}_{\iota_1}R \quad \textbf{where} \quad Q \not\sqsubseteq^* R, Q \not\sqsubseteq^* R \\
& & \textbf{and not } Q \neq C \to \top, R \neq \top \to D, \flat_0 = \flat_1 \\
\\
T & ::= & T_{\mathsf{Q}} \mid (T_r \parallel T_l)
\end{array}
$$

Figure 28. Canonical terms and contexts.

LIFT

$$
\frac{\iota_1 \notin \varsigma \qquad x \in (\mathit{free?}\,\mathcal{G}[\![x]\!])}{\varsigma,\, \mathcal{F}[\![\lambda x.\mathcal{G}[\![x@^{\flat}_{\iota}C]\!]]\!] \;\longmapsto'_{\mathsf{A}}\; (\iota \blacktriangleleft \neg\iota_1) : \varsigma,\, \mathcal{F}[\![(\lambda x.\mathcal{G}[\![x]\!])@^{\flat}_{\iota_1}C \to \top]\!]}
$$

BLAME

$$
\varsigma,\, \mathcal{F}[\![T@^{\flat}_{\iota}\bot]\!] \;\longmapsto_{\mathsf{A}}\; (\iota \blacktriangleleft \mathit{false}) : \varsigma,\, \mathcal{F}[\![T]\!]
$$

SUBSET

$$
\frac{\varsigma,\, M \;\longmapsto'_{\mathsf{S}}\; \varsigma',\, N}{\varsigma,\, M \;\longmapsto'_{\mathsf{A}}\; \varsigma',\, N}
$$

TRACE

$$
\frac{\varsigma,\, M \;\longmapsto'_{\mathsf{A}}\; \varsigma',\, N}{\varsigma,\, \mathcal{T}[\![M]\!] \;\longmapsto_{\mathsf{A}}\; \varsigma',\, \mathcal{T}[\![N]\!]}
$$

Figure 29. Approximations rules.

16. Introducing new Success Contracts

Our core rules in Section 4 and Section 5 follows the overall guideline not to approximate contract violations. However, developers are free to choose a higher degree of optimization, while over-approximating contract failures at compile time and introducing new module boundaries.

This level's rules create some kind of success contracts by over-approximating contracts and by lifting contracts to the upper-most boundary. However, this is not a real optimization as it might lift checks that never happen at run time.

Figure 12 defines the syntax of our canonical terms as a subset of λ_{Con} term M, and Figure 13 shows possible transformation.

Rule LIFT lifts every contract on an argument x. This step over-approximates contract violations as it also lifts contracts nested in conditions and function applications that might never apply.

Rule BLAME immediately updates the constraint state with information about a failing contract. This might result in a blame state, even if the ill behaved term is never executed.

Finally, the rule SUBSET lifts to the *Subset Reduction* $\longmapsto_{\mathsf{S}}$ and rule TRACE unpacks parallel observation. Obviously, rule BLAME replaces the rule with the same name in $\longmapsto_{\mathsf{S}}$ and we are never allowed to unroll a once lifted contract. Otherwise we would end up in a cycle.

17. Propagating Blame

In the preceding example we have seen that all contracts of a failing alternative can be removed and only one ⊥ must remain on the first failing term. As we split alternatives in different observation where every contract must be fulfilled, we can report a violation for that branch immediately when observing a violation. Every other contract can be ignored as this branch definitely runs into an error.

However, this is not completely true. We are only allowed to remove contracts in the same context as the violation occurs. We cannot treat a contract as violated if the violation only occurs in a certain aspect of the program.

