How to Evaluate Blame for Gradual Types

LUKAS LAZAREK, PLT @ Northwestern University, USA
BEN GREENMAN, PLT @ Northeastern University, USA
MATTHIAS FELLEISEN, PLT @ Northeastern University, USA
CHRISTOS DIMOULAS, PLT @ Northwestern University, USA

Programming language theoreticians develop blame assignment systems and prove blame theorems for gradually typed programming languages. Practical implementations of gradual typing almost completely ignore the idea of blame assignment. This contrast raises the question whether blame provides any value to the working programmer and poses the challenge of how to evaluate the effectiveness of blame assignment strategies. This paper contributes (1) the first evaluation method for blame assignment strategies and (2) the results from applying it to three different semantics for gradual typing. These results cast doubt on the theoretical effectiveness of blame in gradual typing. In most scenarios, strategies with imprecise blame assignment are as helpful to a rationally acting programmer as strategies with provably correct blame.

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

Additional Key Words and Phrases: gradual typing, blame

ACM Reference Format:
Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2021. How to Evaluate Blame for Gradual Types. *Proc. ACM Program. Lang.* 5, ICFP, Article 68 (August 2021), 29 pages. https://doi.org/10.1145/3473573

1 DOES BLAME MATTER

Theoreticians of gradual typing have focused on blame theorems from the very beginning [Matthews and Findler 2009; Tobin-Hochstadt and Felleisen 2006]. "Well-typed [components][1] can't be blamed" turned the theorem into a slogan [Wadler and Findler 2009]. Academic systems (Reticulated Python [Vitousek et al. 2014, 2019, 2017] and Typed Racket [Tobin-Hochstadt and Felleisen 2006, 2008, 2010; Tobin-Hochstadt et al. 2017]) come with sophisticated checking and blame assignment strategies (sec. 2). Their academic creators embrace the idea that blame can help practicing programmers find impedance mismatches, that is, disagreements between the type ascriptions of a software component and its behavior.

Industrial implementors of gradual typing systems have almost completely ignored blame assignment. Systems such as Flow, Hack, or TypeScript[2] exploit types for IDE actions and for finding

[1] The original authors got this word wrong. A program has many components; blame helps identify a faulty one.
[2] See https://flow.org, https://hacklang.org, and https://www.typescriptlang.org, respectively.

Authors' addresses: Lukas Lazarek, PLT @ Northwestern University, Evanston, Illinois, USA, lukas.lazarek@eecs.northwestern.edu; Ben Greenman, PLT @ Northeastern University, Boston, Massachusetts, USA, benjaminlgreenman@gmail.com; Matthias Felleisen, PLT @ Northeastern University, Boston, Massachusetts, USA, matthias@ccs.neu.edu; Christos Dimoulas, PLT @ Northwestern University, Evanston, Illinois, USA, chrdimo@northwestern.edu.

2475-1421/2021/8-ART68
https://doi.org/10.1145/3473573

typos in code. Then their compilers remove types and rely on the built-in safety checks of the underlying language to catch any problems.

This contrast between theory and applications of gradual typing raises the question of

> *whether blame assignment adds any value to a gradually typed language, especially for the benefit of the working programmer.*

Given the long-standing academic interest in blame and its complete absence in industrial systems, it comes as an even bigger surprise that the research literature and the industrial blog world do not discuss any possible answers. Instead, when language designers make decisions concerning this aspect, they seem to go one way or another without any scientific justification. Indeed, the community has thus far failed to offer a method for evaluating blame assignment.

This paper's *first contribution* is *a method for evaluating the effectiveness of blame assignment strategies* in the gradual typing world. The top-level innovation is the idea of a *rational programmer*, that is, a programmer that acts only in response to available information (sec. 3). In the case of an impedance mismatch, the available information consists of the error message and the current state of the program. The rational programmer can hence use the former to change the latter—and this systematic, information-driven process can be implemented and tested, at scale, on real programs. Turning this idea into a scientific experiment requires overcoming major challenges: injecting representative impedance mismatches; putting the various kinds of error information to comparable use; and sampling the huge space of possibilities (see sec. 4 and, for details, secs. 5 through 7).

The paper's *second contribution* is a set of *results from applying the evaluation method to three distinct checking regimes for gradual types and their blame assignment strategies in approximately 72,000 different scenarios* (sec. 8): (1) *Transient*, i.e. Reticulated's inlined type assertions and collaborative tracking of typed/untyped boundaries [Vitousek et al. 2017]; (2) *Natural*, i.e. Typed Racket's use of a higher-order contract system and its blame assignment [Findler and Felleisen 2002]; and (3) *Erasure*, i.e. the approach of industrial systems, which forgo type checks and blame in favor of error messages from the safety checks of the underlying language. The results[3] (sec. 9) are at least somewhat surprising. In principle they validate the conjectures behind the work of theoreticians. Run-time type checks and blame work together to help with the search for impedance mismatches between the specified types and the behavior of untyped components. Natural's wrapper-based type checks and blame tracking are more useful than Transient's type in-lined assertions and "collective blame" tracking algorithm, which in turn are superior to Erasure. *But*, the application of the method also indicates problems with the expectations of theoreticians. In contrast with the theoretical differences between the methods, *Natural* is only marginally more useful than *Transient*, and neither of the two checking and blame assignment methods are highly superior to *Erasure*. Additionally, the cost of Transient's blame can be huge. In turn, these problems suggest that, on one hand, the existing theory does not properly predict the behavior of blame in real systems with real programs, and on the other hand, the existing practice lacks data and alternative experiments to assess the entire landscape of the pragmatics of blame.

2 HOW TO THINK ABOUT THREE BLAME SYSTEMS

The three blame strategies rely on three different ways of catching problems with types at run-time: *Natural*, *Transient*, and *Erasure*. This self-contained section summarizes these options with one illustrative example using (Typed) Racket syntax. The informed reader may wish to merely scan it.

Consider the program sketch in figure 1. Each box represents a module: the top bar lists the name and whether it is using typed (blue) or untyped (red) syntax.

[3]The full data set is too large to be hosted on the web, so it is freely available upon request – along with the infrastructure to obtain it. Please reach out at `lukas.lazarek@eecs.northwestern.edu`.

pack-lib (at the top right) represents a library that provides, among others, a function pack. The documentation says this function consumes JSON data and packages it in an association list. In reality, though, the function returns a hash table instead of the association list.

types (at the top left) is one of three modules that overlays types onto this library. This specific modules defines types in common to the two other typed libraries.

typed-pack-lib (at the mid-level on the left) imports pack and re-exports it as typed-pack asserting that it is a function that consumes JSON and returns a list associating Symbols with Strings. In other words, it formalizes the comments in pack-lib.

crypto-pack-lib (at the bottom left) also imports pack and assumes for it the same type as typed-pack-lib. It applies the function in the definition of the exported crypto-pack function, which encrypts its input before passing it to pack.

types : typed/racket

```
(provide Entry Entries)

(define-type Entry
  (Pairof Symbol String))

(define-type Entries
  (Listof Entry))
```

typed-pack-lib : typed/racket

```
(provide typed-pack)

(require types)
(require/typed pack-lib
  [pack (-> JSON Entries)])

(define typed-pack pack)
```

crypto-pack-lib : typed/racket

```
(provide crypto-pack)

(require types)
(require/typed pack-lib
 [pack (-> JSON Entries)])

(: crypto-pack (-> JSON Entries))
(define (crypto-pack d)
  (pack (encrypt d _ _ _)))
```

pack-lib : racket

```
(provide pack _ _ _)

_ _ _  dependencies     _ _ _
_ _ _  and definitions  _ _ _

(define (pack d)
  ;; process JSON data and
  ;; package as a dictionary
  ;; (association list)
  (make-hash _ _ _) ;; BUG!)
```

client : racket

```
(require json)
(require typed-pack-lib)
(reqired crypto-pack-lib)

_ _ _ other dependencies _ _ _
_ _ _ and definitions    _ _ _

;; read data from files, pack
;; and share securely

(define public-data
  (typed-pack
   (read-json
    "public-records")))

(define secret-data
  (crypto-pack
   (read-json
    "medical-records")))

 _ _ _ rest of client _ _ _
```

Fig. 1. One mixed-typed program, three interpretations

`client` (at the bottom right) uses `pack` indirectly. Specifically, it goes through the two intermediary typed modules to use it. Imagine a programmer who relies on the types in the blue modules as checked documentation but prototypes the client in the untyped language.

The mistaken comment in `pack-lib` causes an impedance mismatch, with which each of the three semantics deals differently. Under *the Natural semantics*, functions imported into and exported from typed modules are wrapped in proxies that enforce the static type discipline with run-time checks and track responsibilities [Tobin-Hochstadt and Felleisen 2008; Tobin-Hochstadt et al. 2017]. Thus, when `pack` is imported into a typed module, the run-time system checks that it is a function and wraps it in a protective proxy, which in turn enforces the type of the function result with run-time checks. Analogously, the run-time system wraps each exported function of a typed module such as `crypto-pack` in a proxy that checks its arguments. These checks protect functions exported from typed modules against applications to wrong arguments in untyped code.

As this analysis implies, if a return-type check fails, the problem is that the untyped module, here `pack-lib`, supplied a function that is not a match for the type ascribed by the typed module. Hence either the type at the boundary between the two modules is wrong or, if the programmer trusts the type, the untyped module is at fault. If an argument-type check fails, responsibility lies with `client`. After all, either the type it ascribes to the argument is wrong or the argument it produced clashes with the type. Due to proxies, *Natural* can easily track the boundary, type, and responsible parties that correspond to each check. Thus, in the example of figure 1, as `pack` returns, the return-type check fails and Natural blames the boundary of `pack-lib` with `typed-pack-lib` and `crypto-pack-lib`, respectively, for the two `defines` in `client`.

