Collapsible Contracts: Fixing a Pathology of Gradual Typing

DANIEL FELTEY, Northwestern University, USA
BEN GREENMAN, Northeastern University, USA
CHRISTOPHE SCHOLLIERS, Ghent University, Belgium
ROBERT BRUCE FINDLER, Northwestern University, USA
VINCENT ST-AMOUR, Northwestern University, USA

The promise of gradual typing is that programmers should get the best of both worlds: the static guarantees of static types, and the dynamic flexibility of untyped programming. This is an enticing benefit, but one that, in practice, may carry significant costs. Significant enough, in fact, to threaten the very practicality of gradual typing; slowdowns as high as 120x are reported as arising from gradual typing.

If one examines these results closely, though, it becomes clear that the costs of gradual typing are not evenly distributed. Indeed, while mixing typed and untyped code almost invariably carries non-trivial costs, many truly deal-breaking slowdowns exhibit pathological performance. Unfortunately, the very presence of these pathological cases—and therefore the possibility of hitting them during development—makes gradual typing a risky proposition in any setting that even remotely cares about performance.

This work attacks one source of large overheads in these pathological cases: an accumulation of contract wrappers that perform redundant checks. The work introduces a novel strategy for contract checking—*collapsible contracts*—which eliminates this redundancy for function and vector contracts and drastically reduces the overhead of contract wrappers.

We implemented this checking strategy as part of the Racket contract system, which is used in the Typed Racket gradual typing system. Our experiments show that our strategy successfully brings a class of pathological cases in line with normal cases, while not introducing an undue overhead to any of the other cases. Our results also show that the performance of gradual typing in Racket remains prohibitive for many programs, but that collapsible contracts are one essential ingredient in reducing the cost of gradual typing.

CCS Concepts: • **Software and its engineering** → **Software performance**;

Additional Key Words and Phrases: gradual typing, migratory typing, contracts, runtime support

ACM Reference Format:
Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. Collapsible Contracts: Fixing a Pathology of Gradual Typing. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 133 (November 2018), 27 pages. https://doi.org/10.1145/3276503

1 GRADUAL TYPING, BUT AT WHAT COST?

The literature on gradual typing presents three fundamentally different opinions on how to combine statically typed and dynamically typed code [Chung et al. 2018; Greenman and Felleisen 2018]. One approach is to erase types at runtime [Bierman et al. 2014; Bracha and Griswold 1993; Chaudhuri et

Authors' addresses: Daniel Feltey, Northwestern University, USA, daniel.feltey@eecs.northwestern.edu; Ben Greenman, Northeastern University, USA, benjaminlgreenman@gmail.com; Christophe Scholliers, Ghent University, Belgium, Christophe.Scholliers@UGent.be; Robert Bruce Findler, Northwestern University, USA, robby@eecs.northwestern.edu; Vincent St-Amour, Northwestern University, USA, stamourv@eecs.northwestern.edu.

2475-1421/2018/11-ART133
https://doi.org/10.1145/3276503

al. 2017]. A second approach is to guard the boundaries between typed and untyped regions with first-order checks [Muehlboeck and Tate 2017; Rastogi et al. 2015; Roberts et al. 2018; Vitousek et al. 2017; Wrigstad et al. 2010]. A third approach is to enforce types with higher-order contracts [Allende et al. 2013; Tobin-Hochstadt et al. 2017; Williams et al. 2017]. Each approach offers different static guarantees and performance consequences as well as tradeoffs in expressiveness and flexibility [Greenman and Felleisen 2018].

Among the various strategies, the higher-order approach to gradual typing is the only one known to provide strong guarantees without limiting the expressiveness of dynamically typed code. In a higher-order system such as Typed Racket, a programmer can start with a dynamically typed codebase, add static type checking to any one module, and benefit from both *type soundness* and *correct blame* [Tobin-Hochstadt et al. 2017]. Type soundness guarantees that any runtime violation of the static types is detected as early as possible during the execution of the program; correct blame guarantees that every such violation is attributed to exactly one boundary between a static type annotation and a dynamically typed value. In other words: when a runtime mismatch between a static type and an untyped value occurs, the error message reports the two relevant parties.

Unfortunately, a system that implements the higher-order approach must augment the semantics of a program in order to enforce types and track blame information. The net cost of these additions can be extremely high; Takikawa et al. [2016] report that freely adding types to a Typed Racket program can lead to slowdowns of over two orders of magnitude. This means that programmers must be extremely careful when mixing typed and untyped code in practice, because some combinations may render their program unacceptably slow.

One scenario in which these extreme slowdowns arise is when a higher-order value repeatedly crosses between statically typed and dynamically typed regions of code. Since types are enforced with higher-order contracts, each boundary-crossing wraps the value in a new contract. These layers of indirection can change the asymptotic complexity of a program.

Inspired by Greenberg [2015]'s bold, theoretical claims that contracts (with correct blame) never need more than a constant amount of space, we developed a better runtime representation for contracts that supports a merging operation. That merging operation makes it possible to eliminate the redundancy that results from the above failure mode and brings the overhead of redundant boundary-crossings down to the more reasonable, common case overhead.

We validate our approach by integrating our contract system into Racket [Flatt and PLT 2010] and Typed Racket, a full-fledged, mature gradual typing system. To evaluate the effect of our strategy, we measured our implementation following the state-of-the-art gradual typing benchmarking methods. Our results confirm that our approach yields significant speedups in programs that exhibit the redundant-wrapping pathology, and that our approach's extra bookkeeping imposes only 14% overhead in the worst case. In short, we identify one specific pathology of sound gradual typing and show how to eliminate it. Our results also confirm that more work needs to be done; our work solves only some of the problems that the benchmark programs exhibit.

The rest of the paper starts by illustrating the pathology we target with an example, and introduces some of the constraints a valid cure must satisfy in section 2. Section 3 presents our approach informally, and section 4 and section 5 describe it formally. Section 6 describes the additional constraints that arise from implementing the new checking strategy in an existing, production-quality contract system, and presents key optimizations that help to minimize the cost of the strategy. Section 7 presents the results of our performance experiments, section 8 surveys related work, and section 9 concludes.

2 CONTRACTS GET A BAD WRAP

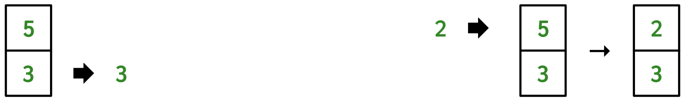
To motivate our improvements to the run-time support for contracts, this section works through the behavior of a Quicksort algorithm written in Racket that uses contracts.

Section 2.1 gives a high level overview of contracts, focusing on the data structures and contracts most relevant to the implementation of Quicksort. Section 2.2 presents the core components of the implementation and analyzes the performance problem that contracts impose on Quicksort. Section 2.3 discusses the concept of blame in detail to understand how it constrains possible solutions to this performance problem. Section 2.4 further illustrates the problem posed by repeated contract wrapping. Finally, section 2.5 reviews the challenges that must be overcome to avoid the build up of contract wrappers.

2.1 Higher-Order Contracts in a Nutshell

The Quicksort implementation we consider accepts a mutable vector and sorts it in place. Before getting into sorting itself, let us first introduce the basics of vectors and contracts in Racket.

The two primary operations on vectors are `vector-ref`, which accesses an element from the vector, and `vector-set!`, which modifies the vector, updating the element at a given position. Vectors and vector operations admit a simple pictorial representation that we use to help explain the performance problem. We show vectors with vertical sequences of boxes, each containing an element of the vector. Vector operations are represented with thick arrows that indicate the flow of a value leaving or entering a vector. In this representation when `vector-ref` is called values leave to the right. Values enter a vector from the left when using `vector-set!`. The image below illustrates these operations on a vector containing the values `5` and `3` at indices `0` and `1`. The thin arrow shows the result of the `vector-set!` operation.



Contracts serve two purposes: ensuring that a value meets a specification and assigning blame to the program component at fault when such a specification is not met. Ordinary contract checking, i.e., for first-order functions, corresponds to the evaluation of two boolean-valued expressions: a pre-condition and a post-condition. Pre-conditions and post-conditions are respectively evaluated just before and after the function is called. When pre-conditions fail, blame is assigned to the caller, when post-conditions fail, blame is assigned to the function itself. This simple story breaks down for higher-order functions and shared mutable state. In particular, simply checking that the contents of a vector satisfy a contract when passed to or returned from a function can lead to incorrect blame assignment. The problem is that a third party may have access to the vector via a shared reference and may mutate the vector separately from any function call or return.

Consider the two program components in figure 1. The `counter.rkt` component provides the function `init`, which returns a reference to a shared vector, and the function `inc!`, which increments the element in the vector (note that `inc!` is buggy; we return to this point later). The definition of `init` uses Racket's `provide/contract` form to attach a contract to the function. In this case, `init` promises that it returns a value satisfying the `(vectorof positive?)` contract. This contract ensures that the return value is a vector containing only positive numbers.

The code in `counter-client.rkt` uses `init` to access the vector, storing it in the local variable `v`. It then uses `inc!` to increment the value in the vector, extracts the value, binding it to `x`, and then takes the reciprocal of `x`. If it were not for the bug in `inc!`, this should print out `1/2`.

Because of the bug, however, `inc!` puts `0` into the vector, which violates the contract that `counter-client.rkt` is relying on. Racket's contract checker catches this violation and correctly counter.rkt

```
#lang racket
(provide/contract
 [init (-> (vectorof positive?))]
 [inc! (-> void?)])

(define shared (vector 1))

(define (init)
  shared)

(define (inc!)
  (define x (vector-ref shared 0))
  (vector-set! shared 0 (- x 1)))
```

counter-client.rkt

```
#lang racket
(require "counter.rkt")

(define v (init))
(inc!)
(define x (vector-ref v 0))
(display (/ 1 x))
```

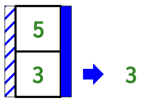
Fig. 1. Using vectors for shared state

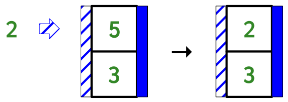
assigns blame to counter.rkt. A simplistic, incorrect contract checker would simply check that the vector contains positive numbers when init returns and do no further checks, leading to a division by zero in this case. Racket's contract checker, however, wraps the vector (just like it does for higher-order functions) and then checks the contracts when the vector's elements are accessed.

Note that this wrapping is an essential aspect of contract checking. In particular, modification of the vector bound to shared by counter.rkt is not required to obey contracts that might accrue on other references to the vector. This separation keeps contracts *optimistic*, ensuring that violations are signaled only when a module actually observes a bad value via a reference that has a contract on it. This behavior of contracts has another benefit, namely it allows references to the same vector to be seen by different parties with different contracts, which enables more opportunities for contract composition. And finally, it ensures that blame assignment is correct [Dimoulas et al. 2012].

An important consequence of that wrapping, however, is that values that accumulate multiple contracts—be it as a result of passing through multiple contract boundaries, or passing through the same boundary multiple times—will also accumulate wrappers. Accessing the original value requires going through all these wrappers, with their associated checks.

We can enrich our pictorial representation of vectors to also represent contracted vectors. We add striped and solid bars to the sides of a vector in order to represent the contract that wraps a vector. These bars represent the contract checks that occur on every vector-ref and vector-set! operation on the vector. The shading of the bars represents the component that is blamed for a contract violation. The striped bars typically represent checks that are the responsibility of client code, whereas the solid bars represent checks for which the component producing the vector is responsible. The images below depict contracted variants of the vectors from the previous diagrams.





2.2 Not-So-Quick Sort

Equipped with this understanding of vectors and contracts, this section turns to the impact of contracts on an implementation of Quicksort. This version of Quicksort sorts a vector of 2D points where a point is represented by a vector of exactly two elements. For simplicity, we use the contract (vectorof (vectorof integer?)) to enforce this invariant instead of a more specific contract, although the development in this section would be the same with more specific contracts supported by Racket's contract system.

```
(provide/contract
  [quicksort! (-> (vectorof (vectorof integer?)) void?)])
;; The details of the implementation of Quicksort are standard
;; and elided in order to focus on those components which are
;; interesting from the perspective of contract checking.
```

Sorting a vector of points requires a comparison function on points. Our Quicksort implementation uses the following lex-order? function, which consumes two points and returns true if the first point is lexicographically smaller than the second.

