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 =
((\(x\) \(\(\text{plus} ((\text{plus } 1) z))\))
\((\lambda x (\lambda y (+ x y)))\))
\((\text{Number} \to (\text{Number} \to \text{Number})) \cap
\text{String} \to (\text{String} \to \text{String}))\)
```

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.\footnote{Assuming the semantics of intersection and union contracts from the literature (Keil and Thiemann 2015a).} 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.

6 \textit{let addOne} =
7 \{(\lambda \mathbf{z} (\text{plus} 1 \mathbf{z}))\}
8 \{(\lambda \mathbf{x} (\lambda \mathbf{y} (+ \mathbf{x} \mathbf{y})))\} @ (\text{Number?} \rightarrow \text{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:

\begin{itemize}
    \item \textbf{Improvement} Each transformation step improves the efficiency of contract monitoring: it either \textit{reduces} or \textit{maintains} the total number of predicate checks at run time. Simplification never increases the number of run-time checks.
    \item \textbf{Strong Blame-Preservation} Each transformation step of the baseline simplification fully preserves the blame-behavior of a program. It maintains the order 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.
    \item \textbf{Weak Blame-Preservation} Transformation steps of the subset simplification may change the order 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.
\end{itemize}

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 guidelines from above.

\textbf{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.

\footnote{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 (Felkai et al. 2009). The implementation is available online at \url{http://anonymous}.  

\textbf{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 Optimization, the first tier in our simplification stack. Section 5 shows the second tier, Subset Optimization, 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 strings simplification.

\section{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.

\subsection{Unrolling Delayed Contracts}

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

1 \textit{let addOne} =
2 \{(\lambda \mathbf{z} (\text{plus} 1 \mathbf{z}))\}
3 \{(\lambda \mathbf{x} (\lambda \mathbf{y} (+ \mathbf{x} \mathbf{y})))\} @ (\text{Number?} \rightarrow \text{Number?})\]

While Number? is a flat contract that is checked immediately, function contracts are 	extit{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 \textit{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 \textit{plus} in \textit{addOne}.

5 \textit{let addOne} =
6 \{(\lambda \mathbf{z} (\text{plus} 1 \mathbf{z}))\}
7 \{(\lambda \mathbf{z} (\lambda \mathbf{y} (+ \mathbf{x} \mathbf{y})))\} @ (\text{Number?} \rightarrow \text{Number?})\]

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.

9 \textit{let addOne} =
10 \{(\lambda \mathbf{z} (\text{plus} 1 \mathbf{z}))\}
11 \{(\lambda \mathbf{z} (\lambda \mathbf{y} (+ \mathbf{x} \mathbf{y})))\} @ (\text{Number?} \rightarrow \text{Number?})\]

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, I 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 ⊤ accepts every value.

let addOne =
(((λ plus
  (λ z (((plus 1) [z @ Number?]) @ 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 different 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 @ Number?]) @ Number?))
(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) [z @ String?]) @ 1 Number?))
(λ z [((plus [1 @ 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 @ y and @ z.

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” ⊥, 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 ⊥]) @ 1 Number?)))
(λ x (λ y (+ x y))) @ (Τ → Number?)) @ 2 (Τ → 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 =
(int (λ plus
  (λ z (((plus [1 @ 2 ⊥]) [z @ 1 Number?])
(λ x (λ y (+ x y))) @ (∀ → 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 (Τ → 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 =
\[(\lambda \text{plus} \ (\lambda z \ ((\text{plus} \ (1 \ @ \ 2) \ z)))\]
\[(\lambda x \ (\lambda y \ (+ \ x \ y))) \ @ 1 \ (\text{Number} ? \rightarrow \text{Number} ?)\]
This step is only possible if the argument contract, here \text{Number} ? \rightarrow \top, is in negative position and if the return contract, \top \rightarrow \text{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 =
\[(\lambda \text{plus} \ ((\lambda z \ ((\text{plus} \ (1 \ @ \ 2) \ z)))\]
\[(\lambda x \ (\lambda y \ (+ \ x \ y))) \ @ 1 \ (\text{Number} ? \rightarrow \text{Number} ?)\] \right)
\[(\text{String} ? \rightarrow \text{String} ?))\]]

Here, \text{Positive}? is a flat contract that checks for positive numbers. In addition to the contract on \text{plus}, \text{addOne} contains another function contract, requiring \text{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 =
\[(\lambda \text{plus} \ ((\lambda z \ ((\text{plus} \ (1 \ @ \ 2) \ z)))\]
\[(\lambda x \ (\lambda y \ (+ \ x \ y))) \ @ 1 \ (\text{Number} ? \rightarrow \text{Number} ?)\] \right)
\[(\text{Positive} ? \rightarrow \text{Positive} ?)\]

The intersection is split into two parallel observations, as indicated by \parallel. 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 \text{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 \text{Positive}? \rightarrow \text{Positive}? is more restrictive than any other contract on \text{addOne}, that is, \text{Number}? \rightarrow \top and \top \rightarrow \text{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 =
\[(\lambda z \ ((\text{plus} \ (1 \ @ \ 2) \ z)))\]