Under *Transient*, typed code is compiled so that all entry points to functions check their arguments at run time and all function calls check their return values against the expected type [Vitousek et al. 2017]. Furthermore, Transient uses *shallow* checks, meaning they inspect only the top constructor of a value. Since retrieving a value from within a structure (or list, array, hash table etc.) is performed via a function call, the contents of a complex value are checked on a piecemeal basis.

As a result, the call to `typed-pack` does *not* signal an error because it takes place in the untyped `client` module, which is compiled in the usual manner. Because `pack` is called in the `crypto-pack-lib` module, Transient's inlined checks make sure that the imported `pack` is a function and that its result is a list. This last check fails in `client`'s call to `crypto-pack`.

In order to locate the corresponding boundaries for failed checks, Transient maintains a map from values to the boundaries between typed and untyped modules that they cross, plus the corresponding types. In the example, the map records that `pack` crosses from `pack-lib` to `typed-pack-lib` and from `pack-lib` to `crypto-pack-lib` with the type that appears in the `required/typed` forms in the example. Since the failed check corresponds to the return type of `pack`, assuming that the type is correct, the responsible party is the source of the two boundary crossings: `pack-lib`. In general though, Transient blames more than one boundary. In fact, the theoretical work of Greenman et al. [2019a] shows that for some programs Transient constructs a blame sequence that excludes responsible parties and includes modules irrelevant to the failing check.

Under *Erasure*, the compiler checks the specified types and then discards them when it generates code. The generated code includes run-time checks that ensure the dynamic safety of all operations as specified in the underlying untyped language. Hence, neither the call to `typed-pack` nor the call to `crypto-pack` signals an error due to the gradual type system. If at some later point `client` tries to inspect the elements of the lists that `typed-pack` and `crypto-pack` are supposed to produce, Racket's safety checks signal a violation and point to some place in `client`. The information in this exception, plus its stack trace, may help the programmer find the source of the impedance mismatch between the specified types of `pack` in the two typed modules and its actual results.

The following table summarizes the illustration. Each cell describes the result of evaluating the column's definition (in `client`) under the row's semantics.

	`public-data`	`secret-data`
Natural	error, blaming the boundary between `pack-lib` and `typed-pack-lib`	error, blaming the boundary between `pack-lib` and `crypto-pack-lib`
Transient	no error	error, blaming the boundaries between `pack-lib` and `typed-pack-lib`/ `pack-lib` and `crypto-pack-lib`
Erasure	no error*	no error*

*but *Erasure* does signal an error on list access

3 WHAT IS A RATIONAL PROGRAMMER

The general challenge of evaluating blame is a methodological one. Unlike most current research on programming languages, the question seems to call for empirical studies similar to those of the human-computer interaction research area. At the same time, a significant result demands a large amount of data. As Lazarek et al. [2020] recently demonstrated, the way around this dilemma is to simulate a programmer algorithmically on a large set of programming scenarios.

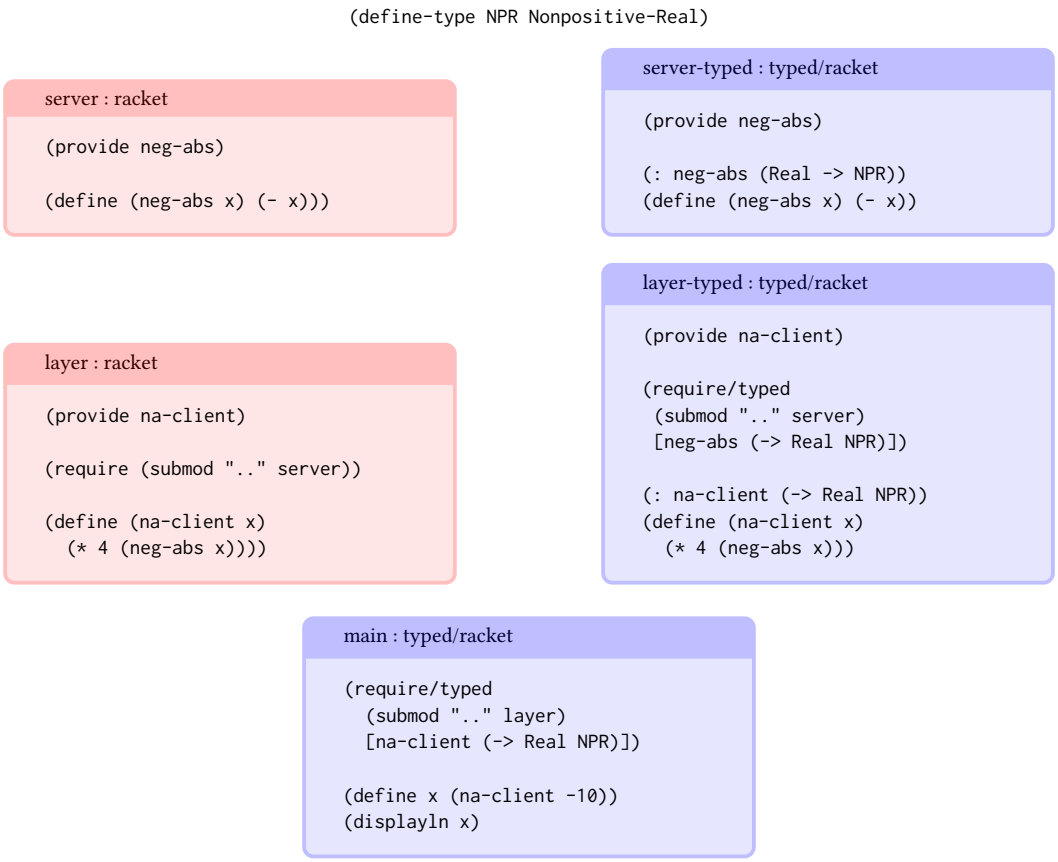


Fig. 2. A simplistic debugging scenario

This paper generalizes Lazarek et. al.'s idea to *the rational programmer*. Like Mill [1874]'s *homo economicus*, the rational programmer approximates the behavior of a software developer who reduces time spent on a task by exploiting the available information. In the context of gradual typing, the rational programmer has two pieces of information when an impedance mismatch signals exceptional behavior: the error message and the state of the program. Hence, the most rational procedure is to use the former to improve the latter. Specifically, the rational programmer translates the Wadler–Findler slogan into a debugging method, searching for the source of the impedance mismatch by adding type annotations to some of the untyped parts of the program identified in the error. If the type checker rejects an annotation derived from the context, the rational programmer has found the source of the problem. Otherwise, the rational programmer concludes that the just-annotated parts are not the problem and re-runs the program—which must, by the slogan, blame a different location for the problem. At this point, the rational programmer can iterate the process. Measuring this simulated behavior on a large number of scenarios yields data that is similar to data collected in a human-facing study.

The idea is best illustrated with an example in Typed Racket's migratory type system. Imagine a code base with dozens of modules in plain Racket. A developer who opens a module for maintenance purposes must study the module's design and, as part of the process, is bound to re-construct the types that went into the module's creation. To help future maintainers, the developer should report these insights as type annotations. Over time, the code base migrates into a mix of typed and untyped modules. As Tobin-Hochstadt et al. [2017] report though, it is equally common that developers add typed modules that depend on the existing modules in the code base.

Now consider the concrete (and simplistic) example of figure 2. Initially the code base consists of the two red modules on the left plus the blue module at the bottom; red indicates untyped, while blue means typed. When a typed module imports an untyped module, it must assign types to the imported identifiers for the type checker's sake. Here `main` specifies that `na-client` consumes a `Real` number and produces a non-positive one.[4] A program execution ends in this error:

```
na-client: broke its own contract
  promised: (<=/c 0)
  produced: 40
  in: (-> any/c (<=/c 0))
  contract from: (interface for na-client)
  blaming: (interface for na-client)
   (assuming the contract is correct)
```

The referenced contract is the compilation of the type of `na-client`. The definitive hint is "`blaming: (interface for na-client)`" with the caveat "`(assuming the contract is correct)`."

Assuming the rational programmer trusts the type of `na-client`, the next step is to inspect the `layer` module and to equip it with type annotations. The result is the blue module in the middle, and `main`'s import is now re-directed there by `(submod ".." layer-typed)`. As predicted by the theory, running the modified program (in the same way as before) yields a different error message:

```
neg-abs: broke its own contract
  promised: (<=/c 0)
  produced: 10
  in: (-> any/c (<=/c 0))
  contract from: (interface for neg-abs)
  blaming: (interface for neg-abs)
   (assuming the contract is correct)
```

[4]Racket's type system reifies reasoning about subsets of numbers, not machine-level representations [St-Amour et al. 2012].

Lastly, the rational programmer assigns types to `server` and re-directs the import of `layer-typed` to `(submod ".." server-typed)`. Now the type checker objects to the conjectured type of `neg-abs`, i.e. the source of the impedance mismatch is found. How to fix it is a separate question.