For example, consider the following source program.

```
let f =
((λ addOne (λ z (addOne z)))
 ((λ plus (λ z ((plus 0) z)))
  [(λ x (λ y (+ x y)))
   @ (Positive? → (Positive? → Positive?))]))
```

After applying some simplification steps, we obtain the following intermediate term.

```
let f =
((λ addOne (λ z (addOne z)))
 ((λ plus (λ z ([(plus [0 @ ⊥]) @ (Positive? → Positive?)] z)))
  (λ x (λ y (+ x y)))))
```

Even though we do not know that *addOne* violates the contract on *plus*, we are not allowed treat the contract as violated because we do not know if *addOne* is ever executed. But, we can replace *addOne*'s body by a blame term that reports a violation whenever the function is executed. In addition, a ⊥ remains on the functions body.

```
let f =
((λ addOne (λ z (addOne z)))
 ((λ plus (λ z ([blame @ ⊥])))
  (λ x (λ y (+ x y)))))
```

Having a *blame* term and ⊥ seems to be twofold, but ⊥ can be lowerd to a new function contract on *addOne*, which is than unrolled to all places where *addOne* is used, as the following example demonstrates.

```
let f =
((λ addOne (λ z ([addOne @ (⊤ → ⊥)] z)))
 ((λ plus (λ z blame)) (λ x (λ y (+ x y)))))
```

Now, the same transformation steps apply as for normal contract. The function contract gets unrolled, whereby ⊥ gets asserted to another term. If it so happens the next function body gets transformed to a blame term with a remaining ⊥.

Continuing in this way propagates the information of a failing contact through applications and conditions until it reaches the outermost boundary. Within this boundary we definitely run into a contract violation. The following example shows the result.

```
let f =
[((λ addOne (λ z (addOne z)))
  ((λ plus (λ z blame)) (λ x (λ y (+ x y))))) @ (⊤ → ⊥)]
```

Benchmark	Full	Baseline
Richards	1 day, 18 hours, 21 min, 20 sec	8 sec
DeltaBlue	2 days, 10 hours, 36 min, 49 sec	4 sec
Crypto	9 sec	8 sec
RayTrace	23 hours, 12 min, 37 sec	4 sec
EarleyBoyer	1 min, 9 sec	53 sec
RegExp	9 sec	8 sec
Splay	19 min, 19 sec	3 sec
SplayLatency	19 min, 19 sec	3 sec
NavierStokes	11 sec	4 sec
pdf.js	6 sec	6 sec
Mandreel	5 sec	5 sec
MandreelLatency	5 sec	5 sec
Gameboy Emulator	4 hours, 28 min, 28 sec	5 sec
Code loading	12 sec	9 sec
Box2DWeb	5 hours, 19 min, 49 sec	6 sec
zlib	11 sec	11 sec
TypeScript	22 min, 46 sec	24 sec

Figure 30. Timings from running the Google Octane 2.0 Benchmark Suite. Column **Full** shows the time required to complete with contract assertions. The last column **Baseline** gives the baseline time without contract assertion.

Benchmark	Asssert	Internal	Predicate
Richards	24	1599377224	935751200
DeltaBlue	54	2319477672	1340451212
Crypto	1	5	3
RayTrace	42	687240082	509234422
EarleyBoyer	3944	89022	68172
RegExp	0	0	0
Splay	10	11620663	7067593
SplayLatency	10	11620663	7067593
NavierStokes	51	48334	39109
pdf.js	3	15	9
Mandreel	7	57	28
MandreelLatency	7	57	28
Gameboy Emulator	3206	141669753	97487985
Code loading	5600	34800	18400
Box2DWeb	20075	172755100	11266494
zlib	0	0	0
TypeScript	4	12673644	8449090

Figure 31. Statistic from running the Google Octane 2.0 Benchmark Suite. Column **Assert** shows the numbers of top-level contract assertions. Column **Internal** contains the numbers of internal contract assertions whereby column **Predicates** lists the number of predicate evaluations.

18. An Evaluation of Contract Monitoring

Dynamic contract checking impacts the execution time. Source of this impact is (1) that every contract extents the original source with additional checks that need to checked when the program executes and (2) that the contract monitor itself causes some runtime overhead.