\[(\lambda (\text{plus} \ (\lambda x \ (\lambda y \ (+ \ x \ y)))) \ @ 1 \ (\text{Number} ? \rightarrow \text{Number} ?)\] \right)
\[(\text{Positive} ? \rightarrow \text{Positive} ?)\]

\[
\begin{align*}
I, M, N & ::= K | x | \lambda x. M & M N & ::= \text{op}(\tilde{M}) | \text{if} \ L M N \\
K & ::= \text{false} | \text{true} & 0 & ::= 1 & \ldots \\
U, V, W & ::= K | \lambda x. M \\
\mathcal{E} & ::= \Box | \mathcal{E} N | V \mathcal{E} | \text{op}(\tilde{V} \mathcal{E}) & \text{if} \mathcal{E} M N
\end{align*}
\]

\[\delta_{blame} = \delta_{\text{false}} \in \text{delta}(\delta_{\text{true}})\]

\[\mathcal{E}[\lambda x. M V] \rightarrow \mathcal{E}[M[x \rightarrow V]] \quad \mathcal{E}[\text{op}(\tilde{V})] \rightarrow \mathcal{E}[\delta_{\text{op}}(\tilde{V})] \quad \tilde{V} \in \text{dom}(\delta_{\text{op}})\]

\[\begin{align*}
C, D & ::= I | Q | C \cup D | I \cap C \\
I, J & ::= \text{flat}(M) \\
Q, R & ::= C \rightarrow D | x \rightarrow A | Q \cap R \\
A & ::= \lambda x. C \\
B & ::= +\text{blame} | -\text{blame} \\
L, M, N & ::= +M \cap C | V \text{flat}(\text{eval} M) | B^\uparrow \\
U, V, W & ::= +M \cap Q | V \text{flat}(\text{eval} Q) \\
\mathcal{E} & ::= +\mathcal{E} \cap C | V \text{flat}(\text{eval} \mathcal{E}) \\
b & ::= b \mid \ell \\
k & ::= b \mid W | b \mid \ell \mid b \mid \ell \rightarrow \ell | b \mid \ell \cap \ell | b \mid \ell \cup \ell \\
\varsigma & ::= \bot \mid k \mid \varsigma
\end{align*}\]

3. Contracts and Contract Satisfaction

This section defines \(\lambda_{\text{Con}}\), an untyped call-by-value lambda calculus with contracts which serves as a core calculus for contract monitoring. It first introduces the base calculus \(\lambda_1\), 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 \(\lambda_1\)

Figure 1 defines syntax and semantics of \(\lambda_1\), 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 \(\rightarrow\) comprises beta-value reduction, built-in partial operations transforming a vector of values into a value, and the reduction of conditionals. We write \(\rightarrow^*\) for its reflexive, transitive closure and \(\rightarrow^*\) for its complement. That is, \(M \rightarrow^* N\) if, for all \(L\) such that \(M \rightarrow^* L\), it holds that \(L \neq N\). We also write \(M \rightarrow\) to indicate that there is no \(N\) such that \(M \rightarrow N\).

3.2 Contracts and Contracted \(\lambda_1\)

Figure 2 defines the syntax of \(\lambda_{\text{Con}}\) as an extension of \(\lambda_1\). 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 contracts \( (C \cup D) \), 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 \( C \to D \) with domain contract \( D \) and range contract \( D \), a dependent contract \( x \to A \) 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 \( V@\delta \) 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 \( \delta \) first erases all contract monitoring from its arguments, then processes the underlying \( \lambda \)-values, and finally returns a \( \lambda \)-value.

The only new source term is contract assertion \( M@C \). The adornments \( b \) and \( i \) are drawn from an unspecified denumerable set of blame identifiers \( b \), which comprises blame labels \( l \) that occur in source terms and blame variables \( i \) that are introduced during evaluation.

In the intermediate term \( V@\delta \)(eval \( M \)), the term \( M \) represents the current evaluation state of the predicate of a flat contract. The \( B^j \) expression signal a contract violation at label \( \beta \). 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 \( \lambda_{\text{Con}} \), contract monitoring occurs via constraints \( \kappa \) 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 \( \varsigma \) during reduction.

### 3.3 Semantics for \( \lambda_{\text{Con}} \)

Figure 3 specifies the small-step reduction semantics of \( \lambda_{\text{Con}} \), as a relation \( \varsigma, M \to \varsigma', N \) on pairs of a constraint list and a term. Instead of raising blame exceptions, the rewriting rules for contract enforcement generate constraints in \( \varsigma \): 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 \texttt{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 \texttt{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 \texttt{UNIT} picks up the result and stores it in a constraint.

The second group of rules \texttt{UNION} and \texttt{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 \texttt{D-FUNCTION}, \texttt{D-DEPENDENT}, and \texttt{D-INTERSECTION} define the behavior of a contracted value under function application. Rule \texttt{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 \texttt{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 \texttt{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, \( \lambda_l \) reductions are lifted to \( \lambda_{\text{Con}} \) reductions using rule \texttt{BASE}. Built-in operations and conditions “see through” contracts (rules \texttt{D-OP} and \texttt{D-IF}) and proceed on \( \lambda_l \)-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, \( b_{\text{subject}} \) and \( b_{\text{context}} \). Intuitively, if \( b_{\text{subject}} \) is false, then the contract associated with \( b \) is not subject-satisfied and may lead to positive blame for \( b \). If \( b_{\text{context}} \) 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 \( \lambda_{\text{Con}} \).

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

**Definition 1.** \( \varsigma \) is a blame state for blame label \( \beta \) iff

\[
\exists \varsigma. (\varsigma_{\text{subject}} \land \varsigma_{\text{context}}) \equiv \text{false}.