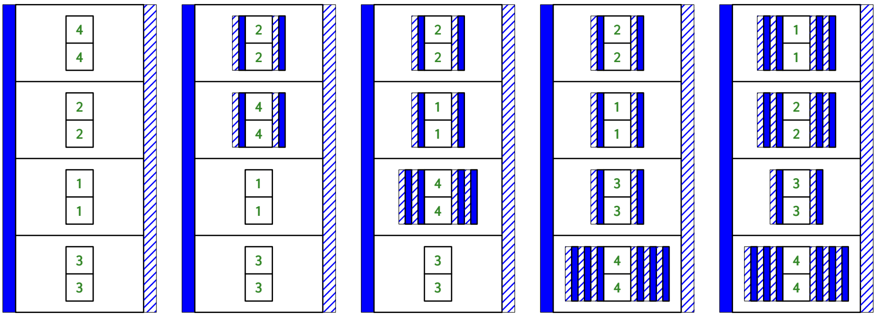
```
(define (lex-order? u v)
  (define u0 (vector-ref u 0))
  (define u1 (vector-ref u 1))
  (define v0 (vector-ref v 0))
  (define v1 (vector-ref v 1))
  (or (< u0 v0)
      (and (= u0 v0) (< u1 v1))))
```

This definition is unremarkable; it simply extracts the values from each point (u and v) and compares them. If u or v happen to have contracts, however, calling lex-order? will trigger contract checks for each call to vector-ref. In particular, if each of u and v each have *n* wrappers requiring that they be vectors of integers, then there will be *4n* checks that the elements are integers.

The other necessary piece of the Quicksort implementation is a function to swap elements of the vector being sorted. The definition of this function, swap!, appears below. Like the definition of lex-order?, the definition of swap! is straightforward, but if the v argument to swap! is wrapped with the (vectorof (vectorof integer?)) contract then every call to swap! will build up wrappers on the values contained in v. More precisely, when vi is defined, a contract check needs to happen to ensure that the extracted value obeys the contract on the vector. In this case, however, the contract is not simply that the contents are integers but it is itself a vector contract. Accordingly, the contract checker wraps the result of vector-ref with a contract. The same thing happens when vi is assigned as one of the elements of the outer vector, resulting in a second contract being attached to vi prior to it being added to the outer vector. This unfortunate build-up of contracts is what causes the terrible performance in this implementation of Quicksort.

```
(define (swap! v i j)
  (define vi (vector-ref v i))
  (define vj (vector-ref v j))
  (vector-set! v i vj)
  (vector-set! v j vi))
```

To see how the problem develops more precisely, consider this series of pictures representing the evolution of the contracts during one run of our Quicksort implementation.



This diagram shows the state of the vector passed to Quicksort as well as its state following each call to `swap!`. Every call to `swap!` adds two layers of contract wrappers to the elements being swapped: one when the element is extracted from the vector, and a second when the element is put back into the vector. Once the vector has been sorted, the maximum number of wrappers on an element of the vector is 6 with an average of 4 wrappers. Somewhat more surprising, however, is the number of times the `integer?` predicate is called as a result of sorting. The implementation of `lex-order?` suggests that there should be approximately four times as many calls to the `integer?` predicate as there are calls to `lex-order?`. In this particular run of Quicksort, the trace produces 6 calls to `lex-order?`, but 168 calls to `integer?`. The extra wrappers cause extra, redundant checks, taking us well beyond simply four additional checks per call to `lex-order?`.

Although these statistics may not seem overly prohibitive towards the use of contracts, consider a larger input to Quicksort. On a randomly ordered vector of 1,000 points, a call to Quicksort can wrap the inner vectors an average of 21 times, call `lex-order?` 11,793 times, and call the `integer?` predicate 134,741,104 times. In this case the performance impact of contracts is severe.

2.3 Wrapping Up Blame

The execution of Quicksort produces a vector of vectors repeatedly wrapped with contracts that perform the same checks. Because they perform identical checks, a single wrapper would suffice to guard each inner vector in this case. Intuitively, it seems that keeping only the first wrapper would alleviate the performance problem of contracts in general. Unfortunately, keeping only that one would be incorrect. The issue lies in managing the information required to correctly assign blame when a violation occurs. If the contract system were to keep only one of the wrappers, blame assignment could shift from a guilty module to an innocent one.

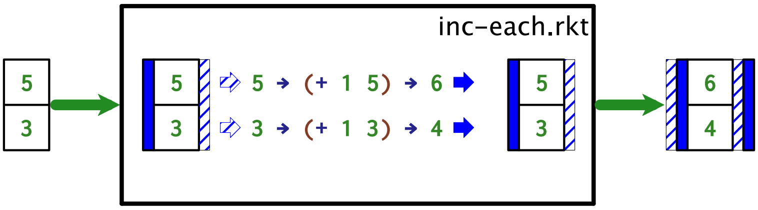
To better understand the issue of blame assignment and how it interacts with multiple wrappers, consider the contrived program in figure 2. It is not a well-designed vector program, but it allows us to focus on the essence of proper blame assignment for contracts on (possibly shared) vectors. The `inc-each.rkt` component provides a function `inc-each!` that takes a reference to a vector, increments each element of the vector by one and returns a reference to the given vector. A vector passed to `inc-each!` receives two contract wrappers, the first when passed as an argument to the function and the second when the function returns. Each of these wrappers corresponds to different configurations of blame, however. Within the body of `inc-each!`, the `inc-each-client.rkt` component is blamed if the vector produces a non-integer value (when `vector-ref` is called). When `inc-each!` returns and the second wrapper is added, `inc-each!` becomes responsible for ensuring that the resulting vector contains integers. Mutation reverses the flow of values and thus reverses the responsibilities [Strickland et al. 2012]. Within the body of `inc-each!`, the function itself is responsible for inserting only integers into the vector and when the function returns the client also gains this responsibility. The image below depicts the client's call to `inc-each!`, where

```
                                         inc-each.rkt

 1 | #lang racket
 2 | (provide/contract
 3 |  [inc-each!
 4 |   (-> (vectorof integer?)
 5 |       (vectorof integer?))])
 6 |
 7 | (define (inc-each! v)
 8 |   (for ([i (in-range (vector-length v))])
 9 |     (define vi (vector-ref v i))
10 |     (vector-set! v i (+ 1 vi)))
11 |   v)
```

```
                        inc-each-client.rkt

 1 | #lang racket
 2 | (require "inc-each.rkt")
 3 |
 4 | (define v1 (vector 5 3))
 5 | (define v2 (inc-each! v1))
 6 | (vector-set! v1 0 "bad")
 7 | (vector-ref v2 0)
 8 |
 9 |
10 |
```

Fig. 2. Interaction between mutable state and blame assignment using vector contracts

the solid blue wrappers represent checks for which `inc-each.rkt` is responsible and the striped blue wrappers represent checks for which `inc-each-client.rkt` is responsible.



Consider the code in `inc-each-client.rkt`. On line 4 the client creates a new vector, `v1`, and passes it to `inc-each!` on line 5, binding the result to `v2`. Note that the vector `v1` is an uncontracted alias of the contracted vector `v2` returned by `inc-each!`. On line 6 the client modifies the vector at index `0` to contain the string `"bad"`. Because this operation is performed on the original, uncontracted reference to the vector, there is no contract violation. Finally, the client attempts to access the element at index `0` of `v2`. The vector `v2` has passed through two boundaries so the value that is extracted from the vector now must travel through those boundaries too. In this case, the first contract check is the client's responsibility and a contract violation is raised blaming `inc-each-client.rkt` for producing the value `"bad"` instead of an integer, just as we would hope.

Despite the fact that the two contracts on `v2` check the same invariants, they each assign blame to different components. Thus, removing the inner wrapper would change the blame assignment in this example, incorrectly blaming `inc-each.rkt`. Similarly, removing the outer wrapper would shift the blame from the client to the server for failing to insert integers into the vector `v2`. Indeed, it is commonly the case that the blame associated with incorrectly mutating a vector is different than the blame associated with extracting an incorrect value from a vector.

Even worse, if there are multiple distinct contract wrappers on a value, but with some of the checks duplicated, removing redundant wrappers could change the order in which contracts are checked, which might result in incorrect blame, but could also result in failures in the code that does the contract checking itself. In particular, the code that implements the checks for one contract might depend on another contract being satisfied. In short, simply removing contract wrappers is not an acceptable solution for the problem of contract build-up.

```
bubble.rkt

#lang racket
(define bubble/c
  (recursive-contract
   (vector/c bubble/c)
   #:chaperone))

(define vec (vector #f))

(provide/contract
 [vec bubble/c])
```

```
bubble-client.rkt

#lang racket
(require "bubble.rkt")

(define (self-assign n)
  (let loop ([n n]
             [vec vec])
    (unless (zero? n)
      (vector-set! vec 0 vec)
      (loop (- n 1)
            (vector-ref vec 0)))))
```

Fig. 3. Exponentially Many Wrappers

2.4 How Bad Can It Get?

These additional contract wrappers can change the asymptotic complexity of the program. For example, if each iteration of a loop adds a contract to a value and then checks all of the contracts, the checking adds an extra factor of n. But it can get significantly worse than that.

Consider the program in figure 3, an example of Takikawa et al. [2015] adapted to vector contracts. It shows a program where the wrapping grows exponentially with the number of iterations of a loop. To understand why, first consider the initial export from `bubble.rkt`. It is a vector with a single wrapper that holds the `bubble/c` contract. Next, imagine what happens on the first iteration of the loop. The vector is inserted into itself, which means that the reference inside the vector now has two `bubble/c` wrappers. Then it is removed, which adds a third wrapper. On the second iteration of the loop, we start with a reference that has three wrappers. When we insert it into itself, each of those three wrappers puts another copy on, for a total of six wrappers. Then, when the reference is extracted, another three wrappers are put on, and we go around the loop again, this time with a reference that has nine wrappers. Each time around the loop, we are (asymptotically) tripling the number of wrappers, leading to an exponential slowdown. While this example is distilled to its essence, we have seen this exponential behavior in real programs.

2.5 Wrapping up the Bad Wrap

Overall this section illustrates, using Racket's vectors, how wrappers can accumulate and lead to a large number of redundant contract checks, with all the performance overhead that ensues. It is clear that reducing this performance impact requires a way to both reduce the amount of wrappers attached to a given value and eliminate redundant contract checks. Unfortunately, the need to preserve correct blame defeats the straightforward approach of removing contract wrappers. These constraints demand a solution that can reduce the number of wrappers and elide unnecessary contract checks without sacrificing correct blame assignment.

3 UNWRAPPING THE SOLUTION

This section presents the two central aspects of our technique for reducing the performance impact of contracts: a new representation of contracts and a way to merge contracts to avoid redundant contract checks that our representation enables. Together, these two innovations prevent contract wrappers from building up on values by dropping contracts that are guaranteed never to fail. Applied to the Quicksort example from the previous section, this approach limits the buildup of contracts on any vector to a single wrapper. Section 3.1 introduces the new data structure square-each.rkt

```
#lang racket
(provide/contract
 [square-each
   (-> (vectorof integer?)
       (vectorof nonnegative-integer?))])
(define (square-each v)
  (for ([i (in-range (vector-length v))])
    (define vi (vector-ref v i))
    (vector-set! v i (* vi vi)))
  v)
```

square-each-client.rkt

```
#lang racket
(require "square-each.rkt")
(square-each (vector -5 3))
```

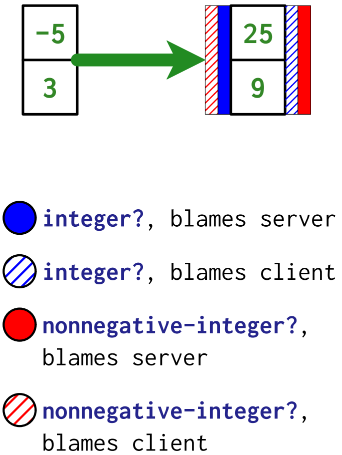

-5
3
25
9
● integer?, blames server
▨ integer?, blames client
● nonnegative-integer?, blames server
▨ nonnegative-integer?, blames client

Fig. 4. Running example: squaring the elements of a mutable vector

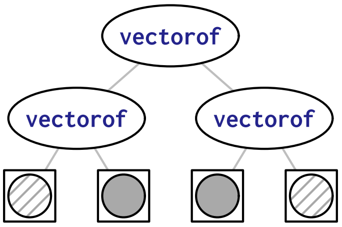
representation of contracts and section 3.2 describes an operation that merges the structures, thus eliminating redundant contracts.

