Does Blame Shifting Work?

LUKAS LAZAREK, Northwestern University, USA
ALEXIS KING, Northwestern University, USA
SAMANVITHA SUNDAR, Northwestern University, USA
ROBERT BRUCE FINDLER, Northwestern University, USA
CHRISTOS DIMOULAS, Northwestern University, USA

Contract systems, especially of the higher-order flavor, go hand in hand with blame. The pragmatic purpose of blame is to narrow down the code that a programmer needs to examine to locate the bug when the contract system discovers a contract violation. Or so the literature on higher-order contracts claims.

In reality, however, there is neither empirical nor theoretical evidence that connects blame with the location of bugs. The reputation of blame as a tool for weeding out bugs rests on anecdotes about how programmers use contracts to shift blame and their attention from one part of a program to another until they discover the source of the problem.

This paper aims to fill the apparent gap and shed light to the relation between blame and bugs. To that end, we introduce an empirical methodology for investigating whether, for a given contract system, it is possible to translate blame information to the location of bugs in a systematic manner. Our methodology is inspired by how programmers attempt to increase the precision of the contracts of a blamed component in order to shift blame to another component, which becomes the next candidate for containing the bug. In particular, we construct a framework that enables us to ask for a contract system whether (i) the process of blame shifting causes blame to eventually settle to the component that contains the bug; and (ii) every shift moves blame "closer" to the faulty component.

Our methodology offers a rigorous means for evaluating the pragmatics of contract systems, and we employ it to analyze Racket's contract system. Along the way, we uncover subtle points about the pragmatic meaning of contracts and blame in Racket: (i) the expressiveness of Racket's off-the-shelf contract language is not sufficient to narrow down the blamed portion of the code to the faulty component in all cases; and (ii) contracts that trigger state changes (even unexpectedly, perhaps in the runtime system's data structures or caches) interfere with program evaluation in subtle ways and thus blame shifting can lead programmers on a detour when searching for a bug. These points highlight how evaluations such as ours suggest fixes to language design.

CCS Concepts: • **Theory of computation → Program specifications**; • **Software and its engineering → Empirical software validation**.

Additional Key Words and Phrases: higher-order contracts, blame, programming languages design evaluation

ACM Reference Format:
Lukas Lazarek, Alexis King, Samanvitha Sundar, Robert Bruce Findler, and Christos Dimoulas. 2020. Does Blame Shifting Work?. *Proc. ACM Program. Lang.* 4, POPL, Article 65 (January 2020), 29 pages. https://doi.org/10.1145/3371133

Authors' addresses: Lukas Lazarek, Northwestern University, USA, lukas.lazarek@eecs.northwestern.edu; Alexis King, Northwestern University, USA, lexi.lambda@gmail.com; Samanvitha Sundar, Northwestern University, USA, samanvithasundar2020@u.northwestern.edu; Robert Bruce Findler, Northwestern University, USA, robby@cs.northwestern.edu; Christos Dimoulas, Northwestern University, USA, chrdimo@northwestern.edu.

2475-1421/2020/1-ART65
https://doi.org/10.1145/3371133

1 THE ORIGIN STORY OF BLAME

The origins of contracts and blame in higher-order languages [Findler and Felleisen 2002] can be traced to an apocryphal story.[1] Once upon a time, a young PhD Student embarked on the mission of building a programming environment for a newly-hatched higher-order language. The road to success was (and still is) strewn with vicious bugs, and the Student fought for days, months and years to weed out as many of them as possible. Some times though, the battle was impossible to win... The Student had to deal with havoc-causing faulty callbacks and other powerful values from other people's code. All the Student could do was labor hard to trace where the values came from, and try even harder to convince the authors' of that other code that the problem was on their end and their responsibility. After repeating this process again and again, the Student finally made a wish; "I wish there was a way to say what values others should give to my code, and if they do not comply then they get blamed!". And so contracts and blame came to be. Happily ever after, contracts caught all the stray values, blame showed where they came from, and the Student had to worry no more about what piece of code was at fault.

Twenty years on, stories like this sustain the folklore belief in the potency of blame for helping programmers find software bugs. Papers about higher-order contracts contain claims in the vein of "blame kicks off the debugging process in the right direction" or "blame narrows down the search for the bug", despite a lack of systematic supporting evidence (e.g. [Dimoulas et al. 2013, 2016; Strickland and Felleisen 2009b; Waye et al. 2017]).

This paper examines whether the reputation of blame is justified. Its **contribution** is the first rigorous empirical methodology for phrasing and answering two questions about contract systems and how their blame assignment relates to the location of bugs. The first question asks whether programmers that heed blame and focus their attention to blamed components eventually locate the faulty component. We refer to *components* as core units of a program's structure (depending on language context, examples might include modules, classes, or definitions). The second question asks whether programmers keep getting "closer" to uncovering the location of the bug when they rely on blame for guidance. Together, these questions aim to determine whether programmers can translate blame information to the location of bugs in a systematic manner. In other words, the paper introduces a means for evaluating the pragmatics of a contract system. After all, the pragmatic value of blame assignment, and thus a key element of a contract system, is determined by its relevance for eliminating bugs.

The main source of inspiration for our methodology is the established programming practice for dealing with blame.[2] If a programmer is convinced that a blamed component does not contain a bug, then the programmer increases the precision of the contracts between the component and other components in an attempt to detect faulty values the component received. That is, the programmer attempts to *shift the blame* to some other part of the program. Using blame shifting, our two questions reduce to whether (i) iterative blame shifting culminates in blame settling on the faulty component, and (ii) blame shifting always moves blame "closer" to the bug.

To study these questions, inspired by recent work for evaluating gradual type systems [Takikawa et al. 2016], we start with a program with a number of components out of which one is known to be faulty, and we construct a lattice with elements that correspond to different configurations of the program. Each configuration describes a different choice of precision for the contracts of the program and, therefore, the lattice enables us to systematically explore the relationship between contract precision and blame. Specifically, we can use the lattice to answer our first question;

[1] This story is a work of fiction. Names, characters, business, events and incidents are the products of the authors' imagination. Any resemblance to actual persons, living or dead, or actual events is purely coincidental.
[2] See for example https://beautifulracket.com/jsonic-2/contracts.html.

whether repeated increase of the precision of the contracts of a program results in blame shifting which eventually causes blame to point to the faulty component of the program.

Similarly, we use the configuration lattice to answer our second question; whether increasing the precision of contracts shifts blame "closer" to the faulty component. To define closer, we draw inspiration from the theory of correct blame [Dimoulas et al. 2011]. While this theoretical work says nothing about blame with respect to bugs, it gives meaning to blame through the flow of the witness of the contract violation, i.e., the value that failed a contract check. In particular, a contract system that blames correctly assigns it to a component that controlled the flow of the witness at some point during the evaluation of a program. This fact jives with the phenomenon that programmers observe when blame points to a component that is not inherently faulty; the component gets blamed because it received a faulty value and either tried to use it or passed it along to another component. For our purposes, the path of the flow of a faulty value from component to component connects the faulty part of a program to the blamed component and, hence, the length of this path gives a natural notion of distance between blame and the location of the bug for each configuration in the lattice.

We have applied our methodology to evaluate Racket's contract system on a corpus of programs and we have discovered that, contrary to the folklore, neither question has a positive answer for all of these programs. First, while blame shifting does identify the faulty component in the majority of cases, there are specific situations where Racket's contract language does not support writing the contracts that would blame the faulty component. Second, while in most cases blame shifting moves blame closer to the bug, Racket's contracts can affect the evaluation of a program in surprising ways that, after blame shifting, lead to an increase rather than a decrease of the distance between the blamed and the faulty component. These issues highlight how investigations such as ours can improve language design.

Section 2 describes the main ideas and insights of the paper with a concrete example. Section 3 turns the insights into testable hypotheses and develops in abstract terms our method for examining whether they hold for a contract system. The instantiation of the method for evaluating Racket's contract system comes in section 4 while section 5 discusses the results of the evaluation. We have collected lessons we learn about contracts and blame in Racket and in general in section 6. Section 7 discusses related work and section 8 gathers some final thoughts about the pragmatic meaning of contracts and blame.

2 A MOTIVATIONAL EXAMPLE AND THE KEY INSIGHTS

Figure 1 depicts a snippet of a Racket program that calculates an infinite stream of prime numbers using the Sieve of Eratosthenes. We use this example to demonstrate how programmers react to a contract violation and how they can use contracts and blame to facilitate debugging. We consider the top level definitions of the program to be its components. The snippet consists of three such components:

- `sift` is a function that consumes a number `n` and a stream of numbers `st`, and returns a stream that contains the same numbers as `st` except those that are multiples of `n`;
- `sieve` consumes a stream `st`, and constructs a new stream with the same head (`hd`) as `st`, and the recursively `sieve`d tail of `st` after `sift`ing from it `hd`;
- `primes` is the stream that `sieve` returns when given the stream of all naturals starting at `2`.

Two of these definitions come with contracts:[3]

- the contract for `sift`, `(-> integer? stream? stream?)`, states that it is a function that consumes an integer and a stream and produces a stream; while

[3]In Racket, programmers can opt to accompany some definitions with a contract using the `define/contract` form.

```
#lang racket
_ _ _ _ _ _ a few lines of code plus dependencies _ _ _ _ _ _ _ _
;; `sift n st` Filter all elements in `st` that are divisble by `n`.
;; Return a new stream.
(define/contract (sift n st)
  (-> integer? stream? stream?)
  (define-values (hd tl) (stream-unfold st))
    (cond [ (= 1 (modulo hd n)) (sift n tl)]
          [else (make-stream hd (λ () (sift n tl)))]))

;; `sieve st` Sieve of Eratosthenes
(define (sieve st)
  (define-values (hd tl) (stream-unfold st))
  (make-stream hd (λ () (sieve (sift hd tl)))))

;; stream of prime numbers
(define/contract primes
  (streamof (and/c integer? prime?))
  (sieve (count-from 2)))
_ _ _ _ _ _ _ _ _ _ _ more lines of code _ _ _ _ _ _ _ _ _ _ _ _
```

Fig. 1. The Sieve of Eratosthenes, with a bug highlighted in red.

- the contract for `primes`, `(streamof (and/c integer? prime?))`, states that it is a stream of integers that are also prime numbers.

These contracts are sufficient to uncover a bug we planted in the implementation of the Eratosthenes sieve. In detail, when we run the program and attempt to inspect the first two elements of `primes`, the contract system complains that the stream's second element is 4, an integer that is definitely not prime. Thus it fails the `prime?` part of the contract of `primes`. Together with the information about which value failed what contract, the contract system provides *blame* information that identifies the definition responsible for the problem. In this case, blame points to `primes`, which promised to be a stream of primes.