To demonstrate, we consider runtime values obtained from the TreatJS contract framework for JavaScript (Keil and Thiemann 2015b). To the best of our knowledge, TreatJS is the contract framework with the heaviest runtime deterioration. Reason for this is its flexibility and its support for full intersection and union contracts (Keil and Thiemann 2015a). The presence of intersection and union contracts require that a failing contract must not signal a violation immediately. As a violation depends on combinations of failures in different sub-contracts the contract monitor muss continue even if a first error occurs and connect the outcome of each sub-contract with the enclosing operations.

To evaluate the runtime deterioration, they applied there framework to benchmark programs from the Google Octane 2.0 Benchmark Suite[4]. The benchmark programs measure a JavaScript engine's performance by running a selection of complex and demanding programs.

For testing, they inferred a function contract for every function expression contained a program and assert the function contract when the program executes.

Figure 14 contains the runtime values of all benchmark programs in two configurations, which are explained in the figure's caption. As expected, all run times increase when adding contracts.

The examples show that the run-time impact of contract monitor depends on the program and on the particular value that is monitored. While some programs like *Richards*, *DeltaBlue*, *RayTrace*, and *Splay* are heavily affected, others are almost unaffected: *Crypto*, *NavierStokes*, and *Mandreel*, for instance.

It follows that the impact of a contract depends on the frequency of its application. A contract on a heavily used function (e.g., in *Richards*, *DeltaBlue*, or *Splay*) causes a significantly higher deterioration of the run-time.

For better understanding, Figure 15 lists some numbers of internal counters. The numbers indicate that the heavily affected benchmarks (*Richards*, *DeltaBlue*, *RayTrace*, *Splay*) contain a very large number of predicate checks. Other benchmarks are either not affected (*RegExp*, *zlib*) or only slightly affected (*Crypto*, *pdf.js*, *Mandreel*) by contracts.

For example, the *Richards* benchmark performs 24 top-level contract assertions (this are all unique contracts in a source program), 1.6 billion internal contract assertions (including top-level assertions, *delayed* contract checking, and predicate evaluation), and 936 million predicate evaluation.

To sum up, we see that number of predicate checks is substantially responsible for the runtime deterioration of a contract system. Thus, reducing the number of predicate checks will entirely improve the runtime.

[4] `https://developers.google.com/octane/`

19. Practical Evaluation, continued

To demonstrate the run time improvement we applied our contract simplification to a number simple examples, as introduces in Section 2 and Section 7. The testing procedure is a simple while loop that uses different versions of an *addOne* function to increase a counter on every iteration.

This section shows the examples programs written in JavaScript and the full table of results. To run the examples we use the TreatJS[5] (Keil and Thiemann 2015b) contract system for JavaScript and the Mozilla SpiderMonkey[6] engine. While there is actually no implementation of our transformation system for JavaScript, we applied all transformation manually and produces the simplified contracts by hand.

In TreatJS, function *assert* asserts a contrat (given as second argument) to a subject value (given as first argument). Constructor *AFunction* creates a function contract from a set of argument contracts (given as first argument) and a return contract (given as second argument). Constructor *Intersection* creates an intersection contract from two other contracts. *_Number*, *_Natural*, *_Positive*, *_Negative*, *_String* are flat contrasts checking for number values, natural numbers, positive numbers, negative numbers, and string values, respectively.

19.1 The example programs

Example 1 (*addOne1*). *In a first example we add a simple function contract to plus restricting its domain and range to number values. Every use of addOne1 triggers three predicates checks.*

```
var addOne1 = (function () {
  var plus = assert(function (x, y) {
    return x + y;
  }, AFunction([_Number, _Number], _Number))
  var addOne = function (x) {
    return plus (x, 1);
  }
  return addOne;
})();
```

Example 2 (*addOne2*). *Our second example adds an intersection contract to plus. As the native + operator is overload and works for strings and numbers, our contract restricts the domain either to string or number values and ensures that the function has to return a string or a number value corresponding to the input. Every use of addOne2 triggers six predicate checks.*

```
var addOne2 = (function () {
  var plus = assert(function (x, y) {
    return x + y;
  }, Intersection(
      AFunction([_Number, _Number], _Number),
      AFunction([_String, _String], _String)
  ));
  var addOne = function (x) {
    return plus (x, 1);
  }
  return addOne;
})();
```