3.1 Representing Contracts

A natural implementation of contracts is to add wrappers to each vector where the wrapper consists of a reference to the original vector, plus some information about the contract. This representation is, indeed, the one that Racket's contract system used before this work. Unfortunately, such an implementation does not provide a straightforward means of manipulating multiple contracts on the same value at once, such as combining multiple contract wrappers into one. Our solution is to represent contracts as explicit data structures that make such manipulations natural.

Our representation for a contract is a tree whose interior nodes correspond to `vector` contracts and whose leaf nodes correspond to the contracts on the elements of the innermost vectors. Each interior node has two children; the right child contains the information necessary to perform contract checking when accessing elements and the left child contains the information necessary to perform contract checking when modifying the vector.

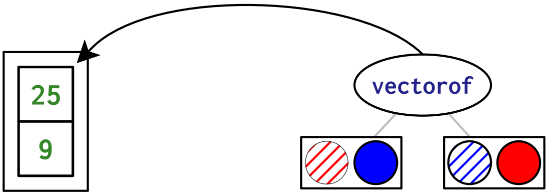
As an example, here is our representation of the vector of vectors contract from the Quicksort example of the previous section.



The root of the tree has two children, indicating that elements that are accessed from the outer vector and stored in it must be vectors; accordingly there are four leaf nodes. Reading from left to right, the leaf nodes represent the contracts for: mutating elements of a vector inserted into the external vector, accessing elements from such a vector, mutating elements of a vector that is extracted from the outer vector, and accessing elements from such a vector. Notice that the leaf nodes are colored and shaded in two distinct ways. These markings represent blame, just as the striped and solid colors in the previous section did. Blame assignment follows the flow of values, so the bottom right, where elements of the inner vector are flowing out of a vector extracted from the vector has the same blame as the bottom left, where values assigned to a vector that was assigned into the outer vector.

Figure 4 shows some code that serves as a running example for the remainder of this section. The `square-each.rkt` component presents the implementation of a function which, when given a vector of integers, produces a vector of non-negative integers by squaring each element of the vector. Because the function `square-each` returns a reference to its argument, the vector it returns is wrapped in two layers of contracts as the image in the upper right of the figure shows, using the notation introduced in the previous section.

Here is a picture of the result vector and its contract:



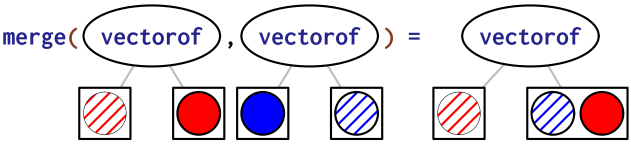
The contract is represented as a reference to the vector (shown with an arrow in the picture) and a tree. In this case, because the vector has passed through two boundaries, the vector contract has two children in each leaf node. The key in figure 4 indicates which checks these correspond to. They are checked in order, from left to right. Note that the order of the colors are reversed between the children, corresponding to the reversal of the flow of values for mutation and access of vectors.

In the general case, the leaves contain multiple contracts, capturing the multiple contract boundaries that the vector has passed through.

3.2 Merging Contracts

Of course, simply collecting all of the contracts in leaf nodes does not solve the performance problem. What we need is a mechanism to limit the redundancy that appears in the leaves. For this purpose, we use an operation called `merge`. It accepts two trees and combines them into a single tree, eliminating the redundancy in the contracts in the process.

When merging the trees, maintaining correct blame assignment for the remaining contracts is subtle. To see how the blame plays out, consider how the merging operation is performed on the contracts we saw in the previous subsection, focusing on the moment when the vector is returned by `square-each`. The two contracts in play then are the two on the left-hand side of this equation. The right-hand side shows the result contract.



The first argument to the merging operation is the new contract to be added to the vector (shown with red contracts in the leaves, following the key in figure 4) and the second one is the contract that is already on the vector.

A simple `merge` function would simply keep all four of the contracts on the leaves, as was shown in the previous section. One of them is redundant, however, and can be dropped. To understand why one (and only one) is redundant, it is important to consider the order in which the contracts are checked, factoring in the blame assignment. First, notice that the red contracts are "outside" the blue contracts, meaning that accesses to the vector will check the blue contract first, but mutations to the vector will check the red contract first. The checks happen in this order so that they follow the flow of values in the program. In particular a value that is already inside the vector first exits from the blue wrapping of the vector and, if that contract check succeeds, exits through the red contract check. In contrast, when a value is inserted into the vector, it first passes through the red contract and, if that check succeeds, then passes through the blue contact.

Since the red contract subsumes the blue one, by checking the red one first, we know that the blue one cannot fail. Thus, we can eliminate the solid blue circle in the left child of the merged result. In the other order, however, when we have checked the blue one, we do not yet know if the red one will fail or not; the red contract is more stringent than the blue one. Therefore, we have to keep both circles in the right subtree of the merged result—either of them may fail and raise blame.

In general, when we merge two contract trees, some of the leaves in the tree may be redundant and will be dropped in this manner. For the Quicksort example from section 2, our library never has more than one contract in any leaf node.

With blame properly taken into account we still need a procedure to determine, in the general case, when a specific contract is actually redundant. For this purpose, Racket provides the function `contract-stronger?`. It accepts two contracts as arguments and returns true when the first contract is guaranteed to accept no more values than the second contract. This function is conservative, however, as Racket's contract system allows the programmer to supply arbitrary predicates. In practice, most of the interesting flat contracts in most programs either repeatedly use the same predicate (in which case `contract-stronger?` trivially succeeds), or use combinators that `contract-stronger?` understands.

4 A MODEL FOR CONTRACT WRAPPING, WITH REDUNDANCY

In order to more precisely present the solution sketched out in the previous section, the following two sections present formal models of contract checking. The models are designed for exposition; they hew closely to our implementation, providing a precise picture of which contracts we check and how exactly we check them. This section matches how Racket checked contracts before our work, possibly creating many wrappers around vectors. The model omits functions (although our implementation handles them) because vectors are sufficient to cover all of the essential issues.

The left-hand side of figure 5 presents the syntax for a core language with vectors and contracts. Programs, **P**, comprise a sequence of simplified module definitions. Each one is meant to suggest a module in Racket that exports the defined function, **d**, with the contract (-> **ctc** ... **ctc**) and implicitly imports other modules to bind the variables from other modules that it refers to. The notation for the contract consists of an -> (meant to evoke Racket's function contract combinator, but it is fixed syntax in this model), a sequence of contracts, **ctc** ..., describing the inputs, and finally a single contract describing the result of the function. Each definition, **d**, gives the function its name, **f**, accepts the parameters **x** ..., and has a body **e**. Expressions, **e**, include real number literals, variables, let expressions, vector and numeric operations, and function calls. Contracts, **ctc**, include a `vectorof` contract form as well as the flat contracts `any/c`, which accepts all values, and `real-in`, which accepts real numbers within a specified range.

$$\begin{aligned}
\mathbf{P} &::= (\texttt{program}\ \mathbf{m}\ \ldots) \\
\mathbf{m} &::= (\texttt{module}\ (\texttt{->}\ \mathbf{ctc}\ \ldots\ \mathbf{ctc})\ \mathbf{d}) \\
\mathbf{d} &::= (\texttt{define}\ (\mathbf{f}\ \mathbf{x}\ \ldots)\ \mathbf{e}) \\
\mathbf{e} &::= \mathbf{r} \mid \mathbf{x} \mid (\texttt{if0}\ \mathbf{e}\ \mathbf{e}\ \mathbf{e}) \\
&\quad \mid (\texttt{let}\ ([\mathbf{x}\ \mathbf{e}]\ \ldots)\ \mathbf{e}) \\
&\quad \mid (\texttt{vector}\ \mathbf{e}\ \ldots) \mid (\mathbf{vec\text{-}op}\ \mathbf{e}\ \ldots) \\
&\quad \mid (\mathbf{op}\ \mathbf{e}\ \ldots) \mid (\mathbf{f}\ \mathbf{e}\ \ldots)^{\mathbf{f}} \\
\mathbf{vec\text{-}op} &::= \texttt{vector-ref} \\
&\quad \mid \texttt{vector-set!} \\
&\quad \mid \texttt{vector-length} \\
\mathbf{op} &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{<} \\
\mathbf{r} &::= \text{real numbers} \\
\mathbf{n} &::= \mathbf{natural} \\
\mathbf{x} &::= \text{identifiers} \\
\mathbf{f}, \mathbf{g} &::= \mathbf{function\text{-}names} \\
\mathbf{function\text{-}names} &::= \text{identifiers} \\
\mathbf{ctc} &::= (\texttt{vectorof}\ \mathbf{ctc}) \mid \mathbf{flat\text{-}ctc} \\
\mathbf{flat\text{-}ctc} &::= (\texttt{real-in}\ \mathbf{r}\ \mathbf{r}) \mid \texttt{any/c}
\end{aligned}$$

$$\begin{aligned}
\mathbf{e} &::= \ldots\ldots \mid \boldsymbol{\sigma} \mid (\texttt{blame}\ \mathbf{f}) \mid (\texttt{mon}^{\mathbf{f}}_{\mathbf{f}}\ \mathbf{ctc}\ \mathbf{e}) \\
\boldsymbol{\Sigma} &::= \text{finite maps from } \boldsymbol{\sigma} \text{ to } (\texttt{vector}\ \mathbf{v}\ \ldots) \\
\mathbf{v} &::= \mathbf{r} \mid \boldsymbol{\sigma}^{*} \\
\boldsymbol{\sigma}^{*} &::= (\texttt{mon}^{\mathbf{f}}_{\mathbf{f}}\ (\texttt{vectorof}\ \mathbf{ctc})\ \boldsymbol{\sigma}^{*}) \mid \boldsymbol{\sigma} \\
\boldsymbol{\sigma} &::= \text{store locations} \\
\mathbf{E} &::= [\,] \\
&\quad \mid (\texttt{if0}\ \mathbf{E}\ \mathbf{e}\ \mathbf{e}) \\
&\quad \mid (\texttt{let}\ ([\mathbf{x}\ \mathbf{v}]\ \ldots\ [\mathbf{x}\ \mathbf{E}]\ [\mathbf{x}\ \mathbf{e}]\ \ldots)\ \mathbf{e}) \\
&\quad \mid (\texttt{vector}\ \mathbf{v}\ \ldots\ \mathbf{E}\ \mathbf{e}\ \ldots) \\
&\quad \mid (\mathbf{vec\text{-}op}\ \mathbf{v}\ \ldots\ \mathbf{E}\ \mathbf{e}\ \ldots) \\
&\quad \mid (\mathbf{op}\ \mathbf{v}\ \ldots\ \mathbf{E}\ \mathbf{e}\ \ldots) \\
&\quad \mid (\mathbf{f}\ \mathbf{v}\ \ldots\ \mathbf{E}\ \mathbf{e}\ \ldots)^{\mathbf{f}} \\
&\quad \mid (\texttt{mon}^{\mathbf{f}}_{\mathbf{f}}\ \mathbf{ctc}\ \mathbf{E})
\end{aligned}$$

Fig. 5. A core language with vectors

Each module, **m**, establishes a contract boundary between the module itself and the code that uses its exported function. To properly assign blame, expressions that can fail or raise a blame error are labeled, **f**, with the name of the function in which they appear.[1]

The right-hand side of figure 5 presents the extensions to the core language to support evaluation. The grammar of expressions is extended with store locations, **σ**, blame expressions, and the `mon` form, which enforces contracts. A store, **Σ**, maps store locations to vectors of values. Values include real number literals, **r**, and possibly-wrapped store locations, **σ***. Each possibly-wrapped store location may be a `mon` expression with a vector contract, or it may be a normal store location. The set of evaluation contexts, **E**, are standard for a call-by-value language.