However, even a cursory inspection of `primes` indicates that the problem is not actually there. As the highlighted condition in figure 1 shows, the problem is with `sift`. In contrast to what it is supposed to do, `sift` fails to remove from its `st` argument elements that are multiples of its `n` argument. Unfortunately the contract of `sift` is not precise enough to detect this discrepancy, and `sieve` does not have a contract at all. This reflects a fundamental aspect of the design of contract systems; programmers can choose the level of precision of the contracts of their components and the contract system reports only a mismatch between the contracts and the program's behavior. Hence, in the absence of precise contracts, blame points to the component whose contracts detect that it handles a faulty value. However, this value may have reached the blamed component from somewhere else in the code under contracts that are insufficient (if there are any at all) to detect the bug contracts. Specifically in our example, `primes` ends up getting blamed because it has blindly trusted these two components to produce values about which `primes` makes promises in its own contract [Dimoulas and Felleisen 2011; Dimoulas et al. 2011].

The above justification of blame is the source of the **key insight** of this work: if we make the contract of `primes` more precise, then the contract system should be able to detect the problem and give us blame information that is more accurate with respect to the location of the bug, that is the contract system should detect that `primes` received a faulty value. In general terms, heeding

```
;;;;;;;;;;;;;;;;;;;;;;; (a) contracts for sieve ;;;;;;;;;;;;;;;;;;;;;;;;
;; (a.1) a tag-checking contract for sieve
(-> stream? stream?)

;; (a.2) a type-like contract for sieve
(-> (streamof integer?) (streamof integer?))

;; (a.3) a very precise contract for sieve
(-> (streamof integer?) sieved-stream/c)

;;;;;;;;;;;;;;;;;;;;;;; (b) contracts for sift ;;;;;;;;;;;;;;;;;;;;;;;;
;; (b.1) a type-level contract for sift
(-> integer? (streamof integer?) (streamof integer?))

;; (b.2) a very precise contract for sift
(->i ([n integer?]
      [st (streamof integer?)])
      [result (n)
              (streamof (and/c integer?
                               (not/c (divisible-by/c n))))])
```

Fig. 2. A precision progression of contracts for (a) `sieve` and (b) `sift`.

blame and increasing the precision of the contracts in a program should eventually lead to the identification of the component that contains the bug. It is worth noting, however, that it is not always possible for a programmer to write a sufficiently precise contract to detect a problem without re-structruring their program. That said, for this study, we elect to consider how we can increase the precision of contracts while leaving the program proper intact. In other words, we examine the relation between blame and bugs within the margins of the expressive power of a contract system and a fixed set of programs.

Back to our example: even though `primes` seems to be as precise as possible, in fact, it is missing something important; `primes` interacts and receives values from `sieve`. Thus increasing the precision of the contract of `primes` requires making the contract of `sieve` more precise, at least for the use of `sieve` in `primes`.[4]

Figure 2 shows three candidate contracts for `sieve` ordered by increasing precision.[5] The first, (a.1), states properties of the tags of the argument and result of `sieve`. Of course, this is insufficient to change the behavior of the program; `sieve` does indeed produce a stream when given a stream. Thus, attempting to inspect the first two elements of `primes` with this new contract results in exactly the same contract error that blames `primes`. The second contract, (a 2), is also insufficient; the result stream does contain integers, just not the right ones.

Further increasing the precision of the contract requires considering the expected behavioral properties of `sieve` beyond "type-level" descriptions. In particular, `sieve` should produce a stream where no integer in the stream is divisible by any of its predecessors. To check this property, the

[4]This is a subtle point of the design of Racket's contract system. Even though we refer to the contract of a component as a single entity that regulates all its interactions with any other component in a program, Racket's contract system pushes programmers to split the contract into multiple contracts spread across a number of components. For the purpose of this study, we treat all of these pieces as the single contract of a component.

[5]In Racket, contracts are ordinary values that programmers can construct with the help of a small domain specific language of contract combinators such as the function contract combinator `->` and predicates such as `prime?`.

last contract for `sieve` in figure 2, (a.3), replaces the range of the previous contract, (a.2), with the custom contract `sieved-stream/c`. The contract verifies that the stream's tail contains only numbers indivisible by its head; then it attaches itself recursively to the tail of that stream, thereby building up the property that no element of the stream is a factor of any later element.

Given this precise contract that captures the functional correctness of `sieve`, inspecting the first few elements of `primes` leads to a new contract error that blames `sieve`. Blame does not yet detect the faulty `sift` but at least it now draws the attention of the programmer to a point earlier in the path of the faulty value from `sift` to `primes`; it singles out the intermediary `sieve`. In this way, blame shifts closer to the location of the bug.

Since an inspection of `sieve` confirms that the bug is not there, the next step is to revisit the contracts of `sift`, the component `sieve` receives values from. The bottom part of figure 2 shows how we can enhance gradually the precision of the contract of `sift` to obtain a contract that, similar to the last one for `sieve`, describes precisely the expected behavior of the function. This *dependent* contract, (b.2), uses the function contract combinator `->i` instead of `->`. The former supports naming the arguments and result values of a function to use them to construct other portions of the contract. In this case, the contract for the result of `sift` depends on the argument `n`, and uses it to enforce that the elements of the result are not divisible by `n`.[6] Hence it is now sufficiently precise to detect the bug and blame finally shifts to `sift`, the definition that contains the bug.

The above exposition offers a view of two promising properties about blame assignment:

- **Blame Trail:** If blame points to a component, either:
  - The component contains the bug, or
  - If a programmer increases the precision of the contracts between the blamed component and those from which it receives values, then blame shifts to another component;
- **Search Progress:** When blame shifts from one component to another due to increasing the precision of contracts, blame moves *closer* to the bug in the program.

The pragmatic consequence of these properties is that repeating the blame shifting process makes blame eventually settle on the faulty component while at each point programmers make progress towards uncovering the bug. Thus, determining whether or not these properties hold for a contract system helps us evaluate the design of the contract system.

3 FROM THE IDEAS TO AN EXPERIMENTAL DESIGN

This section develops our method for examining the truthfulness of the Blame Trail and Search Progress properties for a contract system. The description offers a blueprint to any language designer that would like to use the method to evaluate the design of the contract system of their language. Inspired by Takikawa et al. [2016], the starting point of the method is a set of programs and a lattice of program configurations that we can explore systematically for each one of the programs.

3.1 The Configuration Lattice

There are two requirements for selecting suitable programs for our experiment: (i) each program should contain a number of different components; and (ii) there should be exactly one fault in the program.

Since the central goal of the method is to explain how, in a given contract system, blame assignment changes when modifying the contracts of a program, we assign a list of contracts to

[6] Indeed, the contract for the result of `sift` is identical to the non-recursive portion of `sieved-stream/c` from the last contract for `sieve` except that the latter uses the head of the result of `sieve` instead of `n`.

each component of the selected programs. The list of contracts will be used to automatically select contracts at a given precision level for each component. To that end, the list is ordered in terms of increasing precision; that is, for any two contracts, one (the more precise) must supersede the other. A contract k supersedes a contract k' iff a program using k instead of k' signals a contract violation when the program using k' does so. Intuitively, k should check everything that k' checks, and possibly more. For example, the least precise contract of a contract list may describe a trivial correctness property that every component satisfies;[7] the most precise contract may describe the partial functional correctness of the component and a contract of intermediate precision may describe a type-like property of the component.

Defining the precision comparison in this manner may catch the careful reader by surprise. Since, as we discuss in section 2, the goal of our methodology is to examine blame shifting, the straight-forward approach is to compare the precision of the contracts based on the precision of the contracts for the inputs of a component independently of the precision of the output contracts. In contrast, our comparison takes into account the precision of contracts for inputs and outputs alike. However, from the perspective of blame shifting this does not matter; if a component gets blamed then its output contracts already detect the issue and increasing their precision further is inconsequential. Thus, we opt for a stronger precision comparison that in turn simplifies our experimental setup without loss of generality.

We leave the details about how contract system designers that want to run our experiment should select programs and contracts intentionally abstract beyond their crucial properties. Both the programs and the contracts are parameters of the experimental design.

With programs and contracts in hand, we can create a number of variants of each program, dubbed *configurations*. Configurations are complete programs obtained by selecting a specific contract for every component from its list. Hence, configurations differ from each other in the contracts of their components. For example, in one configuration we may select the least precise contract from the contract list of a component and in another configuration the most precise. We can run a configuration to determine if the contract system detects any bug, and if so which component it blames.

The following definitions summarize the important bits about configurations:

- A *contract map* $KMAP$ relates each component c of a program with a list of contracts ordered by precision $\{K_1 K_2 ... K_n\}$.
- A *configuration* $CONF$ maps each component of a program to an element of $KMAP(c)$; that is, it maps each component to a specific contract in the component's ordered contract list.

Naturally, we can relate configurations to each other based on the precision of their contracts. Given a software system C and a contract map $KMAP$, we say a configuration $CONF$ is a *ascendant* of a configuration $CONF'$ iff there exists a component c such that $KMAP(c) = \{k_1, ..., k_n\}$, $CONF(c) = k_{j+1}$ and $CONF'(c) = k_j$, and for all other c', $CONF(c') = CONF'(c')$. That is, given a configuration, we can obtain its ascendant by replacing the contract of one component c with the immediately more precise contract in c's ordered contract list. We call c the *distinction point* between the configuration and its ascendant. Generalizing the ascendant relation, a configuration $CONF$ is *above* a configuration $CONF'$ iff $CONF = CONF'$, or there exist configurations $CONF_0, \ldots, CONF_n$ such that $CONF_{i+1}$ is a ascendant of $CONF_i$ for $0 \leq i < n$ and $CONF = CONF_0$ and $CONF' = CONF_n$. In other words, $CONF$ is above $CONF'$ iff there exists a chain of ascendants from $CONF$ to $CONF'$.

Consequently, the configurations of a program C with contract map $KMAP$ form a complete lattice $\mathcal{L}[\![C, KMAP]\!]$ of size $\prod_{i=1}^{|C|} |KMAP(c_i)|$. The top element of the lattice is the configuration where all components have their most precise contracts, and the bottom element is the configuration where

[7] In Racket, this corresponds to the `any/c` contract.