\]

\( \varsigma \) is a blame state if there exists a blame label \( \beta \) such that \( \varsigma \) is a blame state for this label.

To model reduction with blame, we define a new reduction relation \( \varsigma, M \to \varsigma', N \) on configurations. It behaves like \( \to \) unless \( \varsigma \) is a blame state. In a blame state, it stops signaling the violation. There are no reductions with \( B \).

\[
\begin{align*}
\exists \varsigma. (\varsigma_{\text{subject}}) \equiv \text{false} & \quad \exists \varsigma. (\varsigma_{\text{context}}) \equiv \text{false} \\
\varsigma, M \to \varsigma', N & \quad \text{if} \quad \varsigma, M \to \varsigma', N
\end{align*}
\]

#### 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.)
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 $\lambda_{\text{Con}}$ terms $M$ and extend syntax and semantics of $\lambda_{\text{Con}}$ 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 $\lambda_{\text{Con}}$.

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 transferred into a constraint and must not be checked like a predicate.

Source terms $M$ now contain partially evaluated contracts on source terms $M@_C$ and a new inversion constraint $b \leftarrow \kappa$ extends the set of constraints $\kappa$. 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 $\text{REDUCE/TRUE}$ and $\text{REDUCE/FALSE}$ creates a constraint for $\top$ and $\bot$. Three special rules, $\text{D-DOMAIN}$, $\text{D-RANGE}$, and $\text{D-FALSE}$ 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 constraint 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 $\lambda_{\text{Con}}$ terms $M$. Appendix B.1 shows its definition.

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

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:

\[
\text{flat}(\lambda x. (x > 0)) \cap \text{flat}(\lambda x. (x \% 2 = 0))
\]

However, there is a twist. Fine-grained sub-contracts enable contracts to be optimised 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:

\[
\text{flat}(\lambda x. (x > 0))
\]
Obviously, one can write a similar contract like this:

\[ \text{flatt}(\lambda x. (x > 0)) \cup \text{flatt}(\lambda x. (x = 0)) \]

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 caused 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. \( \mathcal{E}[M@^\circ C] \rightarrow^* V \) implies \( \mathcal{E}[M@^\circ D] \rightarrow^* V \) and \( \mathcal{E}[M@^\circ D] \rightarrow^* B^2 \) implies \( \mathcal{E}[M@^\circ C] \rightarrow^* B^2 \).

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 \( \text{Positive} = \text{flatt}(\lambda x. x > 0) \) and \( \text{Natural} = \text{flatt}(\lambda x. x \geq 0) \). Then \( \text{Positive} \sqsubseteq^+ \text{Natural} \) as for all \( x \) it holds that \( x > 0 \) implies \( x \geq 0 \).

For another example, \( \text{Positive} \rightarrow \text{Positive} \) is a subcontract of \( \text{Natural} \rightarrow \text{Natural} \) as it is more restrictive on both parties. Furthermore, \( \text{Positive} \rightarrow \text{Natural} \) and \( \text{Natural} \rightarrow \text{Positive} \) are further subcontracts of \( \text{Positive} \rightarrow \text{Positive} \).

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 subject 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 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 judgements are defined in terms of each other and track swapping of responsibilities on function arguments. Both judgements are reflexive and transitive.

We further write \( M \sqsubseteq N \) if predicate \( M \) implies predicate \( N \). For example, \( (x>0) \sqsubseteq (x>0) \) and \( (x=0) \sqsubseteq (x\geq0) \).

However, our subcontracting judgement stops at \( M \sqsubseteq N \). Relation \( \sqsubseteq \) refers to an environment \( \Gamma \) that resolves relations on terms. \( \Gamma \) might be given by a SAT solver or a set of predefined relations.

\[
\begin{align*}
(M \sqsubseteq N) & \in \Gamma \\
\frac{M \sqsubseteq N}{M \sqsubseteq N}
\end{align*}
\]

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

\[
C \sqsubseteq^+ D \land C \sqsubseteq^+ D.
\]

We further define an ordinary subcontracting judgement, \( C \sqsubseteq D \), which corresponds to the usual definitions 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

\[
D \sqsubseteq C \land C \sqsubseteq^+ D.
\]

In addition to the already mentioned notation, we sometimes use \( \sqsubseteq, \sqsubseteq^+, \sqsubseteq^+ \), and \( \sqsubseteq \) 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:

- \( C \sqsubseteq C \)
Subject Subcontract

\[
C \subseteq C \quad I \subseteq J
\]
\[
C_D \subseteq D_D \quad C_R \subseteq D_R
\]
\[
x \rightarrow \Lambda x.C \subseteq x \rightarrow \Lambda x.D
\]
\[
C \subseteq D_L \quad C \subseteq D_R
\]
\[
C_L \subseteq D_L \quad C_R \subseteq D_R
\]
\[
C_L \subseteq D_L \cap D_R
\]
\[
C_R \subseteq D_R
\]
\[
C_L \subseteq D_L \cup D_R
\]
\[
C_R \subseteq D_R
\]

5.4 Subset Transformation

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, union 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' 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' as we do not know if the body is ever executed (e.g. because it is nested in a condition or the lambda term is never used in an application).

Here, η(ι1, ι2) computes a blame term B' resulting from a predicate violation (ι1 ⊨ false) in ι2 (cf. Appendix C.3).