The reduction semantics for the language operates over triples of the form ⟨**Σ**, **P**, **e**⟩. Expressions reduce in the context of a store, **Σ**, and a set of module definitions, **P**. Figure 6 presents the reduction rules for non-contract forms of the language in figure 5. The [let-subst] rule substitutes the values of let-bound variables in the body expression. The [if0-0] and [if0-else] rules cover `if0` expressions. The [delta] rule delegates to the δ function (not shown) to handle the arithmetic operations. The [v-ref], [v-set!], and [v-len] rules handle accessing, mutating, and computing the length of a vector, respectively. They delegate to three metafunctions—**lookup**, **update**, and **length** (not shown)—that manipulate the store and an address in the corresponding manner. The [v-alloc] rule finds a fresh address and extends the store **Σ** with it.

Figure 7 presents the reduction rules for contracts. The [v-ref-ctc], [v-set!-ctc], and [v-len-ctc] rules cover accessing, mutating, and computing the length of contracted vectors. When accessing an element, `vector-ref` performs the operation on the uncontracted value and then adds a monitor around the extracted element. When mutating the vector, the newly inserted element is guarded with the element contract. Note that the blame labels are switched, following the reversal of the flow of values. The `vector-length` operation ignores the contract.

[1] This labeling mimics how the implementation behaves; it annotates cross-module variable references with blame information as part of the compilation process.

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{let}\ ([\mathbf{x}\ \mathbf{v}]\ ...)\ \mathbf{e})]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[\mathbf{e}\{\mathbf{x} \leftarrow \mathbf{v}, ...\}]\rangle \qquad \text{[let-subst]}$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{if0}\ 0\ \mathbf{e}_1\ \mathbf{e}_2)]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[\mathbf{e}_1]\rangle \qquad \text{[if0-0]}$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{if0}\ \mathbf{v}\ \mathbf{e}_1\ \mathbf{e}_2)]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[\mathbf{e}_2]\rangle \ \text{ where } \mathbf{v} \neq 0 \qquad \text{[if0-else]}$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\mathbf{op}\ \mathbf{r}\ ...)]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[\boldsymbol{\delta}[\![(\mathbf{op}\ \mathbf{r}\ ...)]\!]]\rangle \qquad \text{[delta]}$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{vector-ref}\ \boldsymbol{\sigma}\ \mathbf{n})]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[\mathbf{v}]\rangle \ \text{ where } \mathbf{n} < \mathbf{length}[\![\mathbf{\Sigma}, \boldsymbol{\sigma}]\!], \mathbf{v} = \mathbf{lookup}[\![\mathbf{\Sigma}, \boldsymbol{\sigma}, \mathbf{n}]\!] \qquad \text{[v-ref]}$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{vector-set!}\ \boldsymbol{\sigma}\ \mathbf{n}_{index}\ \mathbf{v}_{set})]\rangle \longrightarrow \langle \mathbf{\Sigma}', \mathbf{P}, \mathbf{E}[\mathbf{v}]\rangle \qquad \text{[v-set!]}$$
$$\text{where } \mathbf{n}_{index} < \mathbf{length}[\![\mathbf{\Sigma}, \boldsymbol{\sigma}]\!], \langle \mathbf{\Sigma}', \mathbf{v}\rangle = \mathbf{update}[\![\mathbf{\Sigma}, \boldsymbol{\sigma}, \mathbf{n}_{index}, \mathbf{v}_{set}]\!]$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{vector-length}\ \boldsymbol{\sigma})]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[\mathbf{length}[\![\mathbf{\Sigma}, \boldsymbol{\sigma}]\!]]\rangle \qquad \text{[v-len]}$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{vector}\ \mathbf{v}\ ...)]\rangle \longrightarrow \langle \mathbf{\Sigma}', \mathbf{P}, \mathbf{E}[\boldsymbol{\sigma}]\rangle \ \text{ where } \langle \mathbf{\Sigma}', \boldsymbol{\sigma}\rangle = \mathbf{alloc}[\![\mathbf{\Sigma}, (\texttt{vector}\ \mathbf{v}\ ...)]\!] \qquad \text{[v-alloc]}$$

Fig. 6. Reduction rules for non-contract forms

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{vector-length}\ (\texttt{mon}^{\mathbf{f}}_{\mathbf{g}}\ (\texttt{vectorof}\ \mathbf{ctc})\ \boldsymbol{\sigma}^*))]\rangle \longrightarrow \qquad \text{[v-len-ctc]}$$
$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{vector-length}\ \boldsymbol{\sigma}^*)]\rangle$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{vector-ref}\ (\texttt{mon}^{\mathbf{f}}_{\mathbf{g}}\ (\texttt{vectorof}\ \mathbf{ctc})\ \boldsymbol{\sigma}^*)\ \mathbf{v})]\rangle \longrightarrow \qquad \text{[v-ref-ctc]}$$
$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}^{\mathbf{f}}_{\mathbf{g}}\ \mathbf{ctc}\ (\texttt{vector-ref}\ \boldsymbol{\sigma}^*\ \mathbf{v}))]\rangle$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{vector-set!}\ (\texttt{mon}^{\mathbf{f}}_{\mathbf{g}}\ (\texttt{vectorof}\ \mathbf{ctc})\ \boldsymbol{\sigma}^*)\ \mathbf{v}_{index}\ \mathbf{v}_{set})]\rangle \longrightarrow \qquad \text{[v-set!-ctc]}$$
$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{vector-set!}\ \boldsymbol{\sigma}^*\ \mathbf{v}_{index}\ (\texttt{mon}^{\mathbf{g}}_{\mathbf{f}}\ \mathbf{ctc}\ \mathbf{v}_{set}))]\rangle$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\mathbf{f}\ \mathbf{v}\ ...)^{\mathbf{g}}]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{let}\ ([\mathbf{x}\ (\texttt{mon}^{\mathbf{g}}_{\mathbf{f}}\ \mathbf{ctc}_{dom}\ \mathbf{v})]\ ...)\ (\texttt{mon}^{\mathbf{f}}_{\mathbf{g}}\ \mathbf{ctc}_{rng}\ \mathbf{e}))]\rangle \qquad \text{[call]}$$
$$\text{where } \mathbf{f} \neq \mathbf{g}, (\texttt{module}\ (\texttt{->}\ \mathbf{ctc}_{dom}\ ...\ \mathbf{ctc}_{rng})\ (\texttt{define}\ (\mathbf{f}\ \mathbf{x}\ ...)\ \mathbf{e})) \in \mathbf{P}$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\mathbf{f}\ \mathbf{v}\ ...)^{\mathbf{f}}]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[\mathbf{e}\{\mathbf{x} \leftarrow \mathbf{v}, ...\}]\rangle \qquad \text{[call-self]}$$
$$\text{where } (\texttt{module}\ (\texttt{->}\ \mathbf{ctc}_{dom}\ ...\ \mathbf{ctc}_{rng})\ (\texttt{define}\ (\mathbf{f}\ \mathbf{x}\ ...)\ \mathbf{e})) \in \mathbf{P}$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}^{\mathbf{f}}_{\mathbf{g}}\ \texttt{any/c}\ \mathbf{v})]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[\mathbf{v}]\rangle \qquad \text{[mon-any/c]}$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}^{\mathbf{f}}_{\mathbf{g}}\ (\texttt{real-in}\ \mathbf{r}_{lo}\ \mathbf{r}_{hi})\ \mathbf{v})]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[\mathbf{v}]\rangle \ \text{ where } \mathbf{v} \in [\mathbf{r}_{lo}, \mathbf{r}_{hi}] \qquad \text{[mon-real-in-pass]}$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}^{\mathbf{f}}_{\mathbf{g}}\ (\texttt{real-in}\ \mathbf{r}_{lo}\ \mathbf{r}_{hi})\ \mathbf{v})]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{blame}\ \mathbf{f})]\rangle \ \text{ where } \mathbf{v} \notin [\mathbf{r}_{lo}, \mathbf{r}_{hi}] \qquad \text{[mon-real-in-fail]}$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}^{\mathbf{f}}_{\mathbf{g}}\ (\texttt{vectorof}\ \mathbf{ctc})\ \mathbf{r})]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{blame}\ \mathbf{f})]\rangle \qquad \text{[mon-vectorof-fail]}$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{blame}\ \mathbf{g})]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, (\texttt{blame}\ \mathbf{g})\rangle \ \text{ where } \mathbf{E} \neq \texttt{[]} \qquad \text{[lift-blame]}$$

Fig. 7. Reduction rules for contract forms

The function application rule, [call], also performs contract checking. It reduces to a `let` expression that checks each argument contract and checks that the result satisfies the codomain contract. The side-condition guarantees that the call is crossing a module boundary, and thus contracts must be

$$
\begin{array}{rl@{\qquad}rl}
\mathbf{e} ::= & \ldots\ldots \mid (\text{mon } \mathbf{cctc}\ \mathbf{e}) & \mathbf{ctc} ::= & \ldots\ldots \mid \mathbf{cctc} \\
\mathbf{v} ::= & \mathbf{r} \mid \boldsymbol{\sigma} \mid (\text{mon } \mathbf{cvec}\ \boldsymbol{\sigma}) & \mathbf{cvec} ::= & (\texttt{cvectorof } \mathbf{cctc}\ \mathbf{cctc})^{\mathbf{f}} \\
\mathbf{E} ::= & \ldots\ldots \mid (\text{mon } \mathbf{cctc}\ \mathbf{E}) & \mathbf{cctc} ::= & \mathbf{cvec} \mid (\texttt{leaf } \mathbf{le}\ \ldots) \\
 & & \mathbf{le} ::= & \mathbf{flat\text{-}ctc}^{\mathbf{f}} \mid \mathbf{cvec}
\end{array}
$$

Fig. 8. Extensions to the vector language to support collapsible contracts.

checked. The [call-self] rule covers the situation where no boundary is crossed, and thus no contract checking happens [Findler and Felleisen 2002].

The remaining rules cover `mon` expressions. The [mon-any/c] rule drops `any/c` contracts, because they always succeed. The `real-in` contract succeeds when the value under contract is a real number in the right range and otherwise raises a blame error in rules [mon-real-in-pass] and [mon-real-in-fail] respectively. The [mon-vectorof-fail] rule shows that the application of a vector contract to a non-vector raises blame. There is no reduction rule for the application of a vector contract to a store location because such an expression is already a value. The final rule, [lift-blame], handles reductions for blame expressions. When a blame expression is encountered in the process of evaluation the current context is discarded and that blame expression is the result of the entire program.

5 A SECOND MODEL FOR CONTRACT WRAPPING, SANS REDUNDANCY

The model in section 4 faithfully presents the semantics of vector contract checking, but suffers from the wrapper buildup problem described in section 2. This section introduces a formal model of the contract trees from section 3 in order to precisely explain the construction of the data structure and its merge operation. This language is significantly smaller than the language our implementation supports, but the description here captures exactly how our implementation avoids redundant contracts and provides a faithful predictor for specific programs.

Figure 8 presents the extensions to the language of section 4 necessary to support contract trees. The values, **v**, are now real number literals, store locations, and store locations guarded by a single contract tree. Unlike the language in section 4 where contract wrappers might pile up, this language ensures that a value has at most one contract wrapper.

The syntax of contracts is extended to include collapsible contracts, **cctc**, representing the contract trees described in section 3. A collapsible contract is either an interior node representing a `vectorof` contract (the **cvec** non-terminal) or a leaf containing a list of leaf elements, **le**. The interior nodes representing vector contracts hold two children, each a collapsible contract. To match the structure of the diagrams from section 2 and section 3, the left child is the contract for `vector-set!` and the right child is the contract for `vector-ref`. The **le** nonterminal represents leaves, and includes flat contracts with attached blame labels, as expected, but it also includes vector contracts. A vector contract can appear in a leaf node as a result of merging when it meets a non-vector contract.

The other significant change to the syntax of the language is the addition of a `mon` construct without blame labels. Instead, the blame labels are on the nodes of contract trees.

Figure 9 presents the reduction rules for the language with collapsible contracts. The metafunctions used by the reduction rules appear in figure 10.

The [mon-vectorof-base] rule handles the conversion between a `vectorof` contract and the equivalent collapsible contract via the **reify-contract** metafunction. Given a vector contract and two blame parties, **reify-contract** builds a collapsible contract by recursively constructing collapsible contracts for each configuration of blame and combining them into a collapsible contract. Flat contracts are converted directly into leaf nodes.