Like *homo economicus*, the rational programmer is an approximation. People do not behave in a purely rational manner as economic actors, and they also do not do so as software developers. The point is not to deny the existence of "lucky hunches" programmers or "tinkering works" approaches and so on. It is also not to claim that equipping entire modules with types represents an always feasible approach.[5] *But*, the concept of studying the idea of an economically rational actor has produced benefits to the discipline of political economics, and this paper suggests that implementing and studying the rational programmer will help language designers.

How to Turn the Idea of the Rational Programmer into a Methodology. Every time the rational programmer succeeds, it is validation for programming language researchers. It shows how their theorems, slogans, and tools help programmers. The example of blame assignment mechanisms makes this point clearly. A blame-assignment mechanism provides information that, according to programming language research, points toward the source of the problem.

When the rational programmer fails, it questions programming language research. Specifically, it indicates limited predictive power of programming language theory with respect to the use of languages in practice. Indeed, misleading predictions may even suggest flaws in language design.

In this way, the idea of a rational programmer implies an entire methodology for evaluating the design of programming languages. At this point, this study of methods supplies many questions whose answers might point to suitable evaluation methods:

(1) Does a language design provide information that can guide a rational programmer?
(2) Does the underlying theory suggest actions to the rational programmer?
(3) Can this guidance be formulated as an algorithm?
(4) Does the underlying theory lead to a hypothesis about the effects of these actions?
(5) Can this hypothesis be tested with a large-scale automated experiment?

Here is how the answers to these methodological questions lead to an evaluation method for blame-assignment mechanisms:

(1) The design of blame-assignment mechanisms explicitly advertises the blame information as helpful for debugging impedance mismatches.
(2) The error messages of blame-assignment mechanisms include suspect locations at the boundary of typed and untyped code fragments. The Wadler–Findler slogan suggests that the source of the problem is concealed due to a lack of types, so adding types to the untyped fragment should lead to the source of the impendance mismatch.
(3) The step-by-step construction of paths based on error messages from gradual-typing checks is clearly amenable to implementation, modulo the ascription of types to modules.
(4) The theory conjectures that blame assignment constrains the search space that a developer must inspect to find the problem.
(5) Based on these insights, the remaining sections detail a large-scale automated experiment.

That said, using the method to conduct data-gathering experiments poses several challenges. The specific challenges are spelled out in the next section, and the following three sections explain ways of overcoming them.

[5] Adding types at the expression level, say, as in TypeScript should be considered well within bounds.

4 WHY IT IS HARD TO EVALUATE BLAME

Implementing the method of the preceding section poses three challenges. The first concerns the comparison of the effect of blame on the rational programmer across three different mechanisms; the second challenge is about finding a large number of representative debugging scenarios; and the third is the resulting huge space of possibilities. A coincidental challenge is the disparity of the implementations of gradually typed languages. To eliminate this variable, the authors use Racket, which is thus far the only language in which all three major semantic variants are available in a robust and comparable manner: Typed Racket implements Natural, Shallow Racket [Greenman et al. 2021] Transient, and plain Racket Erasure.

The *first challenge* stems from the differences between the blame assignment mechanisms of the three semantic variants. While Natural assigns blame to *one* component, Transient assigns blame to a sequence of components. The Erasure semantics does not blame components *per se*, but it comes with an exception location and a stack trace, which implicitly suggest fixes. Each strategy triggers different reactions by the rational programmer (and real ones, too).

One way to reconcile these differences is to *equip the rational programmer with modes* that represent the different types of information the rational programmer takes into account when debugging a scenario. Intuitively, different blame strategies correspond to different modes of operation. For instance, one Transient mode may assign types to the oldest element of a blame sequence because it corresponds to the earliest point in the execution that can discover an impedance mismatch. Another mode may opt to treat the sequence as a stack and add types to its newest element. If both modes are equally successful in locating an impedance mismatch, measuring the rational programmer's debugging effort with each mode may answer which is the most effective.

However, attributing the success of the rational programmer to this or that blame mechanism demands a careful analysis of the interplay between blame and the run-time checks of each gradual type system. When a check fails, the Natural and Transient semantics assign blame instead of using the information in the exception from the run-time check. But, the exception information may be as useful to the rational programmer as a blame assignment. If this is the case, then blame *per se* may not play a critical role for the rational programmer. Indeed, precisely because they do not account for such confounding factors, Lazarek et al. [2020] cannot draw any conclusions about blame specifically, despite advertisements to the opposite. Their experiment may conclude only that so-called blame-shifting works, but they cannot attribute this conclusion to blame alone.

Modes offer a uniform way to compare the different semantics and isolate blame from the effect of the semantics' run-time checks. Specifically, the rational programmer comes with a blame mode and an exception mode for Natural and Transient. If the blame mode succeeds in debugging a program while the exception one fails, it is safe to conclude that blame is indeed beneficial for the rational programmer. Put differently, the exception mode serves as the *baseline* for blame's value within a given semantics; if the programmer in this mode performs as well or better than the blame one, a blame assignment mechanism might be useless.

An experiment must also rule out that the usefulness of blame assignment is sheer luck. Hence, a completely random mode provides yet another necessary baseline.

The *second challenge* is to find a representative collection of programs with impedance mismatches.[6] The impedance mismatches must represent mistakes that programmers accidentally create and that the run-time checks of academic systems catch. In other words, the experiment calls for a collection of mistakes in mixed-typed programs that is representative of those "in the wild." Unfortunately no such collection exists, and with good reason. The kind of mistakes needed

[6] Campora and Chen [2020] created a collection of Reticulated Python programs to evaluate their technique of fixing mistakes in type annotations. Their collection does not come with type-level mistakes in the code itself.

are typically detected by unit or integration tests; even if it takes some time to find their sources, these mistakes do not make it into code repositories with appropriate commit messages.

An alternative is to *generate a corpus of mistakes* using mutation analysis [DeMillo et al. 1978; Jia and Harman 2011; Lipton 1971], but conventional mutation analysis is useless. Mutation analysis traditionally aims to inject bugs that challenge test suites, and it discards those that yield ill-typed mutants as *incompetent*. Indeed, mutation analysis frameworks are fine-tuned to avoid them, and yet, it is precisely those mutators that are needed for evaluating blame assignment strategies.

Based on a related experience, Gopinath and Walkingshaw [2017] write, "existing mutation frameworks … do not generate the kinds of mutations needed to best evaluate type annotations" and, worse, "it is surprisingly difficult to come up with mutants that actually describe subtle type faults." While the goal of their work—to evaluate the quality of types in Python—is unrelated to blame, the mechanism is related. And their judgment confirms the experience of the authors.

An experimental analysis of blame needs a mostly new set of mutators. Roughly speaking, the new mutators inject type errors into fully typed programs. Applying such a mutator to any typed component produces a mutated component. A debugging scenario results from removing the types from the mutated component. For the design of such mutators, the authors relied on their own extensive programming experience though not without discovering a major pitfall: some of their original mutators systematically produced programs that immediately revealed the source of the impedance mismatch. All of the remaining ones yield *interesting debugging scenarios* (see sec. 6.3).

The *third challenge* is the explosive number of debugging scenarios that result from the combination of mutation-based scenario generation and mode-based analysis. All three factors—three different gradual typing systems, the large number of mutants, and the number of debugging modes—contribute possibly useful experimental data in a multiplicative manner. Hence, carried out naively, the experiment would demand an infeasible amount of computational resources. A practical execution has no option but to *sample the space of scenarios*, carefully ensuring reproducibility.

The next three sections explain how to overcome the three challenges in detail.

5 HOW TO MAKE COMPARABLE RATIONAL PROGRAMMERS

Section 3 explains how a migratory type setting helps with finding the source of an impedance mismatch. Roughly speaking, it encourages the rational programmer to equip a module with types if it is blamed in an error message. A sequence of such steps makes up a path in the lattice of type migration [Takikawa et al. 2016]. The lattice describes the space in which the modes of the rational programmer search for bugs (sec. 5.1).

Each mode receives different kinds of information and thus may construct different paths in the lattice. As discussed in section 4, evaluating blame relies on comparing modes of the rational programmer within the same semantics and across different semantics. Hence the research problem is to develop modes for the rational programmer and to make them comparable even when they correspond to different semantics and process different kinds of information (secs. 5.2 through 5.4). Programmer effort relative to a fixed debugging scenario introduces another dimension along which the modes become may be compared (sec. 5.5). With these notions in place, it becomes possible to state the experimental questions and describes the process (sec. 5.6).

5.1 The Lattice and the Debugging Scenario

Takikawa et al. [2016] describe the set of all possible type migrations with a lattice. A program P is a set of modules $\bar{c}$. Let a configuration s of P be the subset of $\bar{c}$ that comes with type annotations. These configurations of P are ordered by the subset relation and form a lattice $\mathcal{L}[\![P]\!]$ with $2^{|P|}$ elements. The bottom of $\mathcal{L}[\![P]\!]$ is $\emptyset$, the top one $\bar{c}$ itself; in between are the mixed-typed ones.

Applying a mutator to module c^* of P acts like a homomorphism on the generated lattice. The two lattices differ only in the mutated c^*. Given this formulation, debugging scenarios are those configurations in this new lattice that do not contain the mutated module; at all other configurations, the type checker points out the type-level mistake in the mutated c^*.

The actions of the rational programmer create an ascending chain—dubbed a *trail*—in $\mathcal{L}[\![P]\!]$. The *root* s_0 of a trail is the initial debugging scenario. If the program for some scenario s_i type-checks, the rational programmer runs the program until it raises a run-time error. The rational programmer then uses the information in the error to decide which module to equip with types. This choice constructs scenario s_{i+1} from s_i and thus lengthens the trail. A trail's construction ends successfully when it reaches a scenario that contains the mutated module because the type checker rejects its typed version outright. At this point, the source of the impedance mismatch is identified.