As before, []@ι must remain on the B’ term to remember the violated contract. However, as the whole body transforms to B’, ⊥ 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'. 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 ⊥ 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 on 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 preconditional η(ι1, ι2) = η(ι2, ι2).
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 \( G \simeq H \) 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 \( \lambda x. \) 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 \( T \mapsto \cdot \cdot \cdot \) on terms.

First, rule \textsc{Join} triggers the synchronisation of two canonical terms nested in a parallel observation and rule \textsc{Match} dissolves a parallel observation if both terms are identical.

The rules \textsc{Synchronise/Left} and \textsc{Synchronise/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 \( \square@^1 \cdot \) still remains in the source program.

Finally, rule \textsc{Synchronise/Contract} looks for different contract assertions on a term \( S \) and merges the contract assertions. Precondition \( G \equiv H \) requires that the enclosing contexts are structurally equivalent. Only the contract assertions must be different.

Here, \( A_e \sqcup A_i \) 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.

---

\[
\mathcal{M}, \mathcal{N} := \square | \lambda x. \mathcal{M} | \mathcal{M} \cdot \mathcal{N} | \mathcal{M} \cdot \mathcal{N} | \mathcal{O} (\mathcal{M} \cdot \mathcal{N}) \\
| \text{if } \mathcal{M} \mathcal{N} | \text{if } \mathcal{M} \mathcal{N} | \text{if } \mathcal{M} \mathcal{N} | \mathcal{L} \mathcal{C} \mathcal{E} \mathcal{N} \\
| \mathcal{L}, := \lambda x. \mathcal{M} | \mathcal{M} \cdot \mathcal{N} | \mathcal{M} \cdot \mathcal{N} | \mathcal{O} (\mathcal{M} \cdot \mathcal{N}) \\
| \text{if } \mathcal{M} \mathcal{N} | \text{if } \mathcal{M} \mathcal{N} | \text{if } \mathcal{M} \mathcal{N} | \mathcal{L} \mathcal{C} \mathcal{E} \mathcal{N}
\]

---

**Figure 11.** Subset transformation.

**Figure 12.** Final terms and contexts.
Term Equivalence

\[ K \simeq K \quad x \simeq x \quad \frac{M \simeq N}{\lambda x.M \simeq \lambda x.N} \]

\[ \frac{M \simeq N}{M_0 \simeq N_0 \quad M_1 \simeq N_1} \quad \frac{M_0 M_1 \simeq N_0 N_1}{M \simeq N} \]

\[ \frac{\vec{M} \simeq \vec{N}}{\text{op}(\vec{M}) \simeq \text{op}(\vec{N})} \quad \vec{M}_0 \simeq \vec{N}_0 \quad \vec{M}_1 \simeq \vec{N}_1 \quad \frac{\vec{M}_2 \simeq \vec{N}_2}{\text{if } \vec{M}_0 M_1 M_2 \simeq \text{if } \vec{N}_0 N_1 N_2} \]

\[ \frac{M \simeq N}{M \simeq N @^0 C \simeq N} \quad \frac{M \simeq N @^0 C}{M \simeq N \parallel M \simeq N @^0 C} \quad \frac{M \simeq N @^0 C}{M \simeq N \parallel M \simeq N} \]

Context Equivalence

\[ \square \simeq \square \quad \frac{M \simeq N \quad M \simeq N}{M \parallel C \simeq N \parallel C} \quad \frac{M \simeq N}{M \parallel N @^1 C \simeq N \parallel N @^1 C} \]

\[ \frac{\lambda x.M \simeq \lambda x.N}{M \simeq N} \quad \frac{M \simeq N \quad M \simeq N}{M \parallel M \simeq N \parallel N} \quad \frac{M \simeq N}{M \parallel M \simeq N \parallel M} \]

\[ \frac{M \simeq N \quad \vec{M}_0 \simeq \vec{N}_0 \quad \vec{M}_1 \simeq \vec{N}_1 \quad \vec{M}_2 \simeq \vec{N}_2}{\text{op}(\vec{M}_0 \quad \vec{M}_1 \quad \vec{M}_2) \simeq \text{op}(\vec{N}_0 \quad \vec{N}_1 \quad \vec{N}_2)} \quad \frac{M_0 \simeq N_0 \quad M_1 \simeq N_1 \quad M_2 \simeq N_2}{\text{if } M_0 M_1 M_2 \simeq \text{if } N_0 N_1 N_2} \]

\[ \frac{M \simeq N \quad M_0 \simeq N_0 \quad M_1 \simeq N_1 \quad M_2 \simeq N_2}{\text{if } M_0 M_1 M_2 \simeq \text{if } N_0 N_1 N_2} \]

Figure 13. Term and context equivalence.