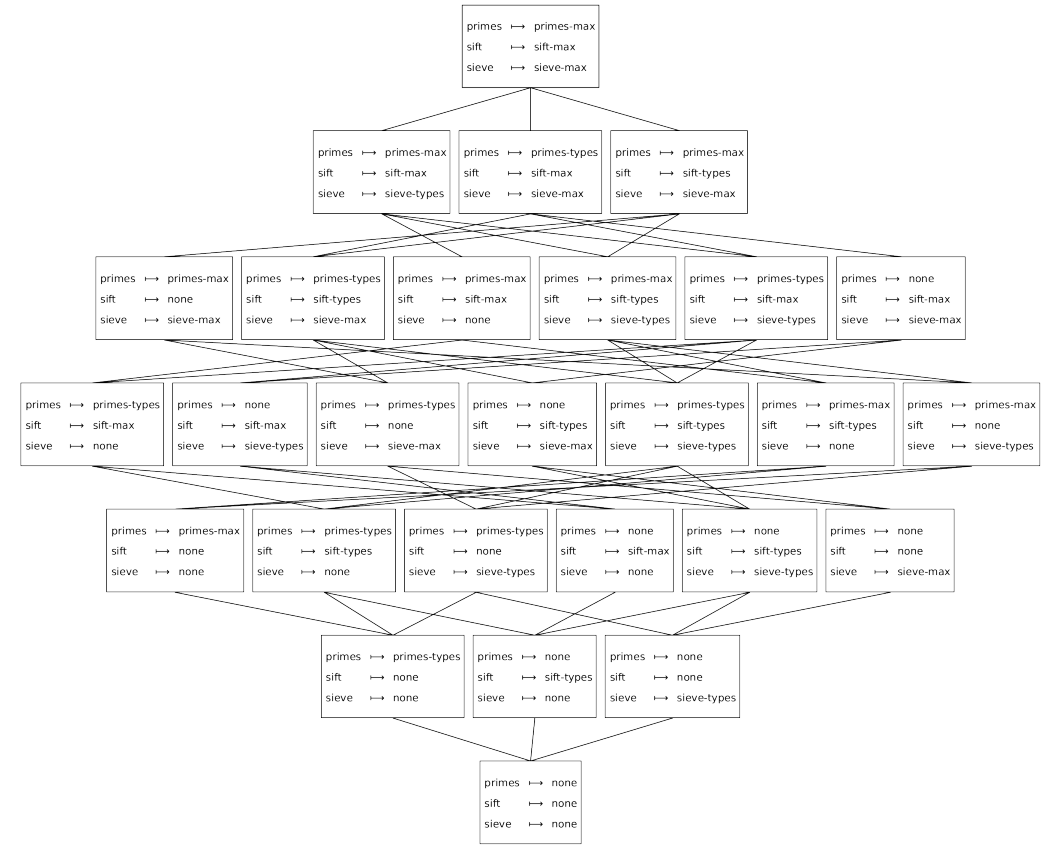


Fig. 3. The lattice for the example from section 2.

all components have their least precise contracts. Figure 3 illustrates the lattice for the example from section 2, if the only components were `sieve`, `sift`, and `primes`.

3.2 The Experimental Procedure

We use the configuration lattice to formalize the relationship between blame and bugs, and to examine whether Blame Trail and Search Progress from section 2 hold for a contract system. First, we introduce a few necessary definitions. We say:

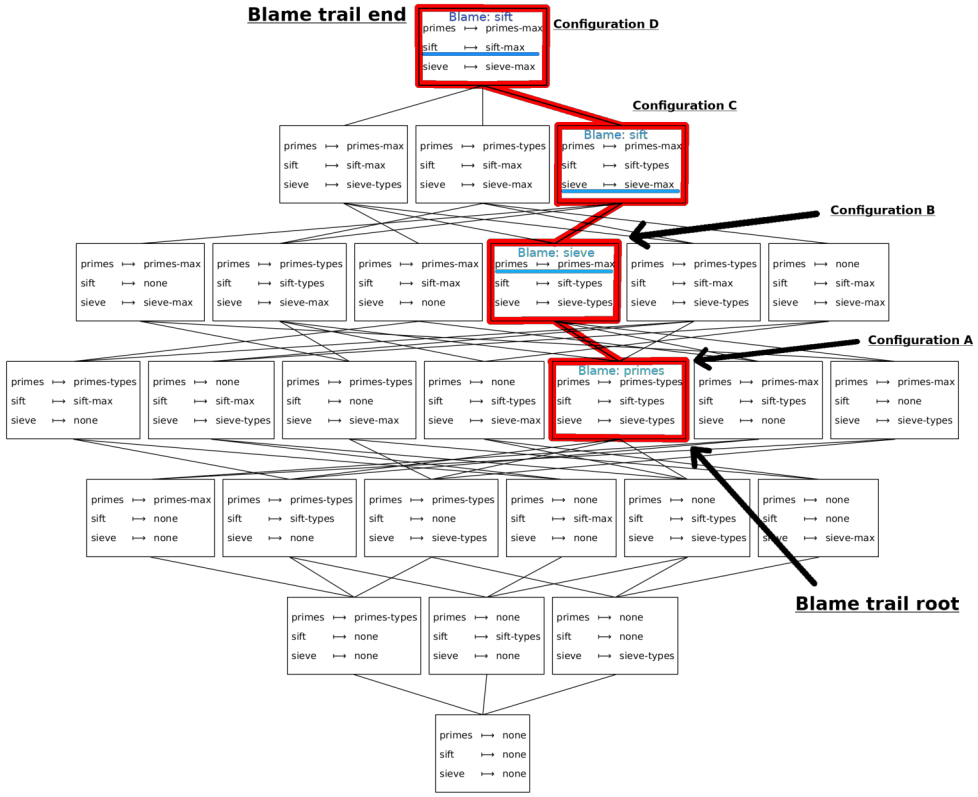


Fig. 4. A blame trail for the example from section 2.

- a configuration $CONF_r$ is *significant* in $\mathcal{L}[\![C, KMAP]\!]$ iff evaluating $CONF_r$ results in a contract violation.
- a *blame trail* in $\mathcal{L}[\![C, KMAP]\!]$ is a sequence of significant configurations $CONF_0, \ldots, CONF_n$ such that $CONF_{i+1}$ is a ascendant of $CONF_i$ and evaluating $CONF_i$ blames the distinction point between $CONF_i$ and $CONF_{i+1}$.
- the configuration $CONF_0$ in a blame trail $CONF_0, ..., CONF_n$ is the *root* of the blame trail.
- the *configuration comparison* $\leq^X_{\mathcal{L}}$ is a function that consumes two configurations in a blame trail and returns whether the first produces blame closer to the faulty component than the second. This function is parameterized over a particular lattice ($\mathcal{L}[\![C, KMAP]\!]$) and faulty component ($X$).

Figure 4 depicts the lattice for the running example from section 2 along with a blame trail. The root configuration of the trail evaluates to blame that points to `primes`. The next configuration along the trail differs from the root in the precision of the contract of `primes`, so it is a ascendant of the root with distinction point `primes`.

Each of the configurations in the lattice may exhibit blame that is closer or further from the bug than the blame of other configurations. For example, the root configuration points to `primes` as the culprit for the contract violation while the bug is in `sift`. In contrast, another configuration blames `sift`, which is the component containing the bug. That is, in terms of the configurations in

figure 4, we have $C \leq^X_{\mathcal{L}} A$. In section 2, we informally describe a distance comparison based on the path that the witness of a contract violation follows from component to component. In section 4 we revisit this comparison in detail. Here, however, we use the configuration comparison to abstract a relative notion of how far the blame of a configuration is from a bug compared to another. In particular, the configuration comparison is not defined precisely beyond its interface with the rest of the experiment.

Hence, together with the selection of the programs for the experiment and the contract map of each program, the configuration comparison is one of the three parameters of the experiment that capture factors that may vary depending on the intentions of the contract system designer that runs the experiment. For instance, contract system designers can choose different sets of programs to study how (a part of) their contract system interacts with a set of language features. Similarly, one designer may want to understand how the contract system behaves for a spectrum of "reasonable" contracts, that is, contracts that the designer deems programmers usually write. In contrast, another designer may be interested in the full spectrum of expressiveness of a contract system. Our experimental design can accommodate both, as they can select their contract map accordingly. Finally, designers can experiment with different configuration comparisons to model the different systematic ways blame shifting makes progress towards the detection of a faulty component, each offering a different perspective on blame in a contract system. We revisit the practical implications of the selection of these parameters in section 4, where we discuss the instantiation of the experimental design for Racket.

Figure 4 suggests how we can use blame trails, contract maps, and the configuration comparison to restate the Blame Trail and Search Progress properties from section 2. The process of shifting blame from component to component by strengthening the precision of the contract around the blamed component forms a blame trail, and that trail should end with blame on the buggy component. Furthermore, each step of the process should bring blame closer to the bug according to $\leq^X_{\mathcal{L}}$. Specifically, given a program C with a faulty component c, contract map $KMAP$, and configuration comparison $\leq^X_{\mathcal{L}}$, we define the properties more precisely as follows:

- **Blame trail:** For every significant configuration $CONF_r$ in $\mathcal{L}[\![C, KMAP]\!]$, all trails that start at $CONF_r$ and cannot be further extended terminate at a configuration $CONF_t$ that blames the faulty component c. In other words, every blame trail in $\mathcal{L}[\![C, KMAP]\!]$ ends with blame on the faulty component.
- **Search progress:** For every blame trail $CONF_0, \ldots, CONF_n$ in $\mathcal{L}[\![C, KMAP]\!]$,

$$CONF_{i+1} \leq^X_{\mathcal{L}} CONF_i$$

In other words, we turn the two properties into testable hypotheses that we can verify or disprove with the configuration lattice. The rest of the experiment is divided into two phases, one per hypothesis, that we repeat for all programs we have selected after constructing their configuration lattice. Figure 5 extends figure 4 with annotations for how elements of the trail contribute to checking each of the hypotheses.

Phase I: Testing Blame Trail. Given a lattice, the first step is to separate its significant configurations from those that do not produce blame. For example, a configuration may not produce blame because certain bugs may cause the configuration to produce a runtime error before any contract violation. Alternatively, some bugs may produce behavior which the configuration's contracts are unable to distinguish from correct behavior because they are not precise enough. To distinguish significant from non-significant configurations we simply run all configurations and check whether they result in blame. Of course, it is possible that no configuration in the lattice is significant and thus the whole lattice is irrelevant. We can quickly exclude such scenarios by running the top configuration of the lattice. After all, if the contracts of that configuration are not precise enough

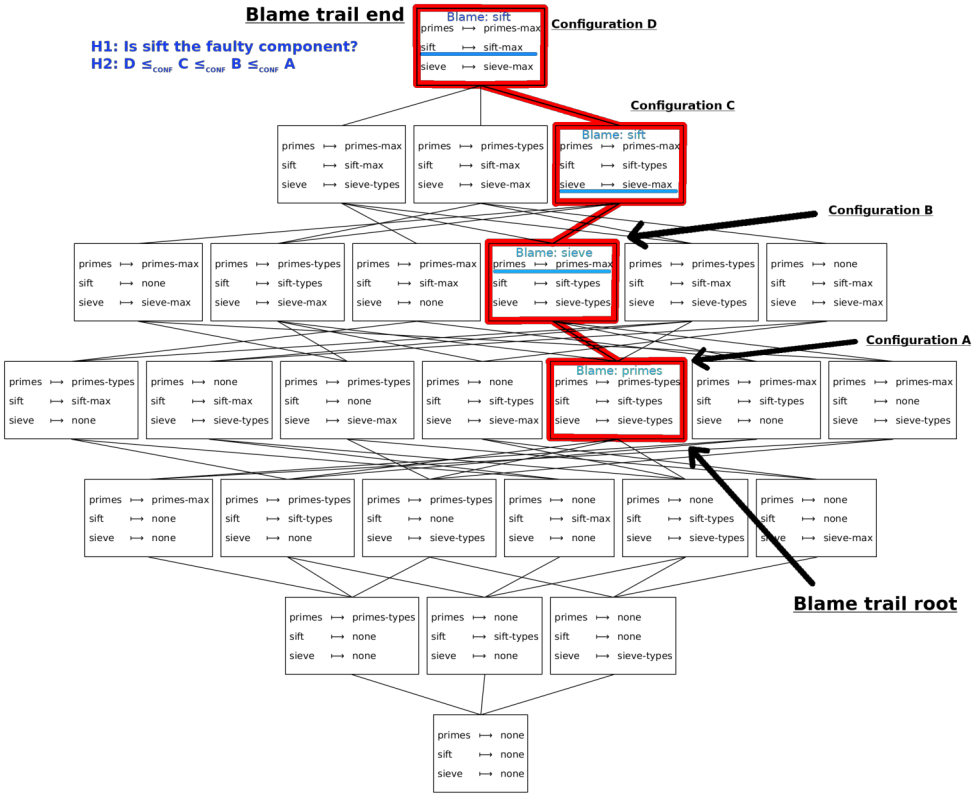


Fig. 5. Checking a blame trail for the two hypotheses.

to detect the bug, then no other configuration can be because the top configuration contracts supercede all others.