5.2 The Natural Rational Programmer

The Natural semantics assigns blame to exactly one boundary. A blame assignment has the following specific meaning: the typed module may make incorrect type assumptions about the untyped module in its interface, or the correct interface exposes a bug in the untyped module (or its dependencies). Our setup rules out the first alternative (but see sec. 11), and therefore the rational programmer extends the trail to a scenario that swaps out the untyped module for its typed counterpart.

Here is a rigorous definition.

> *A* Natural blame trail *is a sequence of scenarios* $s_0, \ldots s_n$ *of a program* P *such that for all* $0 \leq i \leq n-1$, $s_i \subset s_{i+1}$ *and*
>
> $$s_{i+1} \setminus s_i = \begin{cases} \{blame\,[\![P, s_i]\!]\} & \text{if } s_i \text{ produces blame} \\ \{exception\,[\![P, s_i]\!]\} & \text{otherwise} \end{cases}$$
>
> *where* $blame\,[\![P, s]\!]$ *denotes the module (of* P*) that* s *blames, and* $exception\,[\![P, s]\!]$ *denotes the first untyped location in the stacktrace produced by* s*.*

Note how, in the absence of "blame information" in the narrow sense, this definition interprets "blame information" broadly, as in any information from a failing run-time check. In particular, the definition rests on the two options for $s_{i+1} \setminus s_i$. The first part reflects the rational programmer's use of blame to extend the trail. However, blame may not be available in some scenarios. The second part accounts for those scenarios that produce errors from the runtime system before any impedance mismatch can be detected. For example, the fully untyped configuration can only produce such an error, e.g. from `length` receiving a boolean. Exceptions from the runtime do not carry blame, so the rational programmer proceeds using the accompanying stacktrace instead.

When the buggy untyped module of a program is replaced by the typed counterpart, the type checker fails because this module causes the impedance mismatch. Hence, a trail that ends at an ill-typed scenario successfully pinpoints the location of the bug.

> *A* Natural blame trail $s_0, \ldots s_n$ *in a lattice* $\mathcal{L}[\![P]\!]$ *is* successful *iff (the program for) its last scenario* s_n *does not type check. A Natural blame trail* $s_0, .., s_n$ *in a lattice* $\mathcal{L}[\![P]\!]$ *is* failing *iff (the program for)* s_n *type checks and the trail cannot be extended further.*

That is, failing Natural blame trails are those that end in a scenario that does not reveal the bug statically, yet also does not blame an untyped module. Thus the rational programmer has no further hints on how to continue the search for the bug.

While a successful Natural blame trail indicates that it pays off to heed blame assignments while debugging the trail's root, it does not answer whether blame is a critical piece of the rational programmer's process. For instance, typing the top of a failed run-time type check's stack trace, dubbed the location of the exception, might be as useful as typing the blamed one.

To account for this situation, a new mode of the Natural rational programmer follows a migration process based entirely on exceptions.

> *A* Natural exception trail *is a sequence of scenarios* $s_0, \ldots s_n$ *of a program* P *such that for all* $0 \leq i \leq n-1$, $s_i \subset s_{i+1}$ *and* $s_{i+1} \setminus s_i = \{\mathit{exception}\,[\![P, s_i]\!]\}$.

Using Natural exception trails, it becomes possible to factor out "blame information" in the narrow sense from the broad one of the above definition.

The definition of success for a Natural exception trail follows that for a Natural blame trail. Together, the definitions for the two modes allow the comparison of the usefulness of blame with that of mere exceptions for debugging a scenario in the context of Natural semantics.

> *Given a program* P *and a root* s_0 *in* $\mathcal{L}[\![P]\!]$, *Natural blame is* more useful *than Natural exceptions for debugging* s_0 *iff the Natural blame trail that starts at* s_0 *is successful while the Natural exception trail that starts at* s_0 *is failing.*

5.3 The Transient Rational Programmer

The Transient semantics assigns blame to a sequence of modules. The blame assignment says that the value witnessing the impedance mismatch may have crossed the boundaries between elements in the sequence, and that each crossing checked the value's type in a shallow manner.

This ambiguity in Transient blame raises the question of how the rational programmer should react when the language produces a blame sequence. Our answer is that the rational programmer has at least two reasonable options. The first one is to select the untyped module that is added to the blame sequence first and assign types to only that one—after all, if fully checked, the types of this first module should be able to detect an impedance mismatch earlier in the evaluation of a program than the later ones. The second option is to select the module that is added to the blame sequence last, effectively interpreting the blame sequence as a boundary-aware stack.

These two modes of rationalizing give rise to two different notions of trail.

> *A* Transient-first blame trail/Transient-last blame trail *is a sequence of scenarios* $s_0, \ldots s_n$ *of* P *where for all* $0 \leq i \leq n-1$, $s_i \subset s_{i+1}$ *and*
>
> $$s_{i+1} \setminus s_i = \begin{cases} \{\mathit{first}\,[\![\mathit{multiblame}\,[\![P, s_i]\!]]\!] \;/\; \mathit{last}\,[\![\mathit{multiblame}\,[\![P, s_i]\!]]\!]\} & \text{if } s_i \text{ produces blame} \\ \{\mathit{exception}\,[\![P, s_i]\!]\} & \text{otherwise} \end{cases}$$
>
> *where* $\mathit{first}\,[\![\mathit{multiblame}\,[\![P, s]\!]]\!] / \mathit{last}\,[\![\mathit{multiblame}\,[\![P, s]\!]]\!]$ *is the first/last module, respectively, that Transient adds to the blame sequence for* s.

The definition of *Transient exception trails* is analogous to that for Natural. It is used as a baseline for Transient-first and Transient-last blame.[7]

5.4 The Erasure Rational Programmer

Since gradually typed languages with Erasure semantics do not come with blame assignment, a rational programmer can only hope that the underlying safety checks and their exceptions are helpful. Thus, the Erasure rational programmer has a single mode, the Erasure exception mode, and its definition follows that for the Natural exception mode.

5.5 The Programmer Effort

In addition to success and failure information for the various trails, the experimental test bed can record the number of modules a rational programmer has to equip with types along each trail ($|s_n \setminus s_0|$). This number, the *debugging effort*, can serve as an additional metric to compare modes.

[7] A reader may wonder whether the rational programmer should just equip *all* modules in the Transient blame set with types. That might accelerate the search for the impedance mismatch, but if so it would also impose a large migration cost.

Comparing the effort of different modes of the rational programmer can illuminate the comparitive effectiveness of the three gradual typing systems. If, for instance, both Natural blame and Transient-first blame trails are successful for the same scenario, the two modes of the rational programmer can compete to see which one debugs the scenario with less effort. In general, if the effort distribution for a mode of the rational programmer has a shorter tail and more volume around smaller values compared to the effort distribution of another mode, then the first mode is probably the more effective of the two.

Measuring effort can also reveal whether the observed effectiveness of the rational programmer is an artifact of pure chance. In particular, the effort distribution for one mode can be compared with that of the random mode that ignores error information entirely and instead selects which module to type randomly. Since each mutant has a finite number of modules, random mode trails are always successful. However, the random mode's effort distribution should be thinly spread out across the range of trail lengths possible in the set of debugging scenarios. In contrast, the effort distribution of other modes should be quite different if their effectiveness is not coincidental.

5.6 The Experimental Questions

Trails and their properties provide the tools for a rigorous examination of blame for Natural, Transient, and Erasure. In line with the discussion so far, the test bed collects data to answer three initial questions for interesting debugging scenarios:

Q_1 Is blame useful in the context of Natural?
Q_2 Is first blame useful in the context of Transient?
Q_3 Is last blame useful in the context of Transient?

Furthermore, the experiment allows a comparison of the relative usefulness of blame information:

Q_* Is blame for X more useful than blame for Y (for X, Y in Natural, Transient, or Erasure)?

The nearby table summarizes how each question relates to different kinds of trails/modes of the rational programmer. For example, experimental question Q_1 asks whether blame is valuable for Natural and the experiment uses the Natural blame and exception trails to answer it, so Q_1 shows up in the cells for Natural blame and Natural exceptions.

	Natural	Transient	Erasure
Blame	Q_1/Q_*		
First blame		Q_2/Q_*	
Last blame		Q_3/Q_*	
Exceptions	Q_1	Q_2/Q_3	Q_*

In detail, the answer to Q_1 demands a comparison of the success of the Natural blame and Natural exception trails for all debugging scenarios. The first step is to construct each mutant's scenario lattice and identify their debugging scenarios. The test bed then extends the trails that start from those roots according to the Natural-blame programmer. If no scenarios can be added to the trail, the test bed checks whether the last scenario of the trail type-checks or not. If it does not, the Natural-blame trail is successful; otherwise it is failing. Figure 3 summarizes this experimental process for one mode of the rational programmer and connects it with the mutations from section 6. The process is repeated for the same roots with the Natural-exceptions mode. After completing, the test bed reports the success/failure results of the trails to determine the proportion of scenarios where Natural-blame is more useful than Natural-exceptions. Question Q_1 has a positive answer if a root exists where the above is true because it is evidence that there is at least one interesting scenario that the rational programmer manages to debug because of blame information. The process is analogous for Q_2 and Q_3, using the respective modes.