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}^{\mathbf{f}}_{\mathbf{g}}\ (\texttt{vectorof}\ \mathbf{ctc})\ \boldsymbol{\sigma})]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}\ \mathbf{cvec}\ \boldsymbol{\sigma})]\rangle \qquad \text{[mon-vectorof-base]}$$
$$\text{where } \mathbf{cvec} = \mathbf{reify\text{-}contract}[\![(\texttt{vectorof}\ \mathbf{ctc}), \mathbf{f}, \mathbf{g}]\!]$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}^{\mathbf{f}}_{\mathbf{g}}\ (\texttt{vectorof}\ \mathbf{ctc})\ (\texttt{mon}\ \mathbf{cvec}_{1}\ \boldsymbol{\sigma}))]\rangle \longrightarrow \qquad \text{[mon-vectorof-comp]}$$
$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}\ \mathbf{cvec}_{2}\ (\texttt{mon}\ \mathbf{cvec}_{1}\ \boldsymbol{\sigma}))]\rangle \qquad \text{where } \mathbf{cvec}_{2} = \mathbf{reify\text{-}contract}[\![(\texttt{vectorof}\ \mathbf{ctc}), \mathbf{f}, \mathbf{g}]\!]$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}\ \mathbf{cvec}_{2}\ (\texttt{mon}\ \mathbf{cvec}_{1}\ \boldsymbol{\sigma}))]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}\ \mathbf{cvec}_{m}\ \boldsymbol{\sigma})]\rangle \qquad \text{[mon-vectorof-merge]}$$
$$\text{where } \mathbf{cvec}_{m} = \mathbf{merge}[\![\mathbf{cvec}_{2}, \mathbf{cvec}_{1}]\!]$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{vector-length}\ (\texttt{mon}\ (\texttt{cvectorof}\ \mathbf{cctc}_{s}\ \mathbf{cctc}_{g})^{\mathbf{f}}\ \boldsymbol{\sigma}))]\rangle \longrightarrow \qquad \text{[v-len-ctc]}$$
$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{vector-length}\ \boldsymbol{\sigma})]\rangle$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{vector-ref}\ (\texttt{mon}\ (\texttt{cvectorof}\ \mathbf{cctc}_{s}\ \mathbf{cctc}_{g})^{\mathbf{f}}\ \boldsymbol{\sigma})\ \mathbf{v})]\rangle \longrightarrow \qquad \text{[v-ref-ctc]}$$
$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}\ \mathbf{cctc}_{g}\ (\texttt{vector-ref}\ \boldsymbol{\sigma}\ \mathbf{v}))]\rangle$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{vector-set!}\ (\texttt{mon}\ (\texttt{cvectorof}\ \mathbf{cctc}_{s}\ \mathbf{cctc}_{g})^{\mathbf{f}}\ \boldsymbol{\sigma})\ \mathbf{v}_{index}\ \mathbf{v}_{set})]\rangle \longrightarrow \qquad \text{[v-set!-ctc]}$$
$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{vector-set!}\ \boldsymbol{\sigma}\ \mathbf{v}_{index}\ (\texttt{mon}\ \mathbf{cctc}_{s}\ \mathbf{v}_{set}))]\rangle$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}\ (\texttt{cvectorof}\ \mathbf{cctc}_{s}\ \mathbf{cctc}_{g})^{\mathbf{f}}\ \mathbf{r})]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{blame}\ \mathbf{f})]\rangle \qquad \text{[mon-cctc-fail]}$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}\ (\texttt{leaf})\ \mathbf{v})]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[\mathbf{v}]\rangle \qquad \text{[leaf-0]}$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}\ (\texttt{leaf}\ \mathbf{flat\text{-}ctc}^{\mathbf{f}}\ \mathbf{le}\ ...)\ \mathbf{v})]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}\ (\texttt{leaf}\ \mathbf{le}\ ...)\ (\texttt{mon}^{\mathbf{f}}_{\mathbf{g}}\ \mathbf{flat\text{-}ctc}\ \mathbf{v}))]\rangle \qquad \text{[flat]}$$

$$\langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}\ (\texttt{leaf}\ \mathbf{cvec}\ \mathbf{le}\ ...)\ \mathbf{v})]\rangle \longrightarrow \langle \mathbf{\Sigma}, \mathbf{P}, \mathbf{E}[(\texttt{mon}\ (\texttt{leaf}\ \mathbf{le}\ ...)\ (\texttt{mon}\ \mathbf{cvec}\ \mathbf{v}))]\rangle \qquad \text{[leaf-cvec]}$$

Fig. 9. Reduction rules for collapsible contracts

The [mon-vectorof-comp] rule handles the application of a traditional vector contract to a value guarded by a collapsible contract. The outer contract application is converted into the application of a collapsible contract just as in the [mon-vectorof-base] rule.

The [mon-vectorof-merge] rule implements the contract merging process described in section 3. When one mon expression is nested in another, their contracts are merged and they are replaced with a single mon expression guarding a value with a collapsible contract.

The [v-ref-ctc], [v-set!-ctc], and [v-len-ctc] rules mirror those presented in section 4 for vector operations on contracted vectors. For `vector-ref` operations, the operation is performed on the uncontracted vector and the result is guarded with the right child of the contract tree. Similarly for `vector-set!` operations, the modification is performed on the uncontracted vector and the value to be inserted is guarded with the contract stored in the left child of the contract tree. The `vector-length` operation is performed on the uncontracted vector as in the previous section. The [mon-cctc-fail] rule handles the application of a collapsible contract to a non-vector value by simply signaling a blame error.

Finally, the rules [leaf-0], [flat], and [leaf-cvec] handle contract application of leaf nodes. Applying the contract corresponding to a leaf node sequentially applies each contract stored in the leaf. The **merge** metafunction tries to ensure that leaves do not contain redundant flat contracts.

Figure 10 contains the **merge** metafunction. It consumes two collapsible contracts and produces a new collapsible contract. The first argument of **merge** is the newer contract being attached to a value, and the second is the original contract, already attached to the value. The clauses of the **merge** metafunction are ordered to resolve any overlap between the cases.

$$\begin{aligned}
&\mathbf{reify\text{-}contract}\llbracket(\texttt{vectorof}\ \mathbf{ctc}), \mathbf{f}, \mathbf{g}\rrbracket &&= (\texttt{cvectorof}\ \mathbf{reify\text{-}contract}\llbracket\mathbf{ctc}, \mathbf{g}, \mathbf{f}\rrbracket \\
& && \qquad\qquad \mathbf{reify\text{-}contract}\llbracket\mathbf{ctc}, \mathbf{f}, \mathbf{g}\rrbracket)^{\mathbf{f}} \\
&\mathbf{reify\text{-}contract}\llbracket\mathbf{flat\text{-}ctc}, \mathbf{f}, \mathbf{g}\rrbracket &&= (\texttt{leaf}\ \mathbf{flat\text{-}ctc}^{\mathbf{f}}) \\
&\mathbf{merge}\llbracket(\texttt{cvectorof}\ \mathbf{cctc}_{s1}\ \mathbf{cctc}_{g1})^{\mathbf{f}_1}, &&= (\texttt{cvectorof}\ \mathbf{merge}\llbracket\mathbf{cctc}_{s2}, \mathbf{cctc}_{s1}\rrbracket \\
&\qquad (\texttt{cvectorof}\ \mathbf{cctc}_{s2}\ \mathbf{cctc}_{g2})^{\mathbf{f}_2}\rrbracket && \qquad\qquad \mathbf{merge}\llbracket\mathbf{cctc}_{g1}, \mathbf{cctc}_{g2}\rrbracket)^{\mathbf{f}_2} \\
&\mathbf{merge}\llbracket(\texttt{leaf}), (\texttt{leaf}\ \mathbf{le}_{old}\ ...)\rrbracket &&= (\texttt{leaf}\ \mathbf{le}_{old}\ ...) \\
&\mathbf{merge}\llbracket(\texttt{leaf}\ \mathbf{le}_{new}\ \mathbf{le}_{more}\ ...), (\texttt{leaf}\ \mathbf{le}_{old}\ ...)\rrbracket &&= \mathbf{merge}\llbracket(\texttt{leaf}\ \mathbf{le}_{more}\ ...), (\texttt{leaf}\ \mathbf{le}_{old}\ ...)\rrbracket \\
&\quad \text{where } \exists\ \mathbf{le} \in (\mathbf{le}_{old}\ ...) \text{ s.t. } \mathbf{le} \prec \mathbf{le}_{new} && \\
&\mathbf{merge}\llbracket(\texttt{leaf}\ \mathbf{le}_{new}\ \mathbf{le}_{more}\ ...), (\texttt{leaf}\ \mathbf{le}_{old}\ ...)\rrbracket &&= \mathbf{merge}\llbracket(\texttt{leaf}\ \mathbf{le}_{more}\ ...), (\texttt{leaf}\ \mathbf{le}_{old}\ ...\ \mathbf{le}_{new})\rrbracket \\
&\mathbf{merge}\llbracket\mathbf{cvec}, \mathbf{cctc}\rrbracket &&= \mathbf{merge}\llbracket(\texttt{leaf}\ \mathbf{cvec}), \mathbf{cctc}\rrbracket \\
&\mathbf{merge}\llbracket\mathbf{cctc}, \mathbf{cvec}\rrbracket &&= \mathbf{merge}\llbracket\mathbf{cctc}, (\texttt{leaf}\ \mathbf{cvec})\rrbracket
\end{aligned}$$

$$\frac{}{\mathbf{le} \prec \texttt{any/c}^{\mathbf{f}}}\,[\text{any/c}] \qquad \frac{[\mathbf{r}_{l1}, \mathbf{r}_{h1}] \subseteq [\mathbf{r}_{l2}, \mathbf{r}_{h2}]}{(\texttt{real-in}\ \mathbf{r}_{l1}\ \mathbf{r}_{h1})^{\mathbf{f}_1} \prec (\texttt{real-in}\ \mathbf{r}_{l2}\ \mathbf{r}_{h2})^{\mathbf{f}_2}}\,[\text{real-in}]$$

Fig. 10. Merging, reifying, and relating collapsible contracts

Merging two collapsible vector contracts produces a new collapsible vector contract where the corresponding left and right child contracts (with subscripts "s" and "g" for "set" and "get") have been merged. In order to maintain correct blame assignment, the order of arguments in the recursive call on the left children is swapped, mirroring the reversal of the flow of values.

Merging two leaf nodes must preserve all contracts that can raise blame and drop all redundant contracts. The merging process for leaves assumes that all redundant contracts have been eliminated from each leaf individually, thus **merge** keeps all the contracts from the old leaf node and appends those from the new leaf node at end, when they are not stronger than an existing contract.

The ≺ relation captures a partial ordering on contracts. The any/c contract accepts all values and therefore any contract is stronger than the any/c contract. One real-in contract is stronger than another if it describes a subset of real numbers. For any other pair of contracts, neither is stronger than the other. The ≺ relation is a simplified version of Racket's contract-stronger? as described in section 3.

The final two clauses of the **merge** metafunction handles the situation in which leaf and vector collapsible contracts are merged. This can occur when two contracts such as (vectorof any/c) and (vectorof (vectorof any/c)) are attached to a value. In this situation, the vector contract is converted to a leaf and then the two leaves are merged.

There are multiple ways in which the leaf-merging process could be improved in order to merge more contracts. For example, the model could detect when multiple different range contracts can be consolidated or more simply when two existing range contracts together imply a third one. Our model hews to the implementation in an effort to be a faithful exposition of it. Our implementation does not collapse contracts in this case because it does not come up frequently in practice.

6 COLLAPSIBLE CONTRACTS IN PRACTICE

We have implemented collapsible vector and collapsible function contracts as a new feature for Racket. Including the core ideas of our collapsible contract representation into a mature production-quality contract system presents several challenges. A successful implementation of collapsible contracts must: coexist and interoperate with existing non-collapsible contracts, impose a reasonable cost over existing non-collapsible contracts in cases where collapsing does not happen, and maintain all of the invariants expected by the contract system's internal representations.

At the heart of satisfying the performance aspects of these challenges are two optimizations: *adaptive optimization*, to limit merging (and the costs associated with it) to cases where it is likely to improve performance overall; and *caching*, to memoize the results of merging.