Example 3 (*addOne3*). *The third example extends addOne1 by adding another function contract to addOne. The contract restricts addOne's domain to natural numbers and requires a positive number as return. In combination, every use of addOne3 triggers 5 predicate checks.*

```
var addOne3 = (function () {
  var plus = assert(function (x, y) {
    return x + y;
  }, AFunction([_Number, _Number], _Number));
  var addOne =  assert(function (x) {
    return plus (x, 1);
  }, AFunction([_Natural], _Positive));
  return addOne;
})();
```

Example 4 (*addOne4*). *The next example merges the intersection contract on plus (from Example 2) and the contract on addOne (Example 3). Every call of addOne4 leads to a total number 8 predicate checks.*

```
var addOne4 = (function () {
  var plus = assert(function (x, y) {
    return x + y;
  }, Intersection(
      AFunction([_Number, _Number], _Number),
      AFunction([_String, _String], _String)
  ));
  var addOne = assert(function (x) {
    return plus (x, 1);
  }, AFunction([_Natural], _Positive));
  return addOne;
})();
```

Example 5 (*addOne5*). *In this example we also overload addOne by making either string concatenation or addition, depending on addOne's input. While adding an intersection contract to addOne, every use of addOne5 leads to 10 predicate checks.*

```
var addOne5 = (function () {
  var plus = assert(function (x, y) {
    return x + y;
  }, Intersection(
      AFunction([_Number, _Number], _Number),
      AFunction([_String, _String], _String)
  ));
  var addOne = assert(function (x) {
    return (typeof x == 'string') ? plus (x, '1') : plus (x, 1);
  }, Intersection(
       AFunction([_Natural], _Positive),
       AFunction([_String], _String)
  );
  return addOne;
})();
```

Example 6 (*addOne6*). *Our last example simulates the case that we add fine-grained properties to plus and addOne. In this case we state different properties in different function contracts and use intersections to combine those properties. Before simplifying the contract, every call of addOne6 leads to a total number 17 predicate checks.*

```
var addOne6 = (function () {
  var plus = assert(function (x, y) {
    return x + y;
  }, Intersection(
      Intersection(
        AFunction([_Number, _Number], _Number),
        AFunction([_String, _String], _String)),
```

[5] `https://github.com/keil/TreatJS`

[6] `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`

```
    Intersection(
      Intersection(
        AFunction([_Natural, _Positive], _Positive),
        AFunction([_Positive, _Natural], _Positive)),
      AFunction([_Negative, _Negative], _Negative))));
  var addOne = assert(function (x) {
    return plus (x, 1);
  }, AFunction([_Natural], _Positive));
  return addOne;
})();
```

19.2 The Run-Time Improvements

Figure 16 finally contains the execution time and the number of predicate checks during execution of all examples programs in different configuration, as explained in the figures caption.

In addition to the examples from Section J.1, it also contains the run time of the *addOne* program without contracts, and the run time of the *addOne* example where we only wrap the functions in a proxy, but did not apply any contracts.

To sum up, the the *Baseline Simplification* improves the run time by approximately 28%, whereas the *Subset Simplification* makes an improvement by approximately 58.68%. Clearly, the improvement strictly depends on the granularly of predicates and contracts.

The numbers also indicate that the run time improvements remain identical, whether we use an optimizing compiler or not. But, deactivating the optimizing compiler increases the run time of the program without contracts by factor 4.83 (*No-Ion*) and by factor 20.17 (*No-Jit*), whereas the version with contracts increases only by factor 1.5 (*No-Ion*) and factor 2.09 (*No-Jit*).

The numbers also shows that adding contracts, even without any functionality, increases the run time by factor 71.2 (*Full*), factor 13.7 (*No-Ion*), and factor 5 (*No-Jit*).

This illustrates that contracts prevent a program from being optimized efficiently by an optimizing compiler.