For Q_*, the process is a bit more involved. Answering this question calls for a comparison of the percentage of scenarios where one mode is more useful than the other and the inverse. For instance, deciding whether blame for Natural is more useful than Transient-first requires comparing the

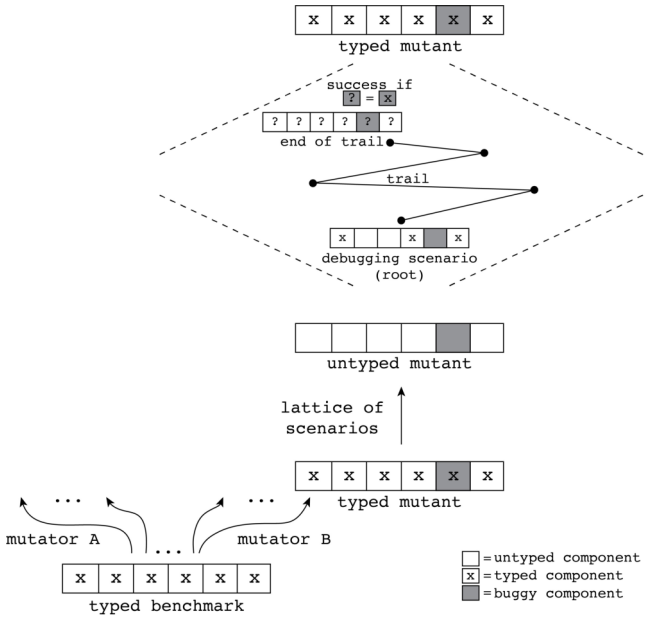


Fig. 3. The experimental process for one mode of the rational programmer

percentage of scenarios where the first is more successful than the second with the percentage where the second is more successful than the first. Repeating the whole process for every pair of modes produces a complete picture of the comparative usefulness of blame.

6 HOW TO MAKE LOTS OF MISTAKES

Putting the rational programmer to work means generating many mutants and turning those into debugging scenarios. The process must start with a suitable collection of representative programs (sec. 6.1). Since existing mutators do not generate useful mutants, the next step is to develop new mutators (sec. 6.2) and to validate their suitability on the benchmarks (sec. 6.3).

6.1 The Experimental Benchmarks

The benchmark programs for a rational-programmer experiment must

(1) vary in size, complexity and purpose;
(2) be fully typed so that the choice of types is fixed;
(3) take advantage of the variety of typing features of a gradually typed; and
(4) have a decent number of type-able modules and a variety of module dependency graphs because mixing of typed and untyped code in Typed Racket takes place at the module level.

Greenman et al. [2019b]'s collection of Typed Racket programs for systematically measuring the implementation's performance satisfies these criteria. The benchmark suite consists of fully typed, correct programs, written by a number of different authors who had maintained and evolved these programs over time. The programs range widely in size, complexity, purpose, origin, and in programming style. They rely on many Typed Racket features: occurrence typing [Tobin-Hochstadt and Felleisen 2010], types for mutable and immutable data structures [Prashanth and Tobin-Hochstadt 2010], types for first-class classes and objects [Takikawa et al. 2012], and types for Racket's numeric tower [St-Amour et al. 2012]. Finally, all of the programs are deterministic, so

Table 1. Summary of benchmarks

name	description	author	loc	mod.
acquire	object-oriented board game implementation	M. Felleisen	1941	9
gregor	utilities for calendar dates	J. Zeppieri	2336	13
kcfa	functional implementation of 2CFA for λ calculus	M. Might	328	7
quadT	converter from S-expression source code to PDF	M. Butterick	7396	14
quadU	converter from S-expression source code to PDF	B. Greenman	7282	14
snake	functional implementation of the Snake game	D. Van Horn	182	8
synth	converter of notes and drum beats to WAV	V. St-Amour	871	10
take5	mixin-based card game simulator	M.Felleisen	465	8
tetris	functional implementation of Tetris	D. Van Horn	280	9
suffixtree	algorithm for common longest subsequences between strings	D. Yoo	1500	6

any changes in the programs' behavior between runs can be solely attributed to the actions of the rational programmer.

Table 1 describes the ten benchmark programs that meet all the criteria, and furthermore come with the largest dependency graphs. This additional filter reflects that Typed Racket demands type assignments for entire modules, and so finding errors in benchmarks with small dependency graphs would almost be trivial for the rational programmer.

6.2 How to Mutate Software

A *mutator* performs a localized syntactic change to a code base. The result is a *mutant*.

For the evaluation of a blame strategy, mutators must produce type-level mistakes that the run-time checks of gradual typing systems or the safety checks of the underlying language can detect. Once detected, the rational programmer should be able to locate the mistake by gradually adding types to blamed modules. In other words, the suitability of the mutators hinges on their ability to generate interesting debugging scenarios (see sec. 6.3).

Table 2 describes 16 mutators that satisfy these constraints. As the last column indicates, some specialize or generalize Lazarek et al. [2020]'s mutators, which in turn are borrowed from the vast literature on mutation testing [Jia and Harman 2011]. Only two are directly inherited; many mutators are brand new. For the latter, the authors relied on their decades-long experience of making type-level mistakes in Typed Racket, some of which take non-trivial effort to debug.

Most of the mutators are self-explanatory. The first six apply to all gradually typed languages; the next six to those that include classes and objects. The last four target distinguishing features of Typed Racket's type system, specifically its sophisticated type system. For example, one mutation produced by `arithmetic` replaces a `+` with a `-` in an attempt to change the type of the arithmetic expression; `+`'s result is a `Positive-Integer` when all arguments are positive integers, while `-` yields `Integer` [St-Amour et al. 2012]. Similarly, the other three aim to confound the occurrence type system. Figure 4 illustrates how this confusion works. The function deals with an input that is either a `Real` or `#false`; the conditional deals with the first type in the `then` branch and the second type in the `else` branch. If a mutator wraps `(not ·)` around the test of the conditional, the resulting

```
(: deal-with [(U Real False) -> Real])
(define (deal-with optional-result)
  (if optional-result
      (+ optional-result OFFSET)
      DEFAULT))

(define DEFAULT 40)
(define OFFSET  11)
```

Fig. 4. Example program using occurrence typing

Table 2. Summary of mutators

name	description	example	origin
`constant`	swaps a constant with another of different type	`5.6` → `5.6+0.0i`	+
`deletion`	deletes the final expression from a sequence	`(begin x y z)` → `(begin x y)`	+
`position`	swaps two sub-expressions	`(f a 42 "b" 0)` → `(f a 42 0 "b")`	++
`list`	replaces append with cons	`append` → `cons`	new
`top-level-id`	swaps identifiers defined in the same module	`(f x 42)` → `(g x 42)`	new
`imported-id`	swaps identifiers imported from the same module	`(f x 42)` → `(g x 42)`	new
`method-id`	swaps two method identifiers	`(send o f x 42)` → `(send o g x 42)`	new
`field-id`	swaps two field identifiers	`(get-field o f)` → `(get-field o g)`	new
`class:init`	swaps values of class initializers	`(new c [a 5] [b "hello"])` → `(new c [a "hello"] [b 5])`	new
`class:parent`	replaces the parent of classes with `object%`	`(class a% (super-new))` → `(class object% (super-new))`	new
`class:public`	makes a public method private and vice versa	`(class object% (define/public (m x) x))` → `(class object% (define/private (m x) x))`	++
`class:super`	removes `super-new` calls	`(class a% (super-new))` → `(class a% (void))`	new
`arithmetic`	swaps arithmetic operators	`+` → `-`	++
`boolean`	swaps and and or	`and` → `or`	‡
`negate-cond`	negates conditional test expressions	`(if (= x 0) t e)` → `(if (not (= x 0)) t e)`	‡
`force-cond`	replaces conditional test expressions with `#t`	`(if (= x 0) t e)` → `(if #t t e)`	new

‡ inherited from, + specializes one of, ++ generalizes one of Lazarek et al. [2020]'s mutators

mutant is ill-typed and, when run, this function eventually causes a run-time type check to signal an impedance mismatch.

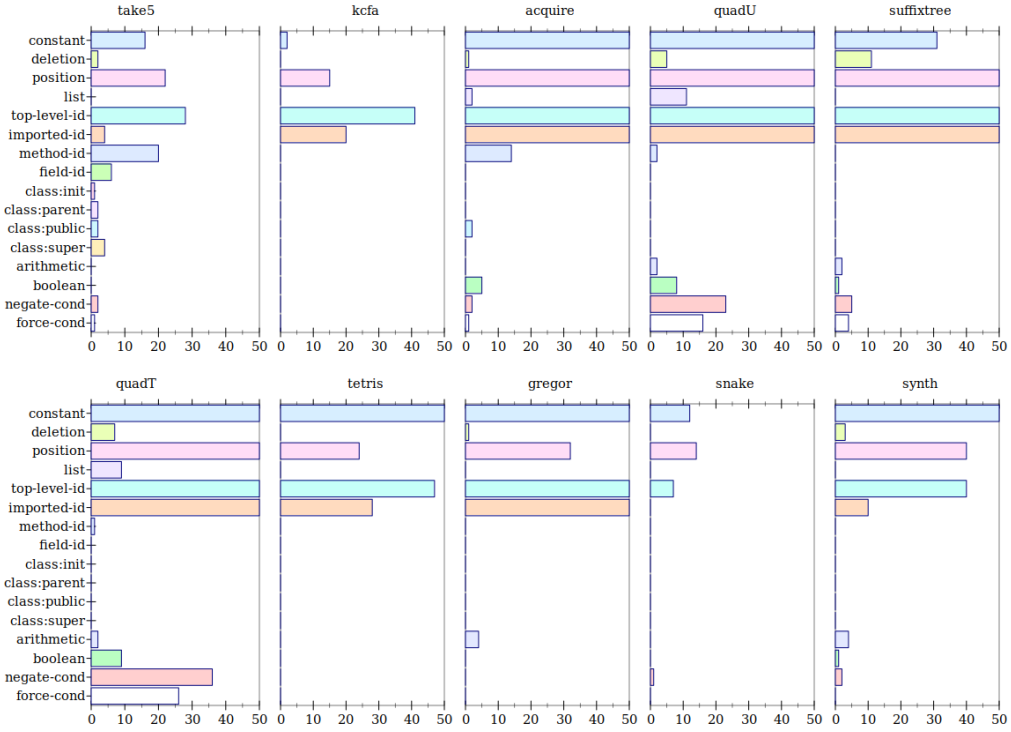
6.3 Are These Mutators Interesting

A type-level mutation is *interesting* (1) if the type checker rejects the fully typed version of the mutant, (2) running the mutant with all type annotations removed raises a run-time error, and (3) that error's stack trace contains source locations from at least three modules.