This section begins with some background on the implementation of the Racket contract system, and explains how collapsible contracts fit within its architecture. Then, each of the two key optimizations are described. Finally this section describes the technical issues related to preserving the invariants of Racket's contract system when it is extended to support collapsible contracts.

6.1 The Racket Contract System

The implementation of the Racket contract system has been in active development over the last 15+ years. One aspect of its evolution has been a tremendous amount of tuning and optimization to reduce the cost of contracts. One key design decision towards that goal is to stage contract application and do as much work up front (in the early stages) as possible. This avoids performing expensive computations in the later (and much more frequently executed) stages.

Therefore, the interface for contracts used by the Racket contract system is a function that accepts the blame value for the server component and returns a function that accepts the value to be contracted and the blame value for the client component. Because idiomatic Racket programs primarily attach contracts at module boundaries and clients cannot be known statically, this interface allows the contract system to prepare the data structures for contract checking before the entire blame is known. In terms of performance, this means that the contract system can perform expensive computations as soon as it receives the blame for the server to avoid this computation every time the contract is attached to a value.

The implementation of collapsible contracts takes advantage of this architecture by building the data structures needed for contract merging when the initial blame value is received. This avoids the cost of constructing collapsible data structures within hot loops for the majority of contracts.

6.2 Adaptive Optimization

The model of collapsible contracts in section 5 eagerly performs the merge operation when a contract is attached to a value that had been previously contracted. In practice, attaching and merging a second contract involves converting contracts into the tree representation, merging the trees, and attaching the combined contract to the value. This sequence of operations is (in the short term) more expensive than simply wrapping the value with an additional contract.

Additionally, checking merged contracts requires accessing and traversing the tree data structures, which has a higher fixed cost than simply accessing a contract wrapper. If a value is contracted but never used, or is never contracted again, then the savings from collapsible contracts may not counterbalance these fixed costs. To avoid paying the cost of data-structure management for collapsible contracts when it is unlikely to benefit overall program performance, our implementation uses a strategy inspired by adaptive optimization [Arnold et al. 2000].

Specifically, our implementation initially relies on the Racket contract system's original implementation strategy—i.e., one wrapper per contract, with wrappers piling up when multiple contracts are applied. It then adaptively switches to a merging-based strategy when it determines that merging is likely to improve performance going forward.

The decision to switch is made based on the number of contracts that have been applied to a value; once a value has been wrapped 10 times, our implementation switches strategies. Therefore, from the 11th contract on, all new contracts will be merged, instead of getting their own wrapper.

With this approach, programs that exhibit pathological contract-wrapping can benfit from merging, but avoid the heavyweight merge-supporting tree data structure when there are few wrappers.

6.3 Caching

An additional cost in the implementation of collapsible contracts is the merge operation. Merging two collapsible contracts requires traversing the two data structures, creating the resulting merged data structure, and filtering out unnecessary contracts in the leaf nodes. In pathological cases of contract wrapping it is likely that the same contract or a small set of contracts is repeatedly applied to a value. Indeed, the number of contracts in the program is (typically) bounded by the size of the program, whereas the number of contract applications is proportional to the length of the execution. Without special care, the cost of merging would be paid on every contract application, despite the results being drawn from a limited overall set.

Even though, as section 6.1 describes, our implementation of collapsible contracts relies on the Racket contract system's multi-stage architecture to avoid building tree data structures in hot code, the cost of merging can remain significant. To reduce the performance impact of the merge operation, our implementation takes advantage of the assumption that, in pathological cases, values will be wrapped repeatedly with the same set of contracts, and caches the results of contract merging. With caching, the cost of merging two specific contracts is paid the first time we merge them; further calls can reuse the result. The cache holds onto merged contracts using weak links, relying on the garbage collector to clear entries from the cache.

6.4 Preserving Invariants

Racket implements contract wrappers using *chaperones*, an interposition mechanism for higher-order values [Strickland et al. 2012]. Chaperones preserve an important invariant of Racket's contract system, namely that the result of attaching a contract to a value should be `equal?` (a notion of structural equality) to the uncontracted value. Racket's `equal?` function cooperates with chaperones, and peels away chaperone wrappers before comparisons to preserve that invariant.

On top of this, Racket provides a `chaperone-of?` function, which returns true when its first argument is a chaperone of its second.[2] This procedure imposes an additional invariant on Racket's contract system, namely that a contracted value must be a `chaperone-of?` the corresponding unwrapped value. The `chaperone-of?` function is used by Racket programmers to determine if two objects behave the same way, except that one might signal an error where the other does not.

Explaining the usefulness of chaperones and the `chaperone-of?` operation is beyond the scope of this paper, but the need to preserve the `chaperone-of?` relationship between values poses a significant challenge for the implementation of collapsible contracts. In order to merge and collapse contracts, contract wrappers (chaperones) must be removed, but naively removing chaperones breaks the `chaperone-of?` relationship. Addressing this problem requires introducing a new runtime primitive, *unsafe chaperones*, and various low-level changes to Racket's runtime system.

Overall, implementing collapsible contracts as part of a practical contract system poses a number of thorny technical issues and took months of effort to reach the quality needed to be considered for inclusion in Racket itself. The code is not yet included in Racket's git master, but we keep it separate only to be able to easily conduct experiments (described in the next section). The main Racket contract system maintainer and the main Typed Racket maintainer both support the change.

[2] This is a simplification of `chaperone-of?`. The full details of the procedure are described in the Racket documentation at `http://docs.racket-lang.org/reference/chaperones.html`

7 EVALUATION

To test the hypothesis that (1) collapsible contracts improve some pathological cases of gradual typing performance, and (2) otherwise impose only a small impact on program performance, we compare the performance of Racket (and Typed Racket) extended with collapsible vector and function contracts against Racket without collapsible contracts. We also report on the overall slowdown of the gradual typing benchmarks after our improvements.

We use Takikawa et al. [2016]'s gradual typing benchmarks plus some additional benchmarks that the benchmark suite maintainers have since added. There are 20 benchmark programs, named: Acquire, Dungeon, Forth, Fsm, Fsmoo, Gregor, Kcfa, Lnm, Mbta, Morsecode, QuadBG, QuadMB, Sieve, Snake, Suffixtree, Synth, Take5, Tetris, Zombie, and Zordoz.

Each benchmark consists of a number of different modules. Each module exists in the benchmark suite as both an untyped module and as a typed one (the only difference being the addition of type annotations). Accordingly, each benchmark that has n modules has 2^n configurations, each one a different, gradually-typed program. The set of benchmarks covers a wide ground in terms of the kinds of contracts used, the amount of contract wrapping that occurs, and the relative performance of a specific configuration compared to the untyped configuration. For more details on the benchmark programs, please see `https://docs.racket-lang.org/gtp-benchmarks/`.

7.1 Experiment

For the main evaluation, we measured two versions of Racket: a fork of Racket v6.12 with a number of contract-related performance improvements, and an extension of that fork that adds support for collapsible contracts. The artifact includes the source for both of these versions of Racket.

For benchmarks with at most 12 modules, we measure the running time of all of the typed/untyped configurations on both versions of Racket. For the three benchmarks with more than 12 modules, we follow Greenman and Migeed [2018]'s lead and randomly select a subset of configurations to measure.[3] The results for the benchmark Gregor (13 modules) include 1,195 sampled configurations, the results for the benchmark QuadBG (14 modules) include 1,332 sampled configurations, and the results for the benchmark QuadMB (14 modules) include 1,344 sampled configurations. In total, there are 6,851 configurations. As context, running the full set of configurations just once takes 52 hours on our benchmark machine.

For each typed/untyped configuration, we measure performance with three steps. First, we ensure the code is compiled to bytecode. (Racket programs can be run without this step but it may confound our measurements.) Second, we run the configuration once and ignore the execution time. Finally, to collect the execution time, we run the configuration as follows. Each run starts in a fresh instance of the Racket VM. For every benchmark, we collected 18 execution times in total via one round of 2 runs and two rounds of 8 runs.

All measurements were collected on a Linux machine with two physical AMD Opteron 6376 processors (a NUMA architecture) and 128GB RAM. The CPU cores on each processor ran at 2.30 GHz using the "performance" CPU governor. The machine is used only for benchmarking and no benchmarks were run in parallel.

7.2 Speedup Results

Because there are many different programs in the benchmarks, we first summarize the results by looking at the best- and worst-case configuration for each benchmark. To start, consider the upper half of figure 11. Each pair of bars corresponds to a single typed/untyped configuration from each

[3] The sampling protocol is designed to produce figure 14. Uses of the sample data in other figures are marked with asterisks (*) because we do not claim they are statistically significant.

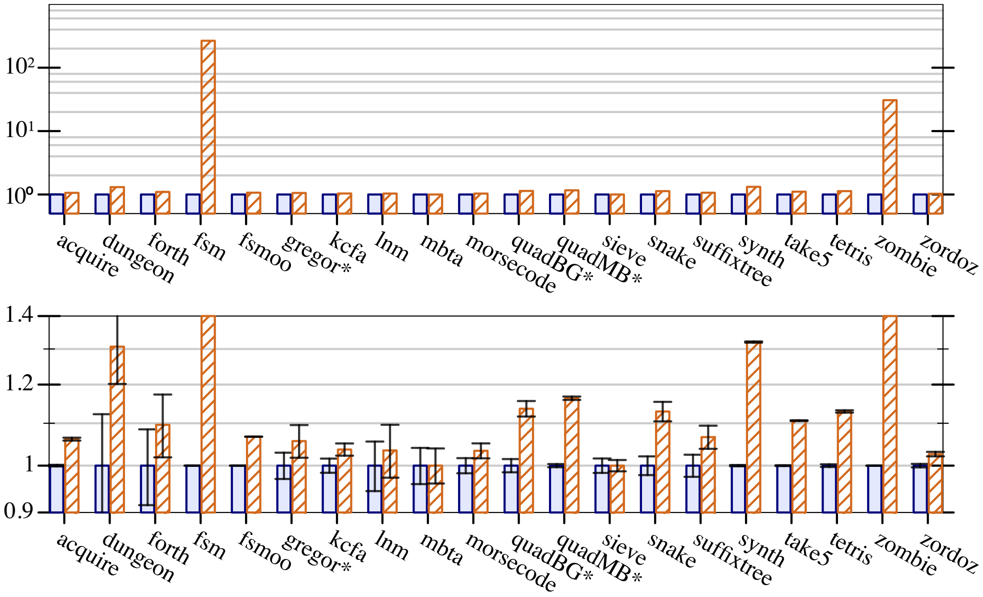


Fig. 11. Best-case improvement in gradual typing overhead for collapsible contracts over all (* = sampled) configurations; higher bars are better. Solid blue (■) bars show the gradual typing overhead for one configuration on Racket v6.12 and striped orange (□) show the gradual typing overhead for the same configuration with collapsible contracts; both bars are normalized to the Racket v6.12 overhead. The lower plot is a zoomed-in version of the upper plot.

benchmark. The configuration chosen is the one with the largest improvement in gradual typing overhead for collapsible contracts, relative to Racket v6.12. The height of the striped orange bar (on the right) is the ratio of gradual typing overheads between Racket with collapsible contracts and Racket v6.12. The height of the solid, blue bar on the left shows the ratio of gradual typing overhead in Racket v6.12 to itself; it is included only to show the error bars on the measurement (which are below the resolution of this plot). The lower half of figure 11 contains the same information, but is zoomed in on the region near 1, making the error bars visible.

Figure 11 shows that we achieve significant improvements of the gradual typing overhead for two of the benchmarks, Fsm and Zombie, which are 275x and 35x faster, respectively. The figure also shows that many benchmarks have at least one configuration for which collapsible contracts reduce the overhead of gradual typing and that collapsible contracts do not introduce additional gradual typing overhead for all configurations of each benchmark.

7.3 Slowdown Results

Of course, it is important to ensure that our handling of pathological contract wrapping does not damage the ordinary case, so we turn to that question next. Consider figure 12. Like figure 11, it has a pair of bars for each benchmark. Unlike figure 11, however, figure 12 picks the configuration with the smallest improvement in gradual typing overhead (equivalently, the largest regression) for collapsible contracts, relative to Racket v6.12. The bars are similar to those in figure 11. On 12 of the 20 benchmarks, the error bars for the slowdown overlap with those for the Racket v6.12 measurements. The other benchmarks show minor slowdowns.