After isolating the significant configurations, we use them as starting points to examine whether all blame trails in the lattice reach configurations that blame the faulty component. To that end, we treat each significant configuration as the root of a blame trail in the lattice. In particular, we check if the configuration has a ascendant with distinction point equal to the configuration's blamed component. If such a ascendant exists, it becomes the next configuration along the blame trail that we examine. We therefore say that we follow the blame trail by repeating the process from the ascendant configuration. If no matching ascendant exists, we check whether the configuration blames the faulty component of our program. If it does, then the property holds for this blame trail and we continue inspecting the lattice until all blame trails have been considered. In the opposite case, we have discovered a violation of Blame Trail.

Before concluding the discussion of phase I, we must consider how blame assignment interacts with a fundamental pragmatic property of contracts. Contracts are typically written in the same language as program's components, which allows programmers to use familiar reasoning, tools, and abstractions in both the specification and implementation of a program. Naturally, however, that aspect makes contracts prone to the same kinds of bugs as the implementation of components. In fact, contracts occasionally share code with components, hence a bug in a component might

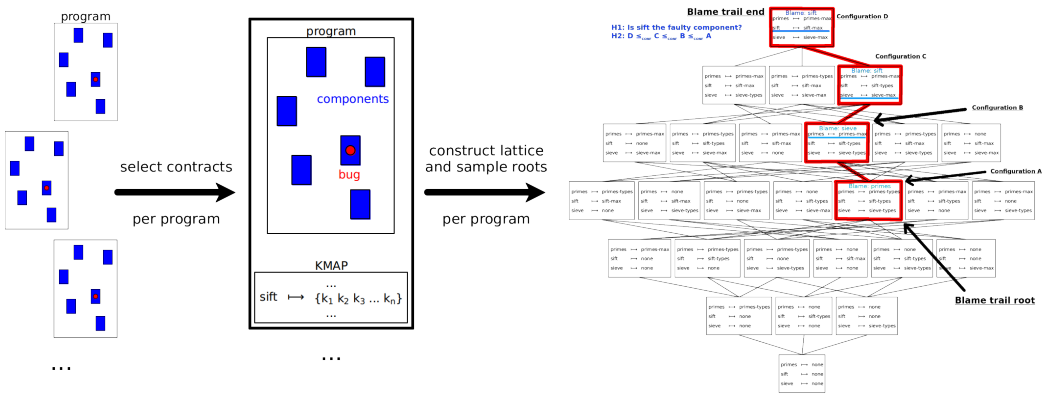


Fig. 6. Overview of experimental procedure.

manifest simultaneously as a bug in a contract, which in turn may affect the properties that contracts check. For example, if a contract uses a faulty version of a predicate then it may detect a violation where it should not and blame an innocent bystander. On the other hand, it may fail to detect a violation when it should, making bogus any result the program produces, including blame.

To account for this in our experimental design, any references inside contract code to program components are always protected by the most precise contracts from the contract map for those components. The intuition behind this precaution is that if a contract uses a faulty component, then the component's most precise contract is the best we can do to detect the issue before it affects contract checking. This reflects that blame, similar to the contract checks that produce it, is only meaningful if contracts are not faulty. In practical terms, it means that programmers must exercise extra caution in the implementation of contracts in order for blame information to be meaningful.

Phase II: Testing Search Progress. We test Search Progress by a *post mortem* analysis of data collected during the first phase. For each blame trail $CONF_0, \ldots, CONF_n$, we check whether $CONF_{i+1} \leq^X_{\mathcal{L}} CONF_i$ for all $0 \leq i < n$. If for a pair of configurations $CONF_1$ and $CONF_2$ that appear in a blame trail, $CONF_2$ is above $CONF_1$ but $CONF_2 \leq^X_{\mathcal{L}} CONF_1$ returns false, then we have discovered a violation of Search Progress. Computing these comparisons may require instrumenting the configuration evaluations during the first phase to record sufficient information about every pair of a configuration and its ascendant visited. Different comparisons will require different information and hence instrumentation, but all likely require at least the faulty component and blamed component of each configuration.

As a final remark on the experimental design, the description so far prescribes that the experiment should examine the full configuration lattice for a program. However, doing so may not be feasible if the program has a large number of components. In this case, we can sample the lattice for significant configurations and focus only on the blame roots starting from these configurations, with the number of samples depending on the desired confidence in the results. Figure 6 depicts a summary of the method using the sampling option.

4 EVALUATING RACKET'S CONTRACT SYSTEM

In order to evaluate the utility of the proposed experimental design, we apply it to Racket's contract system. Herein, we explain how we instantiate each of the parameters that we discuss in section 3.

Table 1. Benchmarks summary.

Program	Description (author)	Features exercised
Dungeon	An imperative program that generates floor maps for an RPG game. (Vincent St-Amour)	lists, structs, first-class classes, objects, vectors, mutable hashes
Forth	An interpreter for the Forth programming language using the object-oriented command pattern. (Ben Greenman)	lists, classes, objects, lists, ho functions
KCFA	A functional implementation of a control flow analysis for the lambda calculus. (Matt Might)	structs, lists, hashes, sets
MBTA	An imperative, object-oriented knowledge base that answers queries about the Boston transit system. (Matthias Felleisen)	lists, classes, objects, hashes
Morsecode	An imperative implementation of the Levenshtein distance algorithm plus some Morse coding. (Neil Van Dyke and John Clements)	lists, vectors, hashes, ho functions
Sieve	Defines a simple stream data type and uses it to implement the Sieve of Eratosthenes. (Ben Greenman)	structs, lists
Snake	A functional implementation of the classic Snake game using basic recursive list processing. (David Van Horn)	structs, lists

4.1 Selection of Programs

We selected seven programs from the gradual typing performance benchmarks of Takikawa et al. [2016] based on the diversity of their features. Some of those programs are highly imperative, some are functional, and others follow an object-oriented design. Furthermore, the programs combine a wide range of Racket constructs such as first class functions, classes, objects, and mutable data. These programs therefore exercise a correspondingly diverse set of Racket's contract system features. Each benchmark comes with an included driver that runs the program on pre-determined inputs, which do a good job of covering the programs; according to Racket's coverage tool, the inputs achieve at least 90% coverage. We summarize the benchmarks in figure 1.

In accordance with the first program selection requirement from section 3, each of these programs consists of a number of components. Specifically, we treat each top level definition per module as a component. This results in a large number of components with involved dependencies and interactions.

The seven programs, though, fail to meet the second program selection requirement; running them does not raise any errors. To remedy this, inspired by mutation testing DeMillo et al. [1988, 1978]; Lipton [1971], we modify the program to produce a synthetic bug[8] in each program in the form of a small syntactic modification, called a *mutation*. For example, figure 7 depicts the original syntax of the Sieve benchmark at the top, and a mutation at the bottom that flips the highlighted conditional expression. In fact, the red highlighted bug in the code from section 2 is the product of

[8] Actually, mutation may lead to an equivalent program, but such cases are already handled by the relevance check in Phase I of the experimental procedure. Section 6.2 discusses further the relationship between mutations and bugs.

```
(define (sift n st)
  (define-values (hd tl)
    (stream-unfold st))
  (cond [(= 0 (modulo hd n))
         (sift n tl)]
        [else
         (make-stream
          hd
          (λ () (sift n tl)))]))
```

```
(define (sift n st)
  (define-values (hd tl)
    (stream-unfold st))
  (cond [ (not (= 0 (modulo hd n)))
         (sift n tl)]
        [else
         (make-stream
          hd
          (λ () (sift n tl)))]))
```

Fig. 7. A mutation of `sift` in the Sieve benchmark.

Table 3. Summary of mutation operators.

operator	description	examples
constants	Replace constants with similar values	`0` ↔ `1`, `True` ↔ `False`
arithmetic	Swap arithmetic operators	`+` ↔ `-`, `*` ↔ `/`
relational	Swap relational operators	`<` ↔ `<=`, `=` ↔ `!=`
logical	Swap logical operators	`and` ↔ `or`
conditional	Negate conditional expressions	`if A` ↔ `if !A`
statement	Delete statements in sequences	`A; B; C` → `B; C`
argument	Swap argument ordering	`f(A, B)` ↔ `f(B, A)`
hide-method	Hide public methods	`public A()` → `private A()`

another mutation that changes the constant `0` to `1`. As these examples illustrate, we can mutate any given component in many ways. We systematically generate a large number of variants for each program, dubbed mutants, using one of the syntactic transformations summarized in figure 3, which are drawn from the standard set of mutation operators Coles [[n. d.]]; DeMillo et al. [1988]. We treat the set of mutants of the original programs as the programs analyzed by our experiment and figure 2 collects the number of mutants for each program along with its number of components and lines of code.