Here is the rationale for these three conditions:

(1) An impedance mismatch is a clash between the type ascription of one module's imports and another module's exports. Hence, type checking should fail for an interesting mutant.

(2) The goal of a comparative evaluation is to give the rational programmer a chance to debug the same scenario using different pieces of information. In the case of gradual typing semantics, a meta-theorem due to Greenman and Felleisen [2018] says that if a program raises an



Each plot shows a breakdown of interesting mutants by mutator. Each mutator corresponds to a bar representing the number of interesting mutants generated by that mutator. The counts are cut off at 50, so those bars reaching the edge of the plot represent 50 or more interesting mutants.

Fig. 5. Breakdown of interesting mutants by mutator, per benchmark.

exception under Erasure, it also errors under all other semantics. Hence, a comparison of blame information insists that an interesting mutant *raises a run-time exception under Erasure*.
Note While this choice favors Erasure over Transient and Natural and, for the same reason, Transient over Natural, some form of bias towards one or the other semantics is unavoidable. Tipping the scales in favor of the theoretically weakest semantics yields the most stable results. Section 9 includes some further discussion of this choice.

(3) If the evaluation of a mutated module immediately raises an exception because of the changes, there is no work for the rational programmer. Indeed, if the stack trace contains source pointers to two modules, the scenario is still uninteresting. Every ordinary benchmark program comes with a `main` module that acts as a driver, whose source is guaranteed to be included in the stack trace. Hence, the definition of interesting mutation insists on the presence of three different modules in the stack trace. This guarantees that the debugging scenario demands a sufficiently sophisticated effort, due to the interaction between the buggy module with its context. In these cases, the rational programmer must contend with at least two modules involved in a faulty interaction.

The definition of interesting mutants creates a powerful filter. All together, the listed mutators produce 16,800 interesting mutants across all benchmarks; see figure 5 for an overview. Broken

down by benchmark, the mutators produce at least 40 interesting mutants for every benchmark, and these mutants originate from at least four different mutators per benchmark. Thus, the mutators result in a sizable and diverse population of scenarios for every benchmark. Furthermore, every mutator contributes interesting mutants in at least one benchmark. Some mutators apply only to a few benchmarks, because they target rather specific features; for instance, the class-focused mutators are mainly effective in a program that makes extensive use of object-oriented features.

The goal of filtering for interesting mutants guided countless iterations of adding, removing, and refining mutators in table 2. For an illustrative example, consider a candidate mutator that casts the tests of conditionals to the `Any` type. Like the example explained at the end of the preceding subsection, this mutant would suppress occurrence typing. But, it would not be interesting because an execution would not raise a run-time error; instead the function would process its input as if nothing had changed. Hence this candidate mutator is not included in the final set.

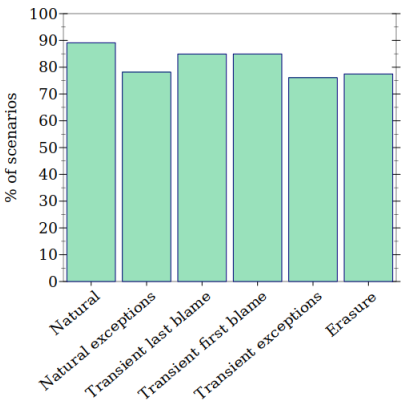
7 HOW TO MAKE THE EXPERIMENTAL SPACE TRACTABLE

As is, the chosen mutators generate approximately one million debugging scenarios for the chosen benchmarks. This number of scenarios is far too large to even identify the interesting ones among them. Furthermore this population is heterogeneous; scenarios come from different mutants, and the mutants are the result of different mutators applied to each benchmark.

To render the experiment computationally feasible and statistically sound, it becomes necessary to sample this large space in a uniform and stratified manner. The first two levels of stratification group mutants first by benchmark and then by mutator, making sure that the sample within each benchmark reflects the diversity of mutants with respect to the mutators that generated them. Specifically, the experiment samples 80 interesting mutants per benchmark, evenly-distributed across all of the mutators that contribute mutants for the benchmark. Some benchmarks have less than 80 mutants with interesting scenarios, in which case the only choice is to include them all. The result is a total of 752 interesting mutants across all benchmarks. Finally, the third level of sampling randomly draws 96 debugging scenarios from each configuration lattice with replacement. The final sample thus consists of 72,192 interesting scenarios.

8 WHAT ARE THE OUTCOMES OF THE EXPERIMENT

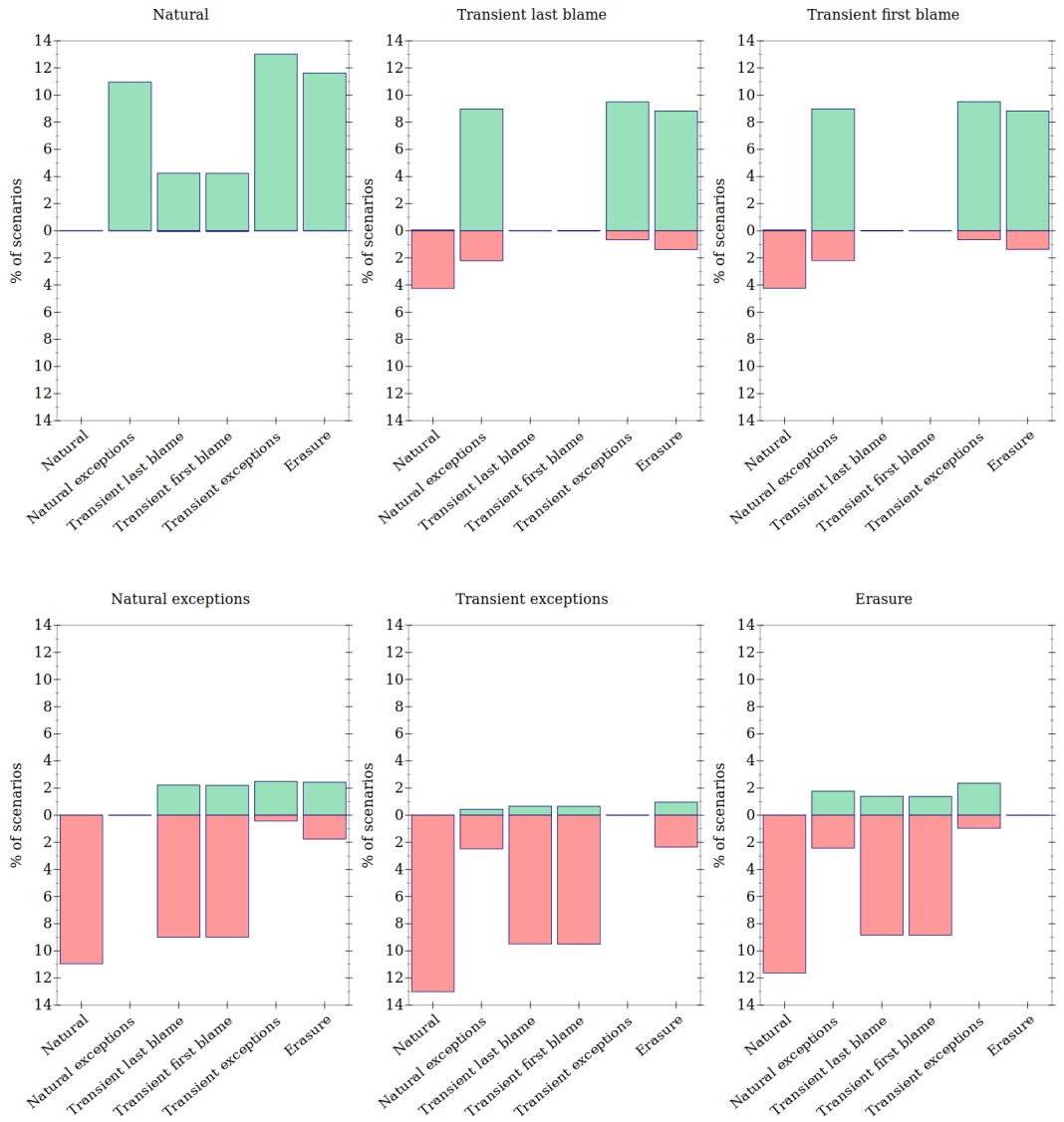
The test bed for executing the experimental process utilizes a machine with two Intel Xeon Gold 6258R processors (28 doubly-threaded cores each) and 500GB of memory. Each debugging scenario had a 4 minute timeout and a 6GB memory limit. Running the experiment on all debugging scenarios took over 30,000 compute hours or roughly three-and-a-half compute years.