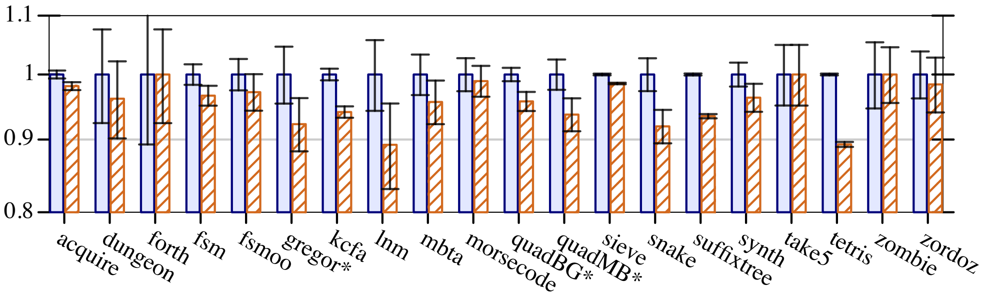


Fig. 12. Worst-case improvements of gradual typing overhead for collapsible contracts compared against Racket v6.12 (lower bars are worse overhead); the colors match figure 11.

To understand the change in gradual typing overhead introduced by collapsible contracts in more detail, consider figure 13. It shows the distribution of the gradual typing overhead across all of the 6,837 configurations that take at least 100 milliseconds (under both the unmodified Racket 6.12 and the version with collapsible contracts). The figure consists of two histograms: the left one counts configurations that run slower (i.e., with a higher gradual typing overhead) with collapsible contracts, and the one on the right counts configurations that improve with collapsible contracts.

As the histograms show, most of the configurations (45%) exhibit less than 2% slowdown or speedup, indicating no significant change in the overhead of gradual typing using collapsible contracts compared to the overhead in Racket v6.12. A small percentage (10%) are slowed down by at least 2%; the remaining 45% are sped up by at least 2%. Only 3 configurations (less than 1%) are slowed down by more than 10%. The worst slowdown is 14%.

7.4 Overall Cost of Gradual Typing

The overall cost of gradual typing remains high for most benchmarks. Figure 14 shows plots in the style of Takikawa et al. [2016]'s gradual typing evaluation for all of the benchmarks we consider. These plots pack a lot of information in a small space, and therefore deserve a careful explanation.

Consider Acquire, the first plot in figure 14. The line that traces the edge of the blue region tells us what percentage of the configurations (on the y-axis) of Acquire have at most a given slowdown (on the x-axis) on Racket v6.12. Vertical ticks appear at 1x, 1.2x, 1.4x, 1.6x, 1.8x, 2x, 4x, and continue with multiples of 2 up to 20x. (For benchmarks where the worst-case slowdown exceeds 20x, the x-axis stretches to fit.) So, for example, a little more than 50% of the configurations run at most 4x slower than the untyped configuration. This can be seen by tracing the horizontal line at 50% over to where it intersects the the curve. More generally, plots with a large blue area (i.e., those where the blue line climbs to 100% the fastest) perform well in the baseline version of Racket. That is, a large percent of their configurations run with little overhead relative to the untyped configuration.

The orange line shows the performance of Racket with collapsible contracts. In Acquire, there is no significant difference and so the orange curve follows the blue curve. In Fsm, however, there is a significant difference, showing that collapsible contracts improve that benchmark. With collapsible contracts, all configurations are better than 4x slower and without them, about half of the configurations are more than 580x slower.

For Gregor, QuadBG, and QuadMB, the blue and orange lines are actually 95% confidence intervals generated from the slowdowns in each sample [Greenman and Migeed 2018].

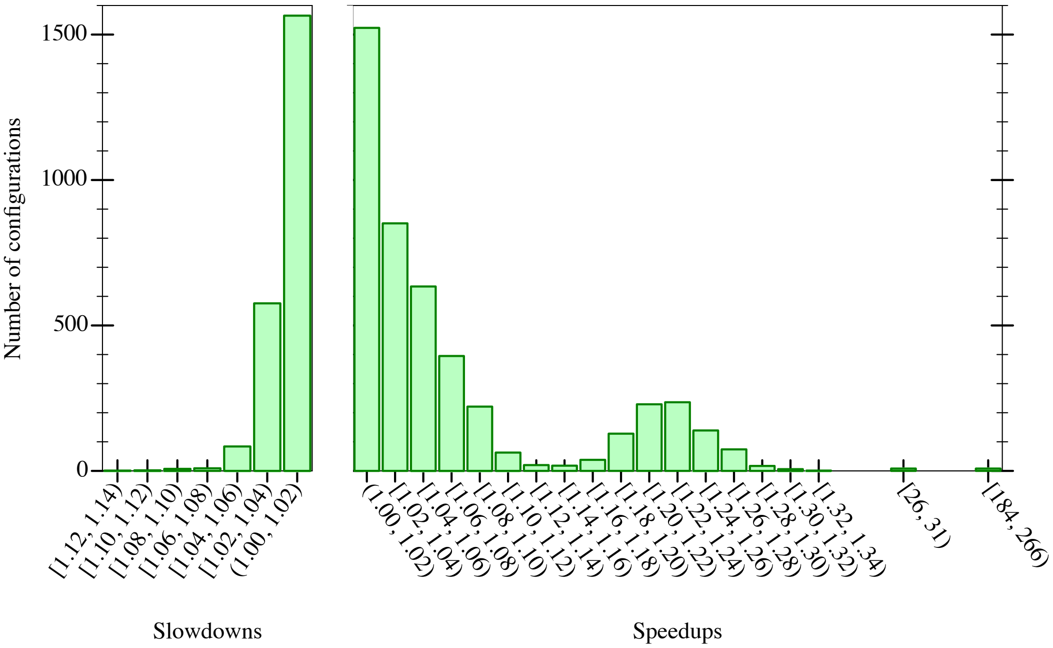


Fig. 13. Histogram of gradual typing overhead across all typed/untyped configurations.

To dig into the performance of typed/untyped benchmarks in more detail, consider figure 15. It shows scatter plots inspired by those in Bauman et al. [2017]'s performance comparison between Racket and Pycket. In our plots, every point corresponds to one configuration of the benchmark shown in the plot. A point located at (x, y) indicates that the configuration runs x times slower than untyped with collapsible contracts and y times slower than untyped in Racket v6.12. Configurations that fall on the line $x = y$ (drawn in grey across the plot) represent configurations for which collapsible contracts do not affect program performance positively or negatively. Points that lie above the line indicate configurations that are improved by collapsible contracts. Points that lie below the line are configurations for which collapsible contracts introduce additional overhead. Lastly, configurations whose x and y coordinates are less than 1 run faster than untyped; these configurations lie within the grey square at the lower-left of each plot.

Most of the plots are like Acquire, where the points all lie close to the line $x = y$. There are a few exceptions. The Fsm and Zombie plots demonstrate configurations where collapsible contracts provide significant speedups, showing points far above the line. The Synth benchmark (and to a lesser extent the Take5 benchmark) shows modest speedups that appear to be proportional to the running time of the benchmark. And finally Tetris shows modest slowdowns.

The Dungeon, Forth, and Fsmoo benchmarks suffer from extreme overhead (>100x) in both versions. Upon further inspection, the overhead is due to redundant wrappers on first-class classes and first-class objects. These overheads suggest a need for collapsible class and object contracts.

8 RELATED WORK

Early efforts to combine static and dynamic typing date back to MACLISP [Moon 1974] and Common Lisp [Steele 1990], which offer syntax for type annotations but let implementations choose how to interpret them. Henglein [1994] and Rehof [1995]s' seminal work put the problem of combining

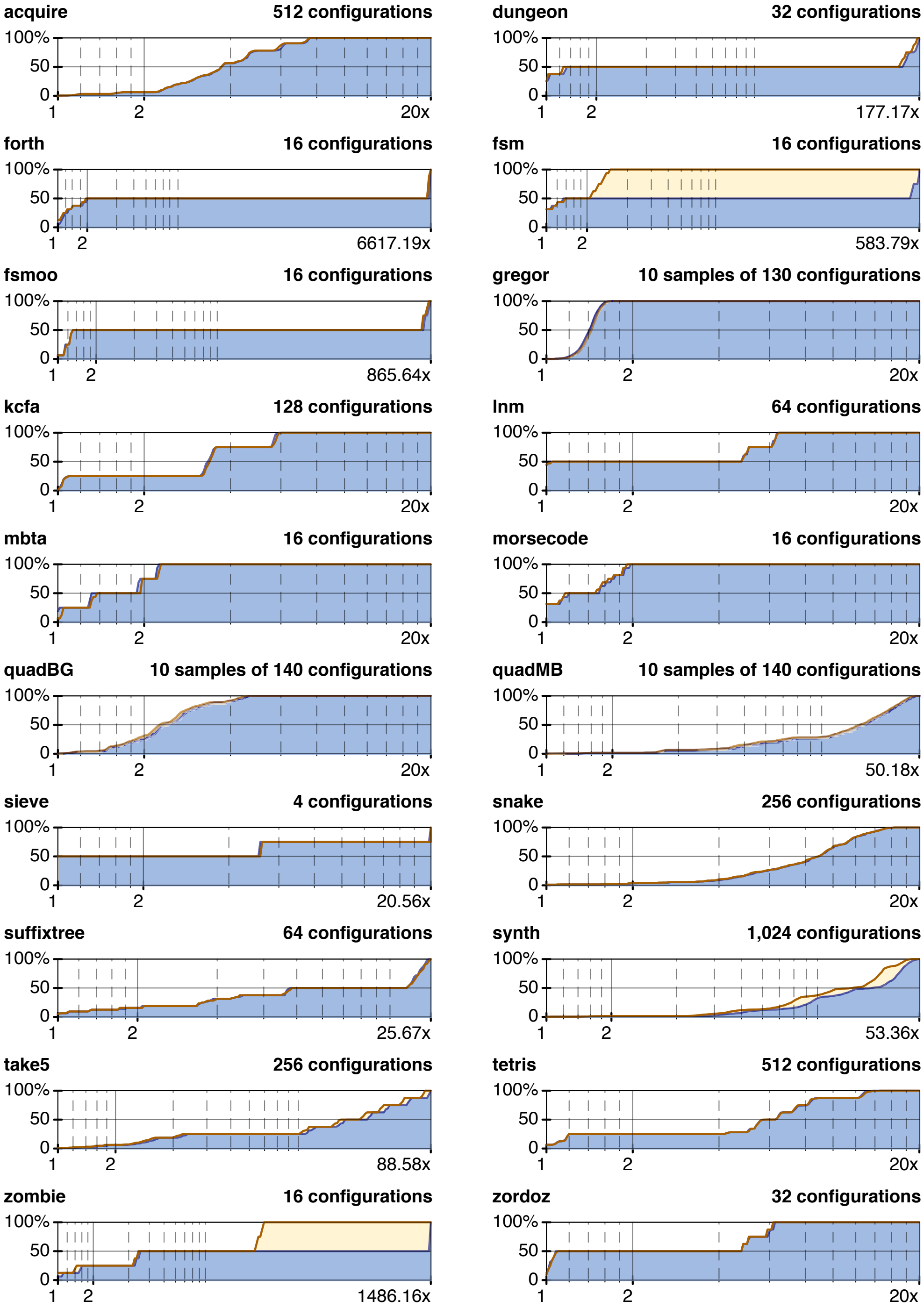


Fig. 14. Overhead of gradual typing in the benchmark programs. The blue (🟦) curve is for Racket 6.12 and the orange (🟨) curve is for collapsible contracts. A point (x,y%) on the line means y% of the configurations incur a slowdown of at most x.