Table 2. Program overview.

program	LOC	components	mutants
Dungeon	541	59	829
Forth	257	27	131
KCFA	230	31	63
MBTA	469	39	177
Morsecode	159	31	105
Sieve	35	10	17
Snake	159	30	89

4.2 The Contract Maps

We have manually implemented contracts for each of the definitions in the selected programs. The contracts come in three levels of precision. The first level, `none`, is the trivial correctness property that holds for any component; we have implemented it using Racket's `any/c`. The second level, `type`, captures type-like properties. For this level, since the programs originate from Typed Racket's performance evaluation benchmark suite, we have translated the Typed Racket types of their definitions into contracts. The third level, `max`, aims for partial functional correctness; it consists of the most precise specification we can express for each definition using Racket's contract combinators and predicates. This level does not aim to be the maximum precision contract that can possibly be implemented, nor do we require them to be so. For instance, we did not implement any contracts that use state to monitor extra-functional properties like how many times a function is invoked. Instead, our selection of contract levels

```
#lang flow-trace
_ _ _ _ _ _ a few lines of code plus dependencies _ _ _ _ _ _ _ _
;; `sift n st` Filter all elements in `st` that are equal to `n`.
;; Return a new stream.
(define/component (sift n st)
  (contract-map
    [max (-> (streamof integer?) sieved-stream/c)]
    [type (-> (streamof integer?) (streamof integer?))])
  (define-values (hd tl) (stream-unfold st))
    (cond [(= 0 (modulo hd n)) (sift n tl)]
          [else (make-stream hd (λ () (sift n tl)))]))

;; `sieve st` Sieve of Eratosthenes
(define/component (sieve st)
  (contract-map
    [max (->i ([n integer?]
               [st (streamof integer?)])
          [result (n)
                  (streamof (and/c integer?
                                   (not/c (divisible-by/c n))))])]
    [type (-> integer? (streamof integer?)
              (streamof integer?))])
  (define-values (hd tl) (stream-unfold st))
  (make-stream hd (λ () (sieve (sift hd tl)))))

;; stream of prime numbers
(define/component primes
  (contract-map
    [max (streamof (and/c integer? prime?))]
    [type (streamof integer?)])
  (sieve (count-from 2)))
_ _ _ _ _ _ _ _ _ _ _ _ _ more lines of code _ _ _ _ _ _ _ _ _ _ _ _ _
```

Fig. 8. Sieve and its Contracts

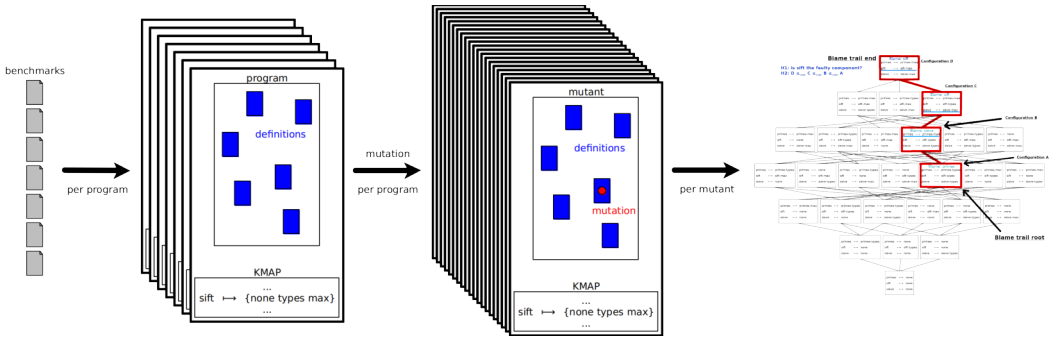
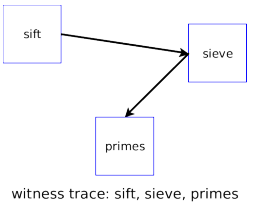


Fig. 9. Racket evaluation experimental procedure.

reflects our effort to understand blame in Racket when programmers take full advantage of its domain specific contract language to express functional specifications.

```
(define/component (sift n st)
  (contract-map
    [max (-> (streamof integer?)
             sieved-stream/c)]
    [type (-> (streamof integer?)
              (streamof integer?))])
  (define-values (hd tl)
    (stream-unfold st))
  (cond [ (not (= 0 (modulo hd n))) (sift n tl)]
        [else
         (make-stream
          hd
          (λ () (sift n tl)))]))
```




```
(define/component primes
  (contract-map
    [max (streamof (and/c integer? prime?))]
    [type (streamof integer?)])
  (sieve (count-from 4 )))
```

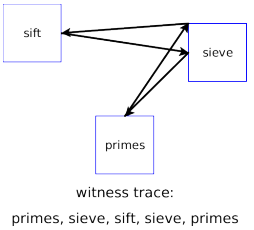


Fig. 10. Two mutations of Sieve and their impact on tracing. The solid arrows denote the flow of a faulty value.

Figure 8 revisits Sieve and shows the contracts of level `type` and `max` for three of its components. We denote components with the `define/component` construct, which allows to specify the list of possible contracts for each component. Since level `none` maps trivially to `any/c`, we omit it. The contracts in the figure are mostly the same as those for these three definitions in figure 2 from section 2. The two differences are that (i) we omit the tag-level contract (a.1) for `sift`, and we add a simple `type` level contract for `primes` alongside the partial-functional-correctness contract from figure 1. Moreover, in this version of the program we bundle all the contracts of each definition in the macro `contract-map`. To obtain the different configurations of the program, we choose a level for each definition at compile time, and keep only the contract of that level in the code we run. Put differently, with the `contract-map` macro we encode the configuration maps for a program's components in the program directly. Finally, since all mutants of Sieve have the same components as the original Sieve, we first add contracts to the latter and then mutate the components (but not the contracts). Figure 9 displays this process.

4.3 The Configuration Comparison

While the Blame Trail hypotheses tells us whether blame achieves its pragmatic goal of helping to narrow down the location of bugs, the Search Progress hypothesis examines whether it does so in a predictable and systematic manner. As we discuss briefly in section 3, the configuration comparison models what it means for blame shifting to move blame "closer" to the faulty component. Hence, the selection of the configuration comparison determines what we learn from examining the Search Progress hypothesis for a contract system.

Specifically, an unfortunate choice of configuration comparison can make the hypothesis trivially true for all contract systems and therefore uninteresting. For example, consider a comparison that returns true if a configuration is closer to the top of the lattice than another. By the construction

```
(define/component (include? hd n)
  (contract-map
    [max (->i ([hd integer?]
               [n integer?])
          [result (hd n)
                  (= hd (* n (/ hd n)))])]
    [type (-> integer? integer? boolean?)])
 (not (= 0 (modulo hd n))) )

(define/component (sift n st)
  (contract-map
    [max (-> (streamof integer?)
             sieved-stream/c)]
    [type (-> (streamof integer?)
              (streamof integer?))])
 (define-values (hd tl)
   (stream-unfold st))
 (cond [ (include? hd n)  (sift n tl)]
       [else
        (make-stream
         hd
         (λ () (sift n tl)))]))
```

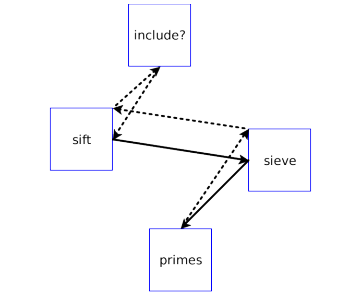


Fig. 11. A refactoring of `sift` and its impact on tracing. The dotted arrows denote influence, while the solid arrows denote the flow of a faulty value.

of the lattice, this is trivially true for all pairs of a configuration and its ascendant along a blame trail, independently of the design of the contract system. Indeed, we do not even need to run the experiment to decide that Search Progress holds!

Other comparisons that at first glance seem like promising candidates turn out to be equally uninteresting. For example, an instinctive choice is to define the comparison based on the distance between the blamed component and the component that contains the fault in the stack trace of a contract violation. However, the stack trace may contain neither the blamed component nor the faulty component. The first mutation of Sieve in figure 10 results in a program with such behavior. In a configuration of the mutant where `sift` has its `max` contract, the program raises a contract violation as it accesses elements of the `primes` stream. The violation points to `sift`, which is also the faulty component in this case, but `sift` is not on the stack of the program when the contract system signals the violation. Hence, this choice of comparison does not help us understand the relationship between blame and bugs even in simple scenarios.

In this light, we have constructed a specialized comparison for our Racket experiment based on the run time flow of the witness of a contract violation from component to component as we discuss in section 2. We cannot argue that our choice is canonical — indeed as we hint with the discussion about configuration comparisons we doubt that a canonical choice exists. That said, we are confident that our metric is suitable for this first study for two reasons. First, the comparison is based on the propagation of ownership annotations from the theory of blame [Dimoulas et al. 2011, 2012]. That theory formalizes intuitions from the practice of contracts in Racket, and has itself lead to numerous enhancements of Racket's contract system (e.g. [Dimoulas et al. 2011; Moore

et al. 2016; Takikawa et al. 2012]). Second (and post-facto), as we discuss in sections 5 and 6, the comparison has revealed interesting and subtle facts about Racket's contract system.

At a first approximation, our comparison relies on an execution trace that keeps track of the flow from one component to another of the witness of the contract violation. The right-hand side of figure 10 shows that information for two different mutations of Sieve. In the first mutation, `sift` places faulty values in its stream result that trickle all the way down to `primes`. In the second one, `primes` constructs a wrong starting stream for `sieve` and when the latter returns its result, the result contains multiples of 4. In both cases, our trace marks what components contribute directly in the flow of the faulty values that trigger the contract violation.

However, a direct role in the flow of these values is not the only way that a bug can cause a contract violation. Figure 11 shows a slight refactoring of the first mutant of Sieve from figure 10 where the faulty component `include?` influences indirectly the contract violation[9]. If we ignore this indirect influence and run this variant of Sieve in a configuration where the contract of `include?` is at `max` level, then the contract system blames the faulty `include?` but the latter is not on the trace. To remedy such situations, our trace keeps track not only of direct control of components on the witness of a contract violation but also indirect influence. The right-hand side of figure 11 explains the trace for this variant of Sieve pictorially. The interested reader can find a precise description of how we compute traces in the appendix in the form of a formal (PLT Redex) model.

Given now two traces for two significant configurations, such that the traces are related by a prefix relation, we define the result of the comparison to be true iff the distance between the occurrences of the blamed and faulty components in the first trace is less than or equal to that in the second trace. Because there may be multiple such occurrences, we conservatively pick the last occurrence of the blamed component and the first occurrence of the faulty one to compute the comparison. This choice is conservative in that it assumes that the bug was encountered the first time that the buggy component has control in the execution, even though the bug may not be triggered until a later point in the computation. We call the difference between the index of the faulty component and the blamed component the *blame-fault distance* of a configuration. For example, if the trace for a configuration that blames component `B` is `[A,B,C,F,C,B,A,C,F,B]` and the faulty component is `F`, then the blame-fault distance for the configuration is 6 since the first occurrence of `F` is at index 3 and the last occurrence of `B` is at index 9. Comparisons between blame-fault distances are only meaningful if they correspond to traces that are related by a prefix relation. For example, consider another trace entirely unrelated to the one above: `[F,C,B]`. The fact that the blame-fault distance for the configuration that produces this trace is only 2 has no meaningful relation to the blame-fault distance of 6 for the configuration above, because the traces represent completely different computations. Hence, our comparison is not defined on such pairs of configurations.

We have implemented an instrumentation framework that produces the trace for a program as a Racket language called `flow-trace`. In the first phase of the experiment we run each configuration under the instrumentation framework, and we stash each trace to analyze it in the second phase.