\[
\begin{align*}
\text{JOIN} & \quad \frac{(M \parallel N) \rightarrow \lambda\cdot j\cdot (M' \parallel N')} {T[(M \parallel N)] \rightarrow \lambda\cdot j\cdot T[(M' \parallel N')]} \\
\text{MATCH} & \quad \frac{(M \parallel M) \rightarrow \lambda\cdot j\cdot M} {T[(M \parallel M)] \rightarrow \lambda\cdot j\cdot T[(M' \parallel M)']}
\end{align*}
\]

\[ \text{SYNCHRONIZE/LEFT: } \frac{M \simeq N} {(M[A_1][S] \parallel N[A_2][B']) \rightarrow \lambda\cdot j\cdot (M[A_1][S] \parallel N[A_2][B'])} \]

\[ \text{SYNCHRONIZE/RIGHT: } \frac{M \simeq N} {(M[B'] \parallel N[A_2][S]) \rightarrow \lambda\cdot j\cdot (M[A_1][S] \parallel N[A_2][S])} \]

\[ \text{SYNCHRONIZE/CONTRACT: } \frac{M \simeq N \quad A_1 \neq A_2} {(M[A_1][S] \parallel N[A_2][S]) \rightarrow \lambda\cdot j\cdot (M[A_1][S] \parallel N[A_2][S])} \]

Figure 14. Join parallel observations.

Conjecture 2 (Weak Blame-preservation). For all \( \varsigma_M, M \rightarrow^* \varsigma_N, N \) it holds that either \( \varsigma_M = \varsigma_M', M \rightarrow^* \varsigma_N, V \) and \( \varsigma_N = \varsigma_N', V \) or \( \varsigma_M, M \rightarrow^* \varsigma_M', B_M^0 \) and \( \varsigma_N, N \rightarrow^* \varsigma_N', B_N^0 \).

6. Technical Results

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

Definition 6 (Size). The size \( |L| \) of a closed term \( L \) is the number of predicate checks during reduction \( \rightarrow \) of \( L \).

Theorem 1 (Improvement). For each term \( M \) with \( \varsigma, M \rightarrow \varsigma, N \) 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 \( \varsigma_M, M \rightarrow^* \varsigma_T, T \).
7. 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.

---

**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.

---

8. 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 to 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 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 aspect 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.

---

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


\[ \tau(V) = \begin{cases} 
\text{false} & \text{if } V = \text{false} \\
\tau(W) & \text{if } W \neq Q \\
\text{true} & \text{otherwise} 
\end{cases} \]

Figure 17. Mapping values to truth values.


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

An interpretation $\mu$ of a constraint list $\zeta$ is a mapping from blame identifiers to records of elements of $\mathbb{B} = \{true, false\}$, such that all constraints are satisfied. We order truth values by $true \sqsubseteq false$ 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

$$\mu \in (\{b\} \times \{subject, context\}) \rightarrow \mathbb{B}$$

where $\{b\}$ ranges over metavariable $b$ and constraint satisfaction by a relation $\models$, which is specified in Figure 2.

In the premises, the rules apply a constraint mapping $\mu$ 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 $W = false$ and otherwise to true, where the function $\tau(\cdot) : \{V\} \rightarrow \mathbb{B}$ translates...
values to truth values by stripping delayed contracts (see Figure 1). A flat contract never blames its context so that b.context is always true.

The rule CT-FUNCTION determines the blame assignment for a function contract b from the blame assignment for the argument and result contracts, which are available through t₁ and t₂. Let’s first consider the subject part. A function satisfies contract b if it satisfies its obligations towards its argument t₁.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 function-arguments.

A function’s context (caller) satisfies the contract if it passes an argument that satisfies contract t₁.context and uses the result according to its contract t₂.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 t₁ and t₂. A subject satisfies an intersection contract if it satisfies both constituent contracts: t₁.context ∧ t₂.context. A context, however, has the choice to fulfill one of the constituent contracts: t₁.context ∨ t₂.context.

Dually, the rule CT-UNION determines the blame assignment for a union contract at b from its constituents at t₁ and t₂. A subject satisfies a union contract if it satisfies one of the constituent contracts: t₁.context ∨ t₂.context. A context, however, needs to fulfill both constituent contracts: t₁.context ∧ t₂.context, because it does not know which contract is satisfied by the subject.
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 and non-canonical terms of the Baseline Transformation.

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.

<table>
<thead>
<tr>
<th>Canonical Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{Val}$ ::= $K \mid \lambda x.S$</td>
</tr>
<tr>
<td>$S_{NonVal}$ ::= $x \mid B^\perp$</td>
</tr>
<tr>
<td>$T_{Const}$ ::= $K \mid T_{Const} \downarrow_\perp$</td>
</tr>
<tr>
<td>$T_{Abs}$ ::= $\lambda x.S \mid T_{Abs} \downarrow_\perp$</td>
</tr>
<tr>
<td>$T_{Val}$ ::= $S_{Val} \mid T_{Val} \downarrow_\perp$</td>
</tr>
<tr>
<td>$T_1$ ::= $S_{NonVal} \mid T_1 \downarrow_\perp$</td>
</tr>
<tr>
<td>$T_Q$ ::= $T_{Val} \mid T_Q \downarrow_\perp$</td>
</tr>
<tr>
<td>$T_{NonQ}$ ::= $T_{Val} \mid T_1$</td>
</tr>
<tr>
<td>$T$ ::= $T_Q$</td>
</tr>
</tbody>
</table>

Non-Canonical Terms

| O, P ::= $\lambda x.O \mid O \cup M.O \mid op(M.O.N)$ |
| $\mid if O M N \mid if L O N \mid if L M O \mid O\downarrow_\perp$ |
| $\mid \forall \forall [x.M] \mid (N\downarrow_\perp.Q) \mid (M\downarrow_\perp.Q) N$ |
| $\mid \forall \forall [V] \downarrow_\perp I \mid \lambda x.(M\downarrow_\perp.C)$ |
| $\mid op(L(M\downarrow_\perp.Q) \downarrow_\perp) \mid if L(M\downarrow_\perp.C) \downarrow_\perp$ |
| $\mid M\downarrow_\perp \uparrow \mid M\downarrow_\perp C \mid M\downarrow_\perp C \cup D \mid M\downarrow_\perp I \cap C$ |
| $\mid (M\downarrow_\perp.Q)\downarrow_\perp I \mid (M\downarrow_\perp.Q)\downarrow_\perp \downarrow_\perp$ |

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.