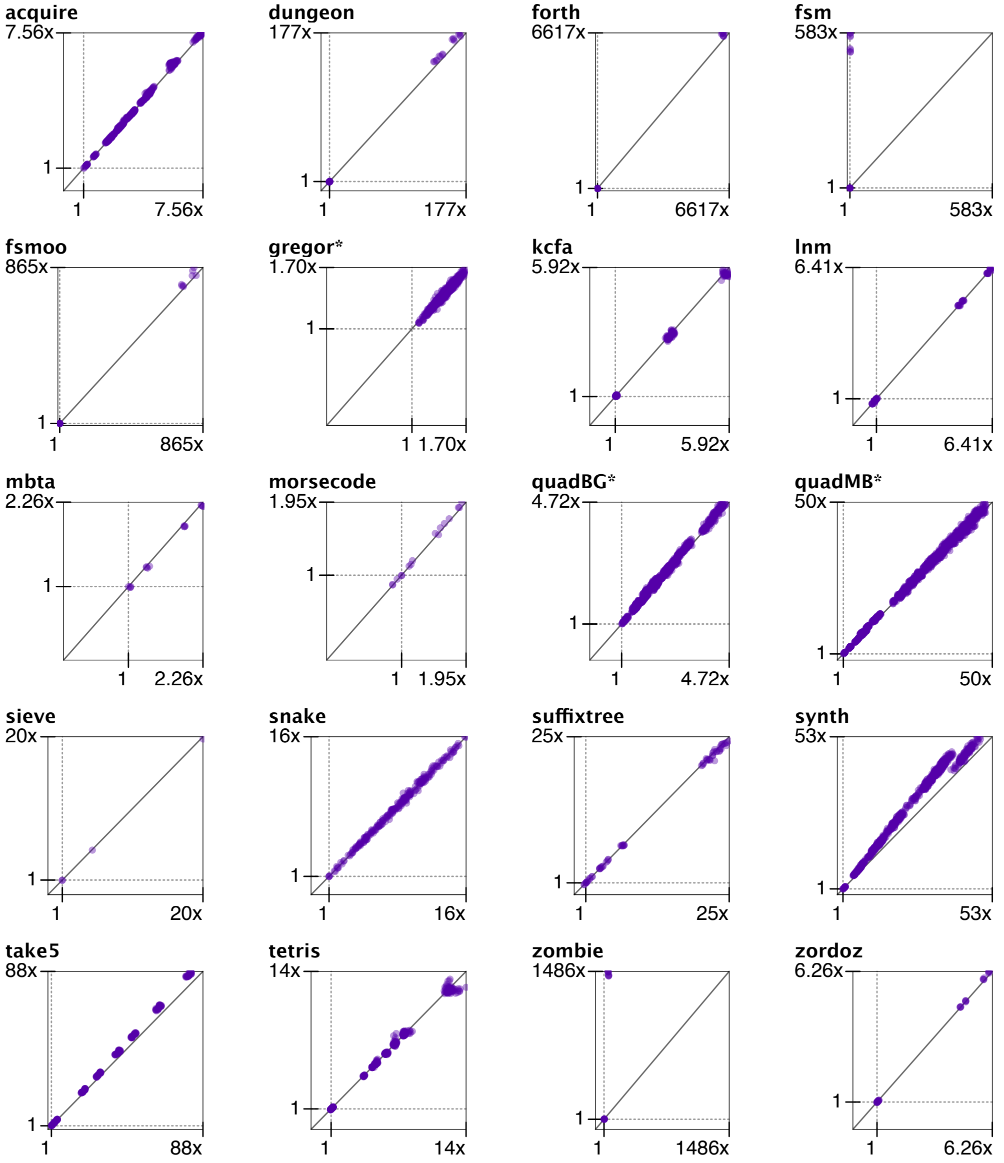


Fig. 15. Scatterplots showing head-to-head performance of collapsible and non-collapsible contracts; points above the line are better for collapsible contracts, below are worse. Points in the box in the bottom left corner, i.e., those with both x and y coordinates less than 1, represent partially-typed configurations that run faster than the untyped configuration.

typed and untyped languages on a theoretically valid footing. More recently, the community has adopted an approach dubbed “gradual typing” [Gronski et al. 2006; Matthews and Findler 2009; Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006]. Attempting to make that work practical led to our understanding of significant performance problems [Takikawa et al. 2016].

Collapsible contracts are one of three recent efforts to improve the performance of higher-order gradual typing. Our collapsible effort adapts Greenberg's theory of *eidetic* contracts [Greenberg 2015, 2016] to Racket, and thereby provides an improved back-end to Typed Racket. In addition to eidetic contracts, Greenberg [2015] proposes two other theories, dubbed *heedful* and *forgetful*, that forgo correct blame but collapse contracts more aggressively. To our knowledge, these alternative semantics have not been implemented.

A second, earlier effort is the Pycket tracing JIT compiler [Bauman et al. 2017]. Pycket is a compiler and runtime system for Typed Racket that is implemented in PyPy [Bolz et al. 2009]. Through a combination of PyPy's tracing JIT and a new representation of contracts as hidden classes, Pycket is able to eliminate the overhead of many Typed Racket programs from Takikawa et al. [2016]'s benchmark suite. Collapsible contracts are orthogonal to Pycket; combining the two should lead to synergistic improvements.

The third recent implementation effort is the Grift compiler for (an extension of) the gradually-typed lambda calculus [Kuhlenschmidt et al. 2018]. Grift implements a theory of compositional coercions [Herman et al. 2010; Siek et al. 2015]; in other words, Grift has a runtime representation of type casts and this representation supports a merge operation analogous to collapsible contracts. Early results suggest that coercions avoid the same pathology as collapsible contracts and add relatively little overhead [Kuhlenschmidt et al. 2018]. That said, a head-to-head comparison of Racket and Grift is difficult for two reasons. First, the semantics of the underlying programming language are subtly different in ways that can have dramatic effects on performance. We discovered three examples by experimenting with the implementation: Grift supports only small integers whereas Racket supports a rich numeric tower; Grift does not support union types whereas Typed Racket accommodates arbitrary unions; and Grift does not yet implement tail calls. Second, Grift's language is too small to be able to run any of our benchmark programs, so it is difficult to understand the performance at the application level (as opposed to the micro-benchmark level).

Whether higher-order gradual typing can be truly practical is still an open question, and so researchers are currently exploring alternatives. One alternative is to use types only for static analysis, and ignore the question of runtime overhead [Bierman et al. 2014; Bracha and Griswold 1993; Chaudhuri et al. 2017]. Another is to design a new language with the flexibility of dynamic typing, but where every value comes with an intrinsic type [Muehlboeck and Tate 2017; Rastogi et al. 2015; Richards et al. 2015; Wrigstad et al. 2010].[4] Finally, a third approach is to offer weaker soundness and blame guarantees [Roberts et al. 2018; Vitousek et al. 2017]. This work is instead an attempt at making higher-order gradual typing efficient without relaxing any of its guarantees or limiting expressiveness.

9 CONCLUSION

This paper introduces collapsible contracts, a novel runtime representation for contract checking. Collapsible contracts address a particular pathological performance problem that occurs in gradually typed programs due to excessive contract wrapping. By providing a way to merge contracts and eliminate unnecessary contract checking, collapsible contracts limit the impact of contract checking in sound gradual type systems. Although collapsible contracts still come with a cost, their performance on a benchmark suite of gradually typed programs indicate that performance can be greatly improved for certain pathological cases. In most other cases, collapsible contracts do not impact program performance, and in a small number of cases they impose a slight overhead.

[4] Dart 2 currently follows the intrinsic-type approach; Dart 1 erases types.

Collapsible contracts do not solve all of the performance problems of sound gradual typing. They do, however, provide a technique that implementations of gradual typing systems and contract systems can take advantage of to reduce the impact of enforcing type invariants at runtime.

ACKNOWLEDGMENTS

Thanks to Christos Dimoulas for comments on the writeup and helping us distill the `bubble/c` example, Matthew Flatt for help with Racket's runtime system, the anonymous reviewers for their feedback, and the NSF for financial support of this work.

REFERENCES

Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for Smalltalk. *Science of Computer Programming* 96(1), pp. 52–69, 2013.

Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, pp. 47–65, 2000.

Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. Sound Gradual Typing: Only Mostly Dead. *Proceedings of the ACM on Programming Languages* 1(OOPSLA), pp. 54:1–54:24, 2017.

Gavin Bierman, Martin Abadi, and Mads Torgersen. Understanding TypeScript. In *Proc. European Conf. Object-Oriented Programming*, pp. 257–281, 2014.

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proc. Wksp. on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pp. 18–25, 2009.

Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, pp. 215–230, 1993.

Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levy. Fast and Precise Type Checking for JavaScript. *Proceedings of the ACM on Programming Languages* 1(OOPSLA), pp. 48:1–48:30, 2017.

Benjamin W. Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. KafKa: Gradual Typing for Objects. In *Proc. European Conf. Object-Oriented Programming*, pp. 12:1–12:23, 2018.

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete Monitors for Behavioral Contracts. In *Proc. European Sym. on Programming*, pp. 214–233, 2012.

Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 48–59, 2002.

Matthew Flatt and PLT. Reference: Racket. PLT Design Inc., PLT-TR-2010-1, 2010. https://racket-lang.org/tr1/

Michael Greenberg. Space-Efficient Manifest Contracts. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 181–194, 2015.

Michael Greenberg. Space-Efficient Latent Contracts. In *Proc. Sym. Trends in Functional Programming*, 2016.

Ben Greenman and Matthias Felleisen. A Spectrum of Type Soundness and Performance. *Proceedings of the ACM on Programming Languages* 2(ICFP), pp. 71:1–71:32, 2018.

Ben Greenman and Zeina Migeed. On the Cost of Type-Tag Soundness. In *Proc. Wksp. on Partial Evaluation and Program Manipulation*, pp. 30–39, 2018.

Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid Checking for Flexible Specifications. In *Proc. Wksp. Scheme and Functional Programming*, pp. 93–104, 2006.

Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming* 22(3), pp. 197–230, 1994.

David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient Gradual Typing. *Higher-Order and Symbolic Computation* 23(2), pp. 167–189, 2010.

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. Efficient Gradual Typing. arXiv, 1802.06375v1, 2018.

Jacob Matthews and Robert Bruce Findler. Operational Semantics for Multi-Language Programs. *Trans. Programming Languages and Systems* 31(3), pp. 12:1–12:44, 2009.

David A. Moon. MACLISP Reference Manual. MIT, Revision 0, 1974.

Fabian Muehlboeck and Ross Tate. Sound Gradual Typing is Nominally Alive and Well. *Proceedings of the ACM on Programming Languages* 1(OOPSLA), pp. 56:1–56:30, 2017.

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 167–180, 2015.

Jakob Rehof. Polymorphic Dynamic Typing: Aspects of proof theory and inference. DIKU, 95/9, 1995.

Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete Types for TypeScript. In *Proc. European Conf. Object-Oriented Programming*, 2015.

Richard Roberts, Michael Homer, Stefan Marr, and James Noble. Shallow Types for Insightful Programs: Grace is Optional, Performance is Not. arXiv:1807.00661v1, 2018.

Jeremy Siek, Peter Thiemann, and Philip Wadler. Blame and coercion: Together again for the first time. In *Proc. ACM Conf. Programming Language Design and Implementation*, pp. 425–435, 2015.

Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Proc. Wksp. Scheme and Functional Programming*, 2006.

Guy L. Steele. *Common Lisp the Language.* 2nd edition. Digital Press, 1990.

T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and Impersonators: Run-time Support for Reasonable Interposition. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, pp. 943–962, 2012.

Asumu Takikawa, Daniel Feltey, Earl Dean, Robert Bruce Findler, Matthew Flatt, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards Practical Gradual Typing. In *Proc. European Conf. Object-Oriented Programming*, 2015.

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max New, Jan Vitek, and Matthias Felleisen. Is Sound Gradual Typing Dead? In *Proc. ACM Sym. Principles of Programming Languages*, pp. 456–468, 2016.

Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage Migration: from Scripts to Programs. In *Proc. Dynamic Languages Symposium*, pp. 964–974, 2006.

Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory Typing: Ten Years Later. In *Proc. Summit on Advances in Programming Languages*, pp. 17:1–17:17, 2017.

Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big Types in Little Runtime: Open-World Soundness and Collaborative Blame for Gradual Type Systems. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 762–774, 2017.

Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. Mixed Messages: Measuring Conformance and Non-Interference in TypeScript. In *Proc. European Conf. Object-Oriented Programming*, pp. 28:1–28:29, 2017.

Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating Typed and Untyped Code in a Scripting Language. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 377–388, 2010.