Our comparison must deal with a characteristic of contracts; contracts consist of ordinary code. Contract checking can therefore contribute to the trace of the program proper. Our instrumentation framework explicitly excludes contract code from extending the trace. After all, extensions to the trace due to contracts do not correspond to behavior of the program.

Extending the trace is not the only way that contract code can influence our comparison. The process of checking contracts may itself produce contract violations. There are two subtly different

[9]The post-condition of the new `->i` contract in figure 11 uses a Racket idiom; if `hd` is divisible by `n` then the post-condition evaluates to `#t` which is the contract that accepts only the value `#t`, and correspondingly for the opposite case

Table 4. Results summary.

program	lattice size	mutants	blame trails	Blame Trail	Search Progress
Dungeon	1.41×10^{28}	20	3728	× (276)	✓
Forth	7.62×10^{12}	48	8920	✓	✓
KCFA	6.17×10^{14}	34	6547	✓	× (3)
MBTA	4.05×10^{18}	32	5999	✓	✓
Morsecode	6.17×10^{14}	25	6011	✓	✓
Sieve	5.90×10^{4}	14	2658	✓	✓
Snake	2.05×10^{14}	55	9882	✓	✓

ways that this can occur. First, a contract may check a property of a value and this check may cause the value to violate another contract. Second, contract code may use a value from a faulty component that in turn triggers a contract violation. Since programmers cannot distinguish contract violations that occur in these ways from those that arise otherwise, our experiment treats all contract violations the same. We give concrete examples from our experiment for both of the above cases as part of the discussion of our experimental results in section 5.

4.4 Lattice Sizes and Sampling

As we mention above we treat each definitions as a component. A consequence of this decision is that examining the whole lattice for each of the mutants becomes unfeasible. As figure 2 shows, most of the programs contain dozens of definitions resulting in lattices with as many as 1.41×10^{28} configurations. Since running a single configuration can take 10 minutes or more (due to contract checking and instrumentation), running every possible configuration is impractical. Evidence from similar contexts suggests that randomly sampling the lattice instead of an exhaustive exploration is an effective alternative [Greenman et al. 2019]. Hence we randomly sample enough significant configurations from the lattice to obtain a confidence of 0.95 (with margin of error 0.05) about whether our hypotheses hold or not. See accompanying appendix for the details of the calculations for the sufficient size of samples for this estimate. Exploring more of the lattice would yield higher confidence in the generalizability of our results for each benchmark in cases where no violations of a property (e.g. Blame Trail) are found. In such cases, we could be more certain that we did not simply miss the configuration(s) that reveal the violation by exploring the lattice more fully. Our choice of random sample counts reflects the (informal) standard practice of estimating results to 95 percent or higher confidence.

5 RESULTS

We run our experiment on Northwestern University's Quest cluster and we set a maximum timeout of 4 hours and space limit of 6 GB per configuration.

The table in figure 4 summarizes the results of our experiment. For each program it reports:

- lattice size. Since mutants have all the same number of components and contract maps, the size of the lattice is the same for all mutants of program.
- mutants. Exploring all mutants of a program is not feasible. As we discuss in section 3, some mutants do not result in blame even for the top configuration of the lattice. Others have configurations that do not terminate within reasonable time or space constraints.
- blame trails. The cumulative number of trail paths we sampled for all considered mutants of a program.

- Blame Trail. Whether or not all blame trails across all mutants for a program satisfy the Blame Trail hypothesis (and count of how many blame trails violate it if any).
- Search Progress. Whether or not all blame trails across all mutants for a program satisfy the Search Progress hypothesis (and count of how many blame trails violate it if any).

We have uncovered violations of both hypotheses. In the remainder of this section we discuss these violations and what they imply for the design of Racket's contract system.

5.1 Blame Trail

Dungeon is the only program with mutants that fail Blame Trail. Since there are mutants with blame trails that terminate at a configuration that blames a component other than the faulty one, blame shifting in Dungeon does not always settle on the component containing the bug.

All of the problematic mutants exhibit a common pattern. In particular, the all such mutants have bugs which affect the order in which different components of Dungeon call a function that produces a stream of numbers; the order of such calls turns out to be critical for the functional correctness of the program.

To make the discussion concrete, consider the simplified program inspired by Dungeon in figure 12. Its `next-number!` function produces numbers from a pre-defined sequence and functions `asks-for-2-small-numbers` and `asks-for-1-small-number` use `next-number!`'s results to call `small-number-please`. The latter requires that its arguments are less than 10, and it has a contract that captures this constraint. The first function (`asks-for-2-small-numbers`) obtains numbers from the sequence and passes them to `small-number-please` in a loop that iterates twice; it does not verify that the numbers are appropriately sized. The second function (`asks-for-1-small-number`) does the same but only once. The original version of the program completes without issue because the sequence of numbers is constructed to start with three small numbers. `asks-for-2-small-numbers` provides the first two of those to `small-number-please`, and `asks-for-1-small-number` provides the third.

However, the mutation noted in `asks-for-2-small-numbers` causes a failure. It changes the number of iterations of the loop from 2 to 3, resulting in `asks-for-2-small-numbers` obtaining all three small numbers from the sequence. As a result, `asks-for-1-small-number` receives `30` from `next-number!` and the contract of `small-number-please` blames `asks-for-1-small-number`. The bug, however, is in `asks-for-2-small-numbers`, and none of Racket's contract combinators can be used to create a contract for `asks-for-1-small-number` that shifts the blame to `asks-for-2-small-numbers`. Hence, the blame settles on `asks-for-1-small-number` despite it not being the faulty component, violating Blame Trail.

In effect, the program assumes a protocol specifying the number of calls of `next-number!`, and Racket's contract combinators cannot express that protocol. While it is possible to write contracts that communicate using shared state to enforce the protocol, Racket's combinators provide no support for specifying such properties. As a result, the bug evades the contracts of the components of Dungeon and eventually changes the functional behavior of some component unrelated to the bug. The contracts therefore do detect the deviation from functional correctness, but the contract system cannot trace it back to the faulty component.

5.2 Search Progress

The table of results also shows that a few of the mutants of KCFA violate Progress Search. This means that, for those mutants, there are blame trails where strengthening the precision of contracts causes blame to move *further* rather than closer to the bug according to our configuration comparison.

```
#lang flow-trace

(define numbers '(1 2 3 30))

(define (next-number!)
  (define n (first numbers))
  (set! numbers (rest numbers))
  n)

(define/component
  (small-number-please n)
  ((<=/c 10) . -> . void?)
  #| omitted |#)

(define (asks-for-2-small-numbers)
  (for ([i (in-range 2 #| mutate to 3 |#)])
    (small-number-please
     (next-number!))))

(define (asks-for-1-small-number)
  (small-number-please
   (next-number!)))

(asks-for-2-small-numbers)
(asks-for-1-small-number)
```

Fig. 12. Simple program inspired by Dungeon that causes a Blame Trail violation.

```
#lang flow-trace

(define/component (wrap x)
  (-> (box/c number?)
      (box/c number?))
  x)

(define/component (wrap-again x)
  (contract-map
   [type (-> (box/c number?)
             (box/c number?))])
  (wrap x))

(define/component the-box
  (box/c number?)
  (wrap-again (box #f) #| BUG |# ))

(define/component (main x)
  (-> (λ (x) (number? (unbox x)))
      number?)
  (unbox x))

(main the-box)
```

Fig. 13. Simple program inspired by KCFA that causes a Progress Search violation.

Blame eventually does settle on the faulty component, as no mutant of KCFA violates Blame Trail, but in the process blame shifting seemingly leads to a detour.

These violations also fall into a common pattern that has to do with a quirk of our configuration comparison and, in particular, the way we compute the blame-fault distance. The simple program in figure 13 demonstrates the problem concretely. The program defines three functions (`wrap`, `wrap-again` and `main`), and a box (`the-box`). The box contains a boolean but, as the comment implies, should contain a number. All `box/c` contracts are "lazy"; the contract system checks them when a component attempts to access a box that has such a contract. The contract of `main`, in contrast, forces strict checking by accessing the box directly. In the configuration where the contract of `wrap-again` is at the `none` level, the contract system blames `wrap-again`, while in an ascendant configuration – where we strengthen the latter function's contract to the `type` level – the contract system blames `the-box`. The difference in blame is due to the different `box/c` contracts `the-box` accumulates as it flows through the program to `main` in the two configurations. The trace, which is common for both configurations, captures this flow:

```
[wrap,wrap-again,the-box,wrap-again,wrap,wrap-again,the-box,main]
```

The first seven entries correspond to the evaluation of the definitions while the last one comes from the call to `main`. The sub-trace

```
[the-box,wrap-again,wrap,wrap-again,the-box]
```

indicates the direct flow of the box through these functions. The first configuration blames `wrap-again`, and the second blames `the-box`. Hence, the blame-fault distance for the first is smaller than the second, despite the second blaming the faulty component. This is because we compute the blame-fault distance as the difference between the index of the first occurrence of the faulty component and the last of the blamed one.

In fact, the decrease in distance is an artifact of excluding contract code from the trace, as we discuss in section 4. Specifically, the trace does not contain any entries due to the contract of `main` accessing `the-box`. If we include those entries, evaluating the first configuration would extend the actual trace with the following suffix on the left, while the running the second configuration would extend the trace with the suffix on the right

```
[the-box,wrap-again,wrap],        [the-box,wrap-again,wrap,wrap-again]
```

In both cases, the traces would end with the point of the contract violation rather than the failed call to `main`, and the distance would decrease as expected.

In other words, in some cases where evaluating contract code triggers other contract checks, even if a configuration blames the faulty component, our comparison conservatively decides that it is further from the bug than a configuration that does not blame the faulty component. As the example demonstrates, this is necessary to obtain sound results given that our traces ignore points where a component influences the evaluation of contract code to capture only the behavior of the program proper we examine.

6 TO SHIFT OR NOT TO SHIFT?

The paper begins with a question; "Can programmers follow blame to find a bug?". Our results do not provide a definite answer. Both Blame Trail and Search Progress hold for the majority of programs and bugs we examined. This implies that in these scenarios, programmers can trust blame, increasing the precision of contracts along the blame trail, and reach the bug. However, the hypothesis violations we discovered indicate that this is not a generalizable strategy, at least for the Racket contract system today. Even though the outcome seems bleak, our experiment has highlighted both actionable limitations of Racket's contract system design and fundamental but subtle issues about the interactions of contracts with the code they monitor. It is worth noting that the theory of blame [Dimoulas et al. 2011, 2012] cannot uncover the same problems. First, the theory focuses on the semantics of blame rather than its pragmatic relation with bugs. Second, the theory deals with abstract models that capture the essence of contract system design without the corner cases of practical implementations. In that sense, the experiment has demonstrated that our experimental design is a method complementary to the theory of blame that deserves a place in the toolkit of programming language designers. This is especially timely in the context of recent work on contract systems and gradual typing that, for pragmatic reasons, explores different blame strategies (e.g. Reticulated Python [Vitousek et al. 2017]).