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 $T_1$ with an indefinite number of delayed contracts on the outermost position. Terms $T_{Val}$ are all values with remaining \perp contracts. Terms $T_1$ are all terms with immediate contracts or \perp 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

| O, P ::= $\lambda x.O \mid K \mid \lambda x.S \mid \lambda x.S \neq \lambda x.B[x\downarrow_\perp]$ |
| $\mid if T_0 T_1 \downarrow_\perp \mid T_1 T_Q$ |
| $\mid S_{Val} \mid T_{Val}$ |
| $\mid op(T_{NonQ})$ |
| $\mid if T_Q A[T_Q] \downarrow_\perp A[T_Q]$ where not $T_Q = T' \downarrow_\perp C$ and $T_Q' = T'' \downarrow_\perp C$ |

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

12.1 Function root-of(b, ς)

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

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

To do so, Figure 5 introduces a constraint context K, which is either a hole \( □ \) or constraint \( κ \) extending a constraint context K.

Figure 6 finally shows the computation of function root-of(b, ς).

12.2 Sign of a Blame Identifier

Following the definition in Section 3, constraints \( κ \) implement a tree rooted in a blame label b. Within the tree every blame identifier b can report positive or negative blame, depending on its constraint when mapping one of its truth values (subject or 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 \( (b ▶ false) : ς \) leads to \( B′ \) on a single path for any b.

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 b.

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 then 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.
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 $T_2^@Φ (I ← (□@Φ, Q))$ 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 $M@Φ Q → ⊤$ 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 \( (\varsigma_M, M) \) and \( (\varsigma_N, N) \) are structural equivalent under renaming \( \alpha \), written \( \varsigma_M, M \equiv_{\alpha} \varsigma_N, N \) with \( \alpha \) of free fresh variables \( \iota \) if \( \varsigma_M, M \equiv_{\alpha} (\varsigma_N, N)[\iota \mapsto \iota' \nu] \) with \( \alpha \equiv \alpha[\iota \mapsto \iota'] \).

Lemma 7. For all configurations \( (\varsigma_M, M) \) and \( (\varsigma_N, N) \) with \( \varsigma_M, M \equiv_{\alpha} \varsigma_N, N \) it holds that \( \varsigma_M, M \rightarrow \varsigma'_M, M' \) and \( \varsigma_N, N \rightarrow \varsigma'_N, N' \) with \( \varsigma_M, M' \equiv_{\alpha} \varsigma_N, N' \).

Proof of Theorem 3. Proof by induction over \( \varsigma \) and \( M \).

14.5 Congruence of States and Terms

Definition 9. State \( \varsigma' \) is an effect-free extension of state \( \varsigma \), written \( \varsigma \equiv_{\iota} \varsigma' \), iff \( \varsigma, M \rightarrow \iota \varsigma', M' \) and \( \varsigma', M \rightarrow \iota \varsigma'' \), \( M' \) with \( \varsigma'' \equiv_{\iota} \varsigma' \).

Lemma 8. For terms \( M \), states \( \varsigma \), and transformations \( \varsigma, M \implies \varsigma' \), \( N \) it hold that \( \varsigma \equiv_{\iota} \varsigma' \).

Proof of Theorem 3. Immediate by the definition of \( \implies \) and \( \equiv \). Every contract is composed of fresh blame variable \( \iota \). Proof by induction over \( \implies \).

Proof Sketch for Conjecture 1. Let reduction output \( O \) be a set \( V \cup B \). For all terms \( M \) and states \( \varsigma_M \), if \( \varsigma_M, M \rightarrow_{\iota \beta} \varsigma_N, N \) it holds that \( \varsigma_M, M \implies \varsigma_N, O \) and \( \varsigma_N, N \implies \varsigma'_N, O \) with \( \varsigma'_N \equiv_{\iota} \varsigma_N \).

As our transformation only propagates contract assertions and did not modify the underlying program, we know that if \( \varsigma, M \rightarrow \iota \varsigma', V \) and \( \varsigma, N \rightarrow \iota \varsigma'', W \) then \( \varsigma, V \rightarrow W \).

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 \( \iota \).

Proof by induction over \( \implies \).

Rule UNFOLD/ASSERT: \( M = T \circ^\circ C, N = T \circ^\circ C \), and \( \varsigma_N = \langle \rho \downarrow \iota \rangle \circ \varsigma_M \).

(1) From \( M \) and \( \implies \) it follows that \( \varsigma_M, T \circ^\circ C \rightarrow^* \varsigma'_M, V \circ^\circ C \) and by Rule ASSERT it follows that \( \varsigma'_M, V \circ^\circ C \rightarrow \langle \rho \downarrow \iota \rangle \circ \varsigma_M, V \circ^\circ C \).