Full										
Benchmark	**Normal**		**Baseline**				**Subset**			
	time (ms)	*predicates*	*time (ms)*		*predicates*		*time (ms)*		*predicates*	
Without Contracts	2	0								
Proxy only	142.33	0								
Example 1	39404	300000	27081	(- 31.27%)	200000	(- 33.33%)	27043	(- 31.37%)	200000	(- 33.33%)
Example 3	87143	600000	58385	(- 33.00%)	400000	(- 33.33%)	46085	(- 47.12%)	300000	(- 50.00%)
Example 4	66474	500000	54396	(- 18.17%)	400000	(- 20.00%)	26518	(- 60.11%)	200000	(- 60.00%)
Example 4	114468	800000	85126	(- 25.63%)	600000	(- 25.00%)	44633	(- 61.01%)	300000	(- 62.50%)
Example 5	148249	1000000	107956	(- 27.18%)	800000	(- 20.00%)	59970	(- 59.55%)	500000	(- 50.00%)
Example 6	295682	1700000	200009	(- 32.36%)	1200000	(- 29.41%)	118579	(- 59.90%)	700000	(- 58.82%)

No-Ion										
Benchmark	**Normal**		**Baseline**				**Subset**			
	time (ms)	*predicates*	*time (ms)*		*predicates*		*time (ms)*		*predicates*	
Without Contracts	9.67	0								
Proxy only	132.67	0								
Example 1	58906	300000	40383	(- 31.44%)	200000	(- 33.33%)	40271	(- 31.64%)	200000	(- 33.33%)
Example 3	131184	600000	87544	(- 33.26%)	400000	(- 33.33%)	69492	(- 47.03%)	300000	(- 50.00%)
Example 4	99776	500000	81719	(- 18.10%)	400000	(- 20.00%)	40218	(- 59.69%)	200000	(- 60.00%)
Example 4	173037	800000	128754	(- 25.59%)	600000	(- 25.00%)	66359	(- 61.65%)	300000	(- 62.50%)
Example 5	221619	1000000	160360	(- 27.64%)	800000	(- 20.00%)	88369	(- 60.13%)	500000	(- 50.00%)
Example 6	441176	1700000	278601	(- 36.85%)	1200000	(- 29.41%)	163272	(- 62.99%)	700000	(- 58.82%)

No-Jit										
Benchmark	**Normal**		**Baseline**				**Subset**			
	time (ms)	*predicates*	*time (ms)*		*predicates*		*time (ms)*		*predicates*	
Without Contracts	40.33	0								
Proxy only	200.67	0								
Example 1	81125	300000	56439	(- 30.43%)	200000	(- 33.33%)	54945	(- 32.27%)	200000	(- 33.33%)
Example 3	186069	600000	124434	(- 33.12%)	400000	(- 33.33%)	96271	(- 48.26%)	300000	(- 50.00%)
Example 4	136728	500000	111596	(- 18.38%)	400000	(- 20.00%)	55186	(- 59.64%)	200000	(- 60.00%)
Example 4	240724	800000	179451	(- 25.45%)	600000	(- 25.00%)	91034	(- 62.18%)	300000	(- 62.50%)
Example 5	315184	1000000	225316	(- 28.51%)	800000	(- 20.00%)	123852	(- 60.71%)	500000	(- 50.00%)
Example 6	597406	1700000	404276	(- 32.33%)	1200000	(- 29.41%)	233124	(- 60.98%)	700000	(- 58.82%)

Figure 32. Results from running the TreatJS contract system. Table ***Full*** shows the results of a run with both JIT compilers, whereas table ***No-Ion*** shows the result of a run without IonMonkey (but the Baseline JIT remains enabled) and table ***No-JIT*** shows the result of a run without any JIT compilation. Column **Normal** gives the baseline execution time and the total number of predicate checks of the unmodified program. Column **Baseline** and column **Subset** contain the execution time and the total number of predicate checks after applying the baseline simplification or the subset simplification, respectively. The value in parentheses indicates the improvement (in percent).