6.1 Lessons Learned

Each of the two ways that our experiment violated the hypotheses brought up distinct points about Racket's contract system and contracts generally. First, the violation of Blame Trail exemplifies an expressiveness problem for Racket contracts. Racket's contract combinators cannot express protocols like the one identified in `Dungeon`, even though they are quite common in real programs. For example, file system APIs implicitly come with protocols about when operations can be applied to a file, and in what order: an open file cannot be re-opened, a closed file cannot be read or written to, and so on. Protocols do not only describe temporal properties, but also other restrictions on the proper use of components, such as security. Thus, protocols are a natural extension for Racket's

contract system. In general, there have been some steps towards protocol contracts [Dimoulas et al. 2016; Disney et al. 2011; Heidegger et al. 2012; Moore et al. 2016, 2014; Scholliers et al. 2015], but understanding how to design and implement them remains largely an open research question.

The violations of Search Progress illustrate a nuanced point about how contracts affect program behavior beyond raising contract violations as expected; the experiment has highlighted a number of the subtle ways that this can manifest. The most obvious way that this happens is related to the length of a program trace. We would anticipate that increasing the precision of contracts always results in traces that are shorter (or equal) to the original, because more precise contracts should detect problems earlier. However, contracts may increase the length of a trace because checking contracts entails running more code than the original program (see for instance [Findler et al. 2007]). We describe this situation together with how our configuration comparison accounts for it by suppressing traces due to contract code in section 4.

Another way that contracts can affect programs is that they use program values that already have contracts. Contract code triggers violations of those contracts that may not manifest in a program with different contracts. That is, these violations are not related to the behavior of the program proper, but rather are due to the contracts exploring the code on their own[10]. In section 5, we explain how this is the cause of violations of Search Progress.

Beyond raising contract errors and extending the program trace, because contracts are ordinary code, they can cause arbitrary effects that in turn may influence the behavior of a program in unexpected ways. For example, in the course of analyzing preliminary data for KCFA, we discovered traces from pairs of configurations that differed substantially in ways that were difficult to explain. These situations arise because the contracts in one configuration subtly alter the internal state of the runtime in ways that the other configuration does not. These contracts do nothing unusual, they merely check whether a value is a member of a set. However, in Racket, every value can be assigned an identifying number by the runtime called an `eq-hash-code`; the way the runtime assigns these numbers is using a simple counter that increments on every request. If a value does not have such a number, asking whether that value is a member of a non-empty set causes the runtime to assign one. Thus, checking a contract can cause the runtime to assign more `eq-hash-codes` than in a program without the contract. Figure 14 demonstrates this situation with a small program; the program creates a set `s` of `data` structs and a function `f` which expects an argument that is not in the set. When we provide to `f` one such value, we observe that it is assigned an `eq-hash-code`. The last two lines of the example illustrate that `e` has been assigned an `eq-hash-code` before `d` due to `f`'s contract (the function `eq-hash-code` returns the code of its argument, requesting a new code if necessary). Normally, the behavior of programs should not rely on `eq-hash-codes`; however, the `eq-hash-code` of values in some data

```
#lang racket

(struct data (x))

(define s (set (data "a")))

(define/contract (f x)
  (-> (λ (x)
        (not (set-member? s x)))
      void?)
  (void))

(define d (data "d"))
(define e (data "e"))

(f e)

(eq-hash-code d) ;; => 12266959
(eq-hash-code e) ;; => 12266958
```

Fig. 14. The interplay between contracts and `eq-hash-codes`.

[10]This extra exploration is a well-known phenomenon of contracts which has an interesting interplay with laziness [Chitil et al. 2003; Degen et al. 2009, 2012; Dimoulas and Felleisen 2011; Hinze et al. 2006].

structures, such as a set, can influence the iteration order of elements in the data structure (along with other similar effects). Hence, the traces revealed how contract code that causes the runtime to assign more `eq-hash-codes` in one configuration but not the other can easily lead to changes in the program behavior. Since this situation can make the traces of two configurations incomparable for the experiment, our instrumentation framework replaces Racket's data structures with versions that do not depend on `eq-hash-code`.[11] As defined in section 4, configurations are only comparable if they produce traces related by a prefix relation. Hence, this compromise is necessary and we discuss its effect on the validity of our results in the next subsection.

In sum, our experiment has provided evidence for the need to understand the interaction between contracts and effects. This interaction goes both ways: how contracts can express the correct behavior of effectful programs, for example with protocols; and how we can protect both programs and contracts from unintentional interference due to effects.

6.2 Threats to Validity

We have identified a number of threats to the validity of our conclusions. First, even though we selected programs with a wide variety of Racket features exercising most of the interesting aspects of Racket's contract system, the programs do not cover the full range of Racket. For example, none of the programs use continuations. Furthermore, the programs don't use any concurrency and parallelism features, the interactions of which with contracts is in fact another largely unexplored area for contracts research [Shinnar 2011].

Second, another threat lies in our use of mutation to inject synthetic bugs. It remains unclear how the synthetic bugs from the mutation operators we use relate to bugs in the wild (compare for example Gopinath et al. [2014] and Just et al. [2014]), especially since many real faults may not even be attributable to a single location in a program [Thung et al. 2012] (this is a limitation of the experimental design as a whole). That said, mutation allows us to apply the experiment in a controlled manner: the nature of mutations being single syntactic variations makes defining the specific location of the fault obvious, and mutations allow us to inject a single fault in programs that we are highly confident are correct.

Third, our selection of contracts for the experiment is also a source of bias. There are many different properties of components one can select to express in a contract, and there are many ways one can implement a contract for a particular property. We implemented the contracts manually, so this opens the possibility that our conclusions fail to generalize to other contracts. To mitigate this threat, we focused on two non-trivial representative kinds of properties: type-like specifications, and partial functional correctness. For the type-like specifications, we followed closely the Typed Racket interfaces that came with the programs. As for functional correctness, we believe that variations in contract implementation are most significant in selecting relatively weak properties to enforce, and that as contracts approach a maximal specification such vagaries are minimized.

Fourth, as we state throughout the paper, the configuration comparison is an important parameter of the experiment and determines the interpretation of the second hypothesis. So our conclusions about how strengthening contracts shifts blame closer to the bug are not generalizable to other notions of distance. However, we have picked this particular comparison because we believe it captures the order in which different components contribute to a contract violation.

Fifth, our decision to use data structures with iteration order independent of `eq-hash-codes`, as we discuss above, is necessary given our selection of configuration comparison, but it deviates in some scenarios from the implementation of Racket. Nonetheless, the behavior of the data structures in our infrastructure conforms with the non-deterministic semantics of Racket about hash-based

[11] According to the Racket documentation, the order of iteration for these data structures is unspecified.

data structure iteration according to its documentation. The only program that we observed being affected by this discrepancy is KCFA.

Sixth, our ability to explore the lattices of the mutants was subject to the constraints of the running environment; it is possible that having access to more powerful machines would have allowed us to explore a larger portion of each lattice as well as more mutants.

7 RELATED WORK

Our work builds on a wide range of results across the areas of programming languages, software engineering, and security. Foremost, we draw inspiration from and rely upon research investigating contract systems and blame. Next, as we describe an evaluation of blame in terms of its relation to the location of bugs in programs, our work relates to research on fault localization and error report accuracy. Finally, our instantiation of the experiment for Racket builds upon ideas from program tracing, provenance, and mutation testing.

7.1 Contracts and Blame

Eiffel is the first programming language to popularize the idea and practice of contracts with the introduction of the "Design by Contract" methodology [Meyer 1988, 1991, 1992]. Findler and Felleisen [2002] use delayed checks to lift contracts to the world of higher order functions. This work has since led to a significant body of research on the design of higher order contract systems [Disney et al. 2011; Feltey et al. 2018; Findler and Blume 2006; Findler et al. 2007; Greenberg 2015; Greenberg et al. 2010; Heidegger et al. 2012; Hinze et al. 2006; Jia et al. 2016; Keil and Thiemann 2015; Moore et al. 2016, 2014; Scholliers et al. 2015; Strickland and Felleisen 2009a,b; Strickland et al. 2012; Swords et al. 2015; Takikawa et al. 2012; Waye et al. 2017] and their semantics [Blume and McAllester 2006; Degen et al. 2008, 2009, 2010, 2012; Dimoulas and Felleisen 2011; Dimoulas et al. 2011, 2012; Findler et al. 2004].

An aspect of the research on the semantics of contracts is to formally describe correctness criteria for blame [Dimoulas et al. 2011]. This work gives meaning to blame as a view of the flow of the witness of a contract violation, but does not investigate the pragmatic relationship between blame and bugs in programs, which is the aim of our paper. However, the theoretical work about blame has been a source of inspiration for the design of our experiment.

Blame also plays an important role in gradual typing [Ahmed et al. 2009; Garcia 2013; Igarashi et al. 2017; Siek et al. 2009, 2015a,b; Siek and Wadler 2010; Vitousek et al. 2014; Wadler and Findler 2009; Williams et al. 2018]. We anticipate that the experimental design we present is also applicable in that setting to explore and evaluate design strategies. For example, Vitousek et al. [2017] modify blame assignment dramatically to meet real-world practical constraints for gradual typing, and our method could help clarify whether blame in this setting can help narrow the search for the bug.

7.2 Fault Localization

The well-established area of fault localization is also related to our work. Its origins go back to the interactive debugging approach of Shapiro [1983], and modern automatic fault localization research build on the work of Agrawal ([Agrawal 1991; Agrawal et al. 1995]) and Jones et al. [2002], who use comparisons of successful and failing executions of a program to deduce a set of likely faulty program statements. Tangentially related along the thread of fault localization is extensive work on the accuracy of type checker error messages, the foundations of which are summarized by Heeren [2005]. However, we expressly do not aim (i) to propose a technique or evaluation method for fault localization, or (ii) to improve the accuracy of the error messages that a contract system produces. Rather, our goal is to analyze blame from a pragmatic perspective, and use this analysis to evaluate contract systems.

7.3 Provenance and Tracing

Our Racket experiment uses a configuration comparison defined over program traces, and hence we build upon a large body of work on program tracing. The tracing semantics of Perera et al. [2012] are a source of inspiration for our traces, though they focus on the application of tracing for debugging. Our traces also have high level similarities to work on provenance. Acar et al. [2013] describe a generic model of data provenance for higher order functional programming languages. Cheney et al. [2007] investigate a formulation of provenance in terms of dependency analysis, and they apply this understanding to databases. Cheney [2011] provides a formal definition of provenance with the aim of defining common security properties in terms of provenance. While these results are clearly related to our traces, we are unable to find a form of provenance or tracing that corresponds precisely to our traces.