The upper bound margin of error is 0.02%.

Fig. 6. Percentage rates of success

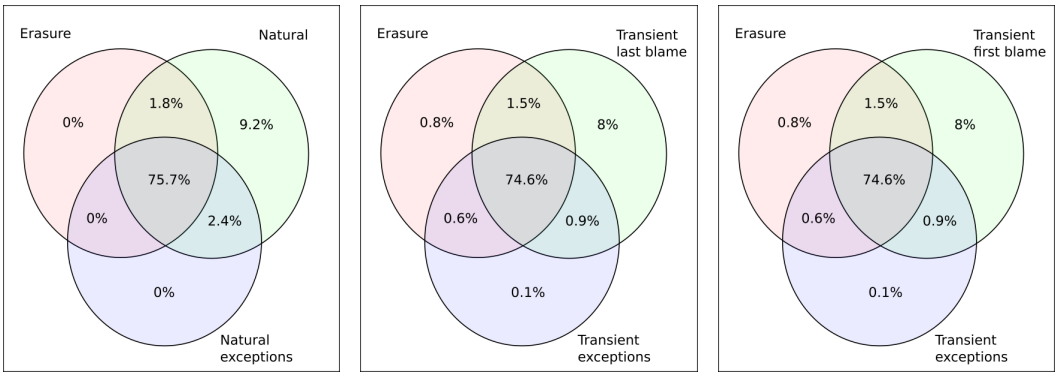
Figure 6 summarizes the overall success rates of every mode. The success rates illustrate a few points that underlie the rest of the analysis. The first notable piece of information from this figure is that every mode has failed debugging scenarios, not just Erasure. This should not come as a surprise to the astute reader. Running a rational programmer mode on a scenario may result in an exception that carries no useful information about which module to equip with types next. For instance the stack trace of the exception may not contain frames from any untyped module of the program. This can happen at any point along a blame trail, causing it to fail.



Each plot depicts a head-to-head comparison of the mode named above the plot vs. every other mode. The (green) portion above 0 is the estimated percentage of scenarios where the named mode is more useful than the other. The (red) portion below 0 is the estimated percentage of scenarios where the named mode is less useful than the other. The upper bound margin of error is 0.02%.

Fig. 7. Usefulness comparisons

While most blame trail failures follow the above pattern, a few do not. Breaking down the failure reasons for Natural blame (1748 in total) reveals an additional cause. For a small set of debugging scenarios (40), Natural produces a run-time type error blaming a non-buggy already-typed module. All these cases are due to known open issues with Typed Racket and class contracts.



Each diagram shows the overlap of the successful scenarios for three modes. For example, in the leftmost diagram, all three modes succeed on the same scenario 57.3% of the time, only Natural and Natural exceptions succeed on 29.1% of the scenarios, only Natural and Erasure succeed on 2.1%, and Natural alone succeeds on 9%. The upper bound margin of error is 0.02%.

Fig. 8. Blame usefulness analysis

In Transient, similar to Natural, most failures are due to unhelpful exception information (1851 for both Transient first and last blame). However, Transient also has a substantial number of failures because scenarios hit the time and/or memory limits of the experiment (~770 scenarios). Additionally, there are nearly 1,000 cases where Transient reports an empty blame set, leaving the rational programmer without hints about how to proceed. Sections 9.5 and 9.6 address these causes of failure for Transient and how they affect the experiment.

The second key observation from figure 6 is that the modes that use blame all outperform those that do not. In particular, Natural and both of Transient's blame modes succeed in 85 - 90% of the scenarios, while their corresponding exception modes succeed in less than 80% of them, and so too for Erasure. The only exception is that the random programmer always succeeds; the figure omits this mode because it just reflects the fact that every scenario has finitely many modules, so the random programmer eventually types the buggy module.

Figure 7 depicts a head-to-head comparison of every mode's performance against every other mode (except Random). The comparison answers the four questions from section 5.6. Each plot shows the proportion of scenarios where one mode performs better or worse than each other mode. In particular, each bar above zero represents the proportion where the plot's named mode succeeds and the mode on the x-axis fails; the corresponding bar below zero represents the proportion of the inverse case. For example, the plot titled "Natural" shows that Natural outperforms Natural exceptions in about 11% of the scenarios, and the inverse (Natural performs worse than Natural exceptions) never happens. Similarly, the plot titled "Transient last blame" shows that Transient last blame outperforms Natural exceptions in about 9% of the scenarios, but conversely it performs worse than Natural exceptions in about 2% of the scenarios.

The figure answers questions Q_1, Q_2, and Q_3 affirmatively. In all three semantics, blame modes outperform their corresponding exception mode by ~10%. The Natural exceptions mode is never more useful than Natural blame, and Transient exceptions are more useful than Transient first and Transient last blame in a small percentage (less than 1%) of the scenarios.

Figure 7 also provides answers for Q_*. Blame for all three semantics is significantly more useful than Erasure exceptions—by almost 12% for Natural and almost 9% for Transient. Natural blame is more useful than both versions of Transient blame by a small percentage (about 4%). The Transient

first and Transient last blame are practically indistinguishable. Finally, Natural exceptions are more useful than Transient exceptions, although only in a small percentage of scenarios (about 2.5%). A rare few scenarios (about 0.5%) show the opposite, despite the theoretically advantageous additional checks of Natural.

An alternative way to understand the answers for questions Q_1 to Q_3, is to analyze the success of each semantics in comparison to Erasure. Figure 8 depicts the results of this analysis. Specifically, the figure shows one Venn diagram per mode of the rational programmer that uses blame. Each diagram shows the overlap of successful scenarios for the blame mode, its corresponding exception mode, and Erasure. For example, the leftmost diagram (Natural) shows that all three modes succeed on 75.7% of the scenarios, only Natural and Natural exceptions succeed on 11.6% of the scenarios, only Natural and Erasure succeed on 1.8%, and Natural alone succeeds on 9.2%. This analysis highlights the success trade-offs each semantics offers against Erasure, with and without blame. For instance, the analysis for Natural clearly illustrates that, when choosing between Natural blame, Natural exceptions, and Erasure, Natural blame is the absolutely most successful: all of the successes of the other two modes are subsets of Natural's successful scenarios. On the other hand, Transient's blame modes fare similarly but the choice is not so clear-cut.

Turning to programmer effort, figure 9 shows the estimated distribution of blame trail lengths for the interesting debugging scenarios. There are two immediate take-aways from the figure. First, the effort for successfully debugging interesting scenarios (in green) for the random mode of the rational programmer is highly spread out, as expected. In contrast, in the other modes, successful effort coalesces at the left side of the plot, meaning that in most cases the programmer needs to type a single module to debug a scenario.

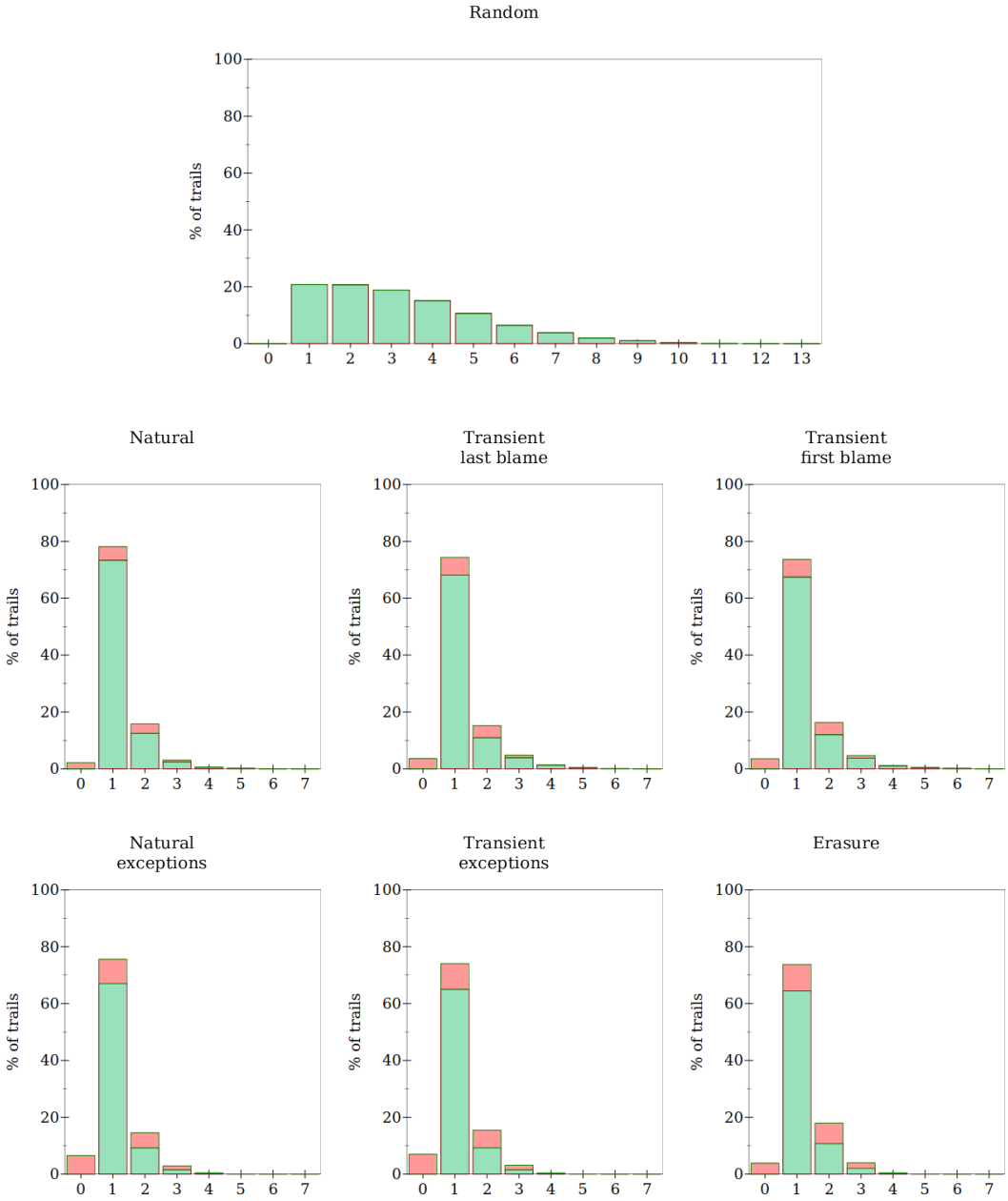
Figure 10 provides head-to-head comparisons of effort. The comparison between two modes boils down to the difference in length between their trails for all scenarios where they both succeed. Hence, each plot in the figure shows the distribution of scenarios with length differences ranging from -3 (the first mode's trail is 3 steps shorter than the second's) to 3 (the first mode's trail is 3 steps longer than the second's). The figure offers several insights about how modes compare in terms of effort that complement the insights about how they compare in terms of success rates from figure 7. First, Natural blame rarely produces shorter trails than Natural exceptions, and occasionally produces slightly longer ones. Hence, the experiment provides evidence that blame helps the rational programmer debug more scenarios but does not shorten the debugging process compared to exceptions. Second, Natural relatively often (close to 8% of the scenarios) produces shorter trails than both Transient blame modes, and sometimes the trails are significantly shorter. Finally, Transient's blame modes share the characteristic with Natural that blame sometimes produces longer trails than their corresponding exception modes.