(2) From \( \varsigma_N, T \circ^\circ C \rightarrow^* \varsigma'_N, V \circ^\circ C \) with \( \varsigma_M \equiv_{\iota} \varsigma'_N \).

Claim holds because \( \rho \downarrow \iota \).

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

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

Rule UNFOLD/D-DUNCTION: \( M = (T_1 \circ^\circ (C \rightarrow D))T_2, N = (T_1 (T_2 \circ^\circ (C \rightarrow D)) \circ \circ D \), and \( \varsigma_N = \langle \iota \downarrow \iota' \rangle \circ \varsigma_M \).

(1) From \( M \) and \( \implies \) it follows that \( \varsigma_M, (T_1 \circ^\circ (C \rightarrow D)) \rightarrow^* \varsigma'_M, (V_1 \circ^\circ (C \rightarrow D) \circ \circ V_2 \circ \circ B \).

By rule D-DUNCTION it follows that \( \varsigma'_M, (V_1 \circ^\circ (C \rightarrow D)) \rightarrow \langle \iota \downarrow \iota' \rangle \circ \varsigma_M, (V_1 (V_2 \circ^\circ (C \rightarrow D)) \circ \circ D \).

(2) From \( \varsigma_N, (T_1 (T_2 \circ^\circ (C \rightarrow D)) \circ \circ D \rightarrow^* \varsigma'_N, (V_1 (V_2 \circ^\circ (C \rightarrow D)) \circ \circ D \).

Claim holds because \( \iota \downarrow \iota' \).

Proof of Theorem 1. Proof by induction over \( \implies \). 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 rules 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 \( \rightarrow \), we have that for all \( \lambda x. M (N \circ^\circ Q) \) first \( N \rightarrow^* V \) and by rule Beta it follows that \( \lambda x. M (V \circ^\circ Q) \rightarrow \lambda x. M [x \mapsto (V \circ^\circ Q)] \).

Rule UNROLL we have that \( \lambda x. M (N \circ^\circ Q) \) translates to \( \lambda x. M [x \mapsto (x \circ^\circ Q)] \).

From \( \rightarrow \), we now have that \( N \rightarrow^* V \) and by rule Beta if follows that \( \lambda x. M [x \mapsto (x \circ^\circ Q)] \rightarrow^* V \), 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.
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 subcontrack relation to simplify contracts before unfolding.

For example, the intersection contract \( \text{Positive}? \rightarrow \text{Natural}? \cap \text{Natural}? \rightarrow \text{Positive}? \) can be simplified to a single function contract \( \text{Positive}? \rightarrow \text{Positive}? \). 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 \( \text{Natural}? \rightarrow \text{Positive}? \) is an ordinary subset of \( \text{Positive}? \rightarrow \text{Natural}? \) (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 \textit{Unfold/Assert} in this cases.
\[
S_{\text{Val}} ::= K | \lambda x. S \quad \text{where} \quad \lambda x. S \neq \lambda x. B[x@1^C]
\]
\[
S_{\text{NonVal}} ::= x | B^b
\]
\[
| T_i T_Q | K T_Q | S_{\text{Val}} T_i | S_{\text{Val}} S
\]
\[
| \text{op}(T_i) | \text{op}(S_{\text{Val}})
\]
\[
| \text{let} T_Q A[T_Q] @ T_i' Q | \text{let} T_Q @ T_i' Q \quad \text{where not} \quad T_Q = T'_@1^C \text{ and } T_Q' = T''_@1^C
\]
\[
T_i ::= S_{\text{NonVal}} | S_{\text{NonVal}} @^2 I
\]
\[
| \text{let} T_i@^3 I \quad \text{where} \quad I \nsubseteq J, J \nsubseteq I
\]
\[
T_Q ::= S_{\text{Val}} | T_i | S_{\text{Val}} @^3 Q | T_i @^3 Q
\]
\[
| \text{let} T_Q @^3 Q \quad \text{where} \quad Q \nsubseteq R, Q \nsubseteq R
\]
\[
\quad \text{and not} \quad Q \neq C \rightarrow \top, R \neq \top \rightarrow D, \beta_0 = \gamma_1
\]
\[
T ::= T_Q | (T_r \parallel T_l)
\]

---

**Figure 28.** Canonical terms and contexts.

---

**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 \( \lambda_{\text{Con}} \) 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** \( \rightarrow_S \) and rule **Trace** unpacks parallel observation. Obviously, rule **Blame** replaces the rule with the same name in \( \rightarrow_S \) 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 \( \perp \) 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.

\[
\begin{align*}
&\text{let } f = \\
&\quad ((\lambda \text{addOne } (\lambda z (\text{addOne } z))) \\
&\quad ((\lambda \text{plus } (\lambda z ((\text{plus } 0) z))) \\
&\quad ((\lambda x (\lambda y (\text{positive? } \rightarrow \text{positive?}))) @ (\text{positive? } \rightarrow \text{positive?})))
\end{align*}
\]

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

\[
\begin{align*}
&\text{let } f = \\
&\quad ((\lambda \text{addOne } (\lambda z (\text{addOne } z))) \\
&\quad ((\lambda \text{plus } (\lambda z (((\text{plus } 0) @ \perp)) @ (\text{positive? } \rightarrow \text{positive?}))) z)) \\
&\quad (\lambda x (\lambda y (\text{positive? } \rightarrow \text{positive?})))
\end{align*}
\]

Even though we do not know that \text{addOne} violates the contract on \text{plus}, we are not allowed to treat the contract as violated because we do not know if \text{addOne} is ever executed. But, we can replace \text{addOne}'s body by a blame term that reports a violation whenever the function is executed. In addition, a \( \perp \) remains on the function body.