7.4 Mutation Testing

Our Racket experiment obtains faulty programs using mutations. Research on mutation testing began with the work of DeMillo et al. [1988, 1978] and Lipton [1971], and has since seen significant interest in the field of software engineering. Jia and Harman [2011] provide a cogent overview of the history of mutation testing, its prevalent techniques, and its limitations. While mutation testing was first developed in the context of imperative programming languages, Le et al. [2014] describe the application of mutation testing techniques to higher order functional programs and demonstrate its effectiveness. However, the applicability of mutation testing techniques to generate faults in place of real faults in research is not immediately clear, and the kinds of faults generated by mutation are often quite distinct from those in real programs, as described by Gopinath et al. [2014]. On the other hand, Just et al. [2014] describe the traditional use of mutation testing for fault injection, and provides empirical evidence that such faults effectively simulate real faults in the context of test suite evaluation. We discuss what this implies for our work in section 6.

8 PRAGMATICS, NOT ONLY SEMANTICS

This paper introduces a principled method to explore whether the design of a contract system realizes the true purpose of blame; helping programmers track down bugs. As evidence in favor of our method, we use it to evaluate Racket's contract system. Even though the result of the evaluation is not definitive, the method highlights corner cases that demonstrate its utility as an analytical tool for understanding the pragmatic meaning of blame.

The pragmatic meaning of blame captures aspects of contract systems that semantics alone does not. In particular, it reflects how informative blame is with respect to bugs. Hence, language designers can use our method to guide the evolution of contract systems based on factors beyond semantic correctness. For example, the designers of Racket can now ask, "Does the introduction of the semantically correct `->i` result in pragmatic gains?", while the designers of Reticulated Python can ask, "How do practically-motivated tradeoffs that affect the precision of blame impact its pragmatic value?" In this light, this paper is a call to investigate the pragmatics of contract systems alongside their semantics.

ACKNOWLEDGMENTS

We would like to thank the POPL reviewers for their insightful feedback. Many thanks to Matthias Felleisen, Ben Greenman, Shu-Hung You, and Spencer Florence for their comments on earlier drafts of this work. Thanks to Quest for providing the resources to run our experiment. Thanks to the AEC reviewers for their comments on our artifact. Thanks to the NSF for their support of this work.

REFERENCES

Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. 2013. A core calculus for provenance. *Journal of Computer Security* 21, 6 (2013), 919–969.

Hiralal Agrawal. 1991. Towards automatic debugging of computer programs. *PhD thesis, Purdue University, SERC* (1991).

Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. 1995. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering*. IEEE, New York, NY, 143–151.

Amal Ahmed, Robert Bruce Findler, Jacob Matthews, and Philip Wadler. 2009. Blame for All. In *Proceedings for the 1st Workshop on Script to Program Evolution*. ACM, New York, NY, USA, 1–13.

Matthias Blume and David McAllester. 2006. Sound and complete models of contracts. *Journal of Functional Programming* 16, 4-5 (2006), 375–414.

James Cheney. 2011. A formal framework for provenance security. In *Computer Security Foundations Symposium (CSF)*. IEEE, New York, NY, 281–293.

James Cheney, Amal Ahmed, and Umut A. Acar. 2007. Provenance as dependency analysis. In *International Symposium on Database Programming Languages*. Springer, New York, NY, 138–152.

Olaf Chitil, Dan McNeill, and Colin Runciman. 2003. Lazy assertions. In *Symposium on Implementation and Application of Functional Languages*. Springer, New York, NY, 1–19.

Henry Coles. [n. d.]. Mutators Overview. http://pitest.org/quickstart/mutators/. Accessed: 2019-07-02.

Markus Degen, Peter Thiemann, and Stefan Wehr. 2008. Contract Monitoring and Call-by-name Evaluation. In *Nordic Workshop on Programming Theory*. Tallinn, Estonia.

Markus Degen, Peter Thiemann, and Stefan Wehr. 2009. True lies: Lazy contracts for lazy languages (faithfulness is better than laziness). In *4. Arbeitstagung Programmiersprachen*. Lübeck, Germany.

Markus Degen, Peter Thiemann, and Stefan Wehr. 2010. Eager and delayed contract monitoring for call-by-value and call-by-name evaluation. *The Journal of Logic and Algebraic Programming* 79, 7 (2010), 515–549.

Markus Degen, Peter Thiemann, and Stefan Wehr. 2012. The Interaction of Contracts and Laziness. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. ACM, New York, NY, USA, 97–106.

Richard A. DeMillo, Dana S. Guindi, Kim King, Mike M. McCracken, and Jefferson A. Offutt. 1988. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*. IEEE, New York, NY, 142–151.

Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.

Christos Dimoulas and Matthias Felleisen. 2011. On Contract Satisfaction in a Higher-Order World. *ACM Transactions on Programming Languages and Systems* 33, 5 (2011), 16:1 – 16:29.

Christos Dimoulas, Robert Bruce Findler, and Matthias Felleisen. 2013. Option contracts. In *ACM Conference on Object-Oriented Programming Systems, Languages & Applications*. ACM, New York, NY, 475–494.

Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. 2011. Correct Blame for Contracts: No More Scapegoating. In *ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 215 – 226.

Christos Dimoulas, Max S New, Robert Bruce Findler, and Matthias Felleisen. 2016. Oh Lord, please don't let contracts be misunderstood (functional pearl). In *ACM International Conference on Functional Programming*. ACM, New York, NY, 117–131.

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Complete monitors for behavioral contracts. In *European Symposium on Programming*. Springer, New York, NY, 214–233.

Tim Disney, Cormac Flanagan, and Jay McCarthy. 2011. Temporal higher-order contracts. In *ACM International Conference on Functional Programming*. ACM, New York, NY, 176–188.

Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. Collapsible contracts: fixing a pathology of gradual typing. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 133.

Robert Bruce Findler and Matthias Blume. 2006. Contracts as Pairs of Projections. In *Proceedings of the 8th International Symposium on Functional and Logic Programming*. Springer, New York, NY, 226–241.

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *ACM International Conference on Functional Programming*. ACM, New York, NY, 48–59.

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

Robert Bruce Findler, Shu-yu Guo, and Anne Rogers. 2007. Lazy Contract Checking for Immutable Data Structures. In *Revised Papers of the 16th International Workshop on Implementation of Functional Languages (IFL)*. Springer, New York, NY, 111–128.

Ronald Garcia. 2013. Calculating Threesomes, with Blame. In *ACM International Conference on Functional Programming*. ACM, New York, NY, 417–428.

Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Mutations: How close are they to real faults?. In *Software reliability engineering (ISSRE), 2014 IEEE 25th international symposium on*. IEEE, New York, NY, 189–200.

Michael Greenberg. 2015. Space-Efficient Manifest Contracts. In *ACM Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 181–194.

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2010. Contracts Made Manifest. In *ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 353–364.

Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. 2019. How to evaluate the performance of gradual type systems. *Journal of Functional Programming* 29 (2019), e4.

Bastiaan J Heeren. 2005. *Top quality type error messages*. Utrecht University.

Phillip Heidegger, Annette Bieniusa, and Peter Thiemann. 2012. Access permission contracts for scripting languages. In *ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 111–122.

Ralf Hinze, Johan Jeuring, and Andres Löh. 2006. Typed contracts for functional programming. In *International Symposium on Functional and Logic Programming*. Springer, New York, NY, 208–225.

Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. 2017. Gradual Session Types. *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 38 (Aug. 2017), 28 pages.

Limin Jia, Hannah Gommerstadt, and Frank Pfenning. 2016. Monitors and Blame Assignment for Higher-order Session Types. In *ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 582–594.

Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2011), 649–678.

James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *International Conference on Software Engineering*. IEEE, New York, NY, 467–477.

René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, 654–665.

Matthias Keil and Peter Thiemann. 2015. Blame Assignment for Higher-order Contracts with Intersection and Union. In *ACM International Conference on Functional Programming*. ACM, New York, NY, USA, 375–386.

Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. 2014. Mucheck: An extensible tool for mutation testing of Haskell programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. ACM, New York, NY, 429–432.

Richard J Lipton. 1971. Fault diagnosis of computer programs.

Bertrand Meyer. 1988. *Object-oriented Software Construction*. Prentice Hall, London, UK.

Bertrand Meyer. 1991. Design by Contract. In *Advances in Object-Oriented Software Engineering*. Prentice Hall, London, UK, 1–50.

Bertrand Meyer. 1992. *Eiffel: The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. 2016. Extensible access control with authorization contracts. In *ACM Conference on Object-Oriented Programming Systems, Languages & Applications*. ACM, New York, NY, 214–233.

Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. 2014. SHILL: A Secure Shell Scripting Language. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, USA, 183–199.

Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. 2012. Functional programs that explain their work. In *ACM International Conference on Functional Programming*. ACM, New York, NY, 365–376.

Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. 2015. Computational Contracts. *Science of Computer Programming* 98, P3 (Feb. 2015), 360–375.

Ehud Y. Shapiro. 1983. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA.

Avraham Ever Shinnar. 2011. *Safe and effective contracts*. Harvard University, Boston, MA.

Jeremy Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the design space of higher-order casts. In *European Symposium on Programming*. Springer, New York, NY, 17–31.

Jeremy Siek, Peter Thiemann, and Philip Wadler. 2015a. Blame and Coercion: Together Again for the First Time. In *ACM Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 425–435.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015b. Monotonic references for efficient gradual typing. In *European Symposium on Programming Languages and Systems*. Springer, New York, NY, 432–456.

Jeremy G. Siek and Philip Wadler. 2010. Threesomes, with and Without Blame. In *ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 365–376.

T. Stephen Strickland and Matthias Felleisen. 2009a. Contracts for first-class modules. In *Proceedings of the Symposium on Dynamic Languages*. ACM, New York, NY, 27–38.

T. Stephen Strickland and Matthias Felleisen. 2009b. Nested and Dynamic Contract Boundaries. In *International Workshop on Implementation of Functional Languages (IFL)*. Springer, New York, NY, 141 – 158.

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

Cameron Swords, Amr Sabry, and Sam Tobin-Hochstadt. 2015. Expressing Contract Monitors As Patterns of Communication. In *ACM International Conference on Functional Programming*. ACM, New York, NY, 387–399.

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S New, Jan Vitek, and Matthias Felleisen. 2016. Is sound gradual typing dead?. In *ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 456–468.

Asumu Takikawa, T Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual typing for first-class classes. In *ACM Conference on Object-Oriented Programming Systems, Languages & Applications*. ACM, New York, NY, 793–810.

Ferdian Thung, David Lo, Lingxiao Jiang, et al. 2012. Are faults localizable?. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. IEEE, New York, NY, 74–77.

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *Proceedings of the Symposium on Dynamic Languages*. ACM, New York, NY, USA, 45–56.

Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big types in little runtime: open-world soundness and collaborative blame for gradual type systems. In *ACM Symposium on Principles of Programming Languages*. ACM, New York, NY, 762–774.

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems*. Springer, New York, NY, 1–16.

Lucas Waye, Stephen Chong, and Christos Dimoulas. 2017. Whip: higher-order contracts for modern services. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 36.

Jack Williams, J. Garrett Morris, and Philip Wadler. 2018. The Root Cause of Blame: Contracts for Intersection and Union Types. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 134 (Oct. 2018), 29 pages.