\[
\begin{align*}
&\text{let } f = \\
&\quad ((\lambda \text{addOne } (\lambda z (\text{addOne } z))) \\
&\quad ((\lambda \text{plus } (\lambda z (\text{blame} @ \perp))) z)) \\
&\quad (\lambda x (\lambda y (\text{positive? } \rightarrow \text{positive?})))
\end{align*}
\]

Having a blame term and \( \perp \) seems to be twofold, but \( \perp \) can be lowered to a new function contract on \text{addOne}, which is then unrolled to all places where \text{addOne} is used, as the following example demonstrates.

\[
\begin{align*}
&\text{let } f = \\
&\quad ((\lambda \text{addOne } (\lambda z (\text{addOne } z))) \\
&\quad ((\lambda \text{plus } (\lambda z (\text{blame} @ \perp))) z)) \\
&\quad (\lambda x (\lambda y (\text{positive? } \rightarrow \text{positive?})))
\end{align*}
\]

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

Continuing in this way propagates the information of a failing contract 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.

\[
\begin{align*}
&\text{let } f = \\
&\quad (((\lambda \text{addOne } (\lambda z (\text{addOne } z))) @ (\top \rightarrow \perp)) z)) \\
&\quad ((\lambda \text{plus } (\lambda z (\text{blame}))) (\lambda x (\lambda y (+ x y))) @ (\top \rightarrow \perp))
\end{align*}
\]
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 must 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. 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.

---

1https://developers.google.com/octane/
19. Practical Evaluation, continued

To demonstrate the runtime improvement we applied our contract simplification to a number simple examples, as introduces in Section 2 and Section 7. The testing process 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\(^3\) (Keil and Thienn 2015b) contract system for JavaScript and the Mozilla SpiderMonkey\(^6\) engine. While there is actually no full 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 contract (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 predicate checks.

```javascript
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.

```javascript
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.

```javascript
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.

```javascript
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.

```javascript
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], 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.

```javascript
var addOne6 = (function () {
  var plus = assert(function (x, y) {
    return x + y;
  }, Intersection(
      AFunction([Number, Number], Number),
      AFunction([String, String], String)
    ));
  return addOne;
})();
```

---

\(^3\) 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 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 granularity 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.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Normal</th>
<th>Baseline</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time (ms)</td>
<td>predicates</td>
<td>time (ms)</td>
</tr>
<tr>
<td>Without Contracts</td>
<td>2</td>
<td>0</td>
<td>200000</td>
</tr>
<tr>
<td>Proxy only</td>
<td>142.33</td>
<td>0</td>
<td>46085</td>
</tr>
<tr>
<td>Example 1</td>
<td>39404</td>
<td>300000</td>
<td>27081</td>
</tr>
<tr>
<td>Example 3</td>
<td>87143</td>
<td>600000</td>
<td>58385</td>
</tr>
<tr>
<td>Example 4</td>
<td>66474</td>
<td>500000</td>
<td>54396</td>
</tr>
<tr>
<td>Example 5</td>
<td>114468</td>
<td>800000</td>
<td>85126</td>
</tr>
<tr>
<td>Example 6</td>
<td>295682</td>
<td>1700000</td>
<td>107956</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Normal</th>
<th>Baseline</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time (ms)</td>
<td>predicates</td>
<td>time (ms)</td>
</tr>
<tr>
<td>Without Contracts</td>
<td>9.67</td>
<td>0</td>
<td>40383</td>
</tr>
<tr>
<td>Proxy only</td>
<td>132.67</td>
<td>0</td>
<td>69492</td>
</tr>
<tr>
<td>Example 1</td>
<td>58906</td>
<td>300000</td>
<td>40383</td>
</tr>
<tr>
<td>Example 3</td>
<td>131184</td>
<td>600000</td>
<td>87544</td>
</tr>
<tr>
<td>Example 4</td>
<td>99776</td>
<td>500000</td>
<td>81719</td>
</tr>
<tr>
<td>Example 5</td>
<td>173037</td>
<td>800000</td>
<td>128754</td>
</tr>
<tr>
<td>Example 6</td>
<td>441176</td>
<td>1700000</td>
<td>278601</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Normal</th>
<th>Baseline</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time (ms)</td>
<td>predicates</td>
<td>time (ms)</td>
</tr>
<tr>
<td>Without Contracts</td>
<td>40.33</td>
<td>0</td>
<td>54945</td>
</tr>
<tr>
<td>Proxy only</td>
<td>200.67</td>
<td>0</td>
<td>59970</td>
</tr>
<tr>
<td>Example 1</td>
<td>81125</td>
<td>300000</td>
<td>56439</td>
</tr>
<tr>
<td>Example 3</td>
<td>186069</td>
<td>600000</td>
<td>124434</td>
</tr>
<tr>
<td>Example 4</td>
<td>136728</td>
<td>500000</td>
<td>111596</td>
</tr>
<tr>
<td>Example 5</td>
<td>240724</td>
<td>800000</td>
<td>179451</td>
</tr>
<tr>
<td>Example 6</td>
<td>315184</td>
<td>1000000</td>
<td>225316</td>
</tr>
</tbody>
</table>

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).