9 WHAT CAN PROGRAMMERS LEARN

Interpreting the numeric summaries and aggregations of the preceding section demands an intuitive understanding of what blame trails look like in practice. A concrete example of blame trails and programmer modes is a good basis for synthesizing this kind of intuition.

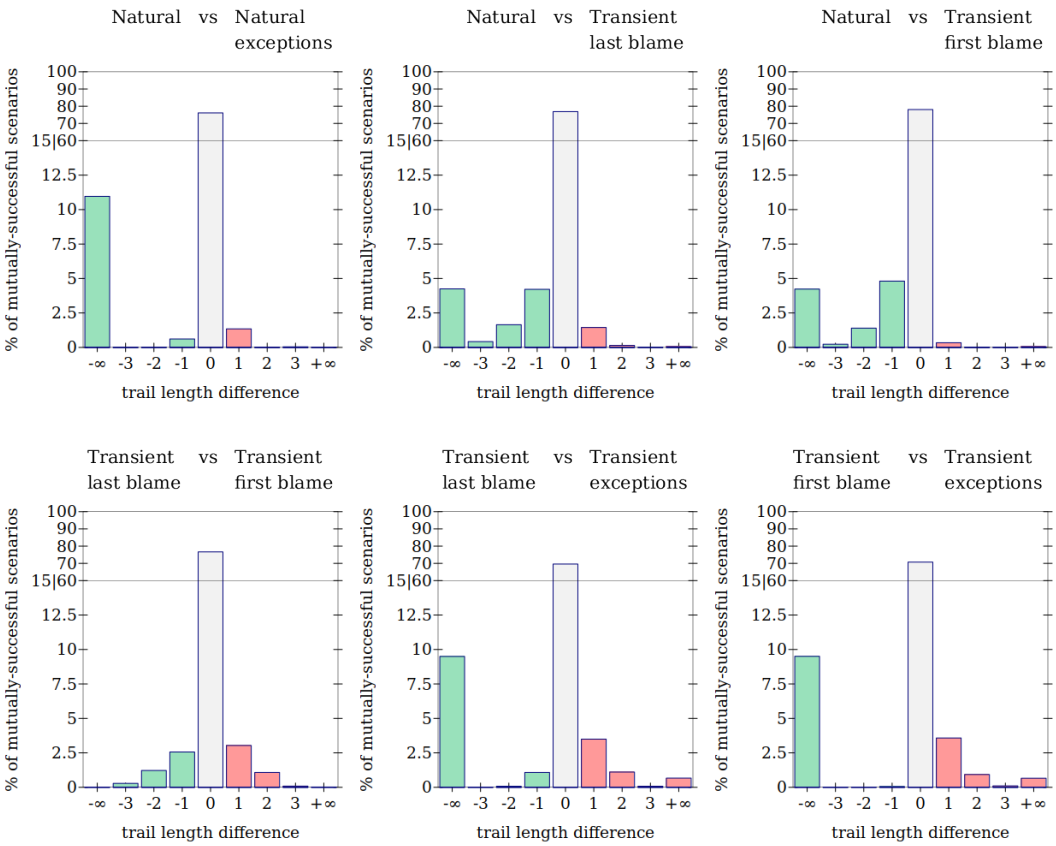
Figure 11 summarizes one particularly interesting debugging scenario from the `take5` benchmark. The module dependency graph of this benchmark is shown in the top left of the figure. Its mutated `player` module provides a method under a different name than the client module, `dealer`, expects. In Typed Racket's gradual type system, this mistake corresponds to a type-impedance mismatch—and all rational-programmer modes come to different conclusions.

The rest of figure 11 illustrates the blame trails for every mode of the rational programmer (except Random) for the debugging scenario in two different ways:



Each plot depicts the distribution of trail lengths for a given mode across all benchmarks. The upper bound margin of error is 0.05%.

Fig. 9. Programmer effort

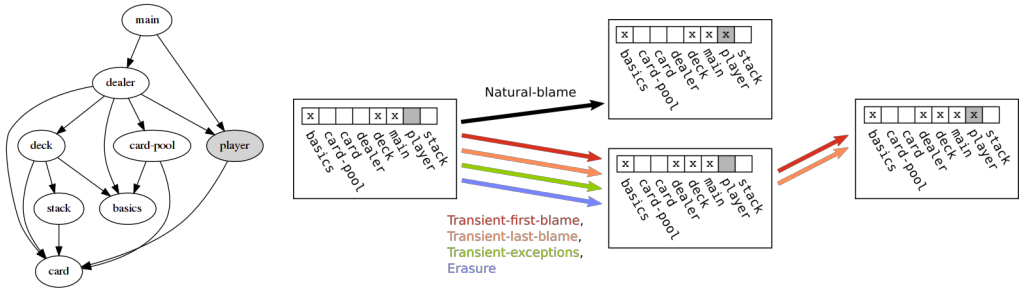


Each plot depicts the distribution of scenarios with trail length differences ranging from -3 to 3. A $-x$ difference denotes that the first mode's trail is x steps *shorter* than the second mode's trail for the same scenario; a positive difference denotes the inverse. A difference of ∞ indicates one mode's trail succeeds while other mode's fails. The 15|60 on the y-axis indicates that the axis is truncated between 15 and 60%. The upper bound margin of error is 0.03%.

Fig. 10. Effort comparisons

- The top right shows the blame trails produced by every mode of the rational programmer as paths through the configuration lattice starting at the root (leftmost) configuration. Each configuration is represented by a sequence of boxes corresponding to modules in the program, with an x indicating that the module is typed. The mutated module has a gray box.
- The table in the middle expands the information in the diagram with the details of every step in each trail. Every row of the table represents the trail of one mode. The middle-three columns depict the steps of a blame trail:
  Root describes the result of running the root configuration in this row's mode.
  Step 1 is the result of the rational programmer's reaction to the outcome of running the root.
  Step 2 shows the result of reacting to the outcome of running step 1 configuration, if any.
  Finally, the **Success?** column summarizes whether exploring the trail succeeds.

To make this table concrete, compare rows 1 and 4. The first one shows that running the root configuration under the Natural-blame mode fails due a dynamic type check and blames the `player`



the dependency graph — the paths taken by each mode through the configuration lattice

Mode	Root config	Root result	Root stack	Step 1 config	Step 1 result	Step 1 stack	Step 2 config	Step 2 result	Success?
Natural -blame	x x x	player	main main	x x x x	$\tau_\times$				✓
Transient -first-blame *and* -last-blame	x x x		dealer dealer dealer main	x x x x	player dealer	dealer	x x x x	$\tau_\times$	✓
Erasure	x x x		dealer dealer dealer main	x x x x		dealer dealer dealer main			X
Natural -exceptions	x x x		main main						X
Transient -exceptions	x x x		dealer dealer dealer main	x x x x		dealer			X

Legend

config Each box corresponds to a module and indicates (with x) if it is typed. The mutated module is gray.

result

symbol	denotation
	the configuration signals a dynamic type check failure, blaming the module(s) below
$\tau_\times$	the configuration does not type check
	the configuration fails a check by the runtime system
	the configuration signals a dynamic type check failure for which blame is ignored

Fig. 11. An example scenario from take5, with every mode's resulting trail.

module; typing that module and running again then results in a type error, and hence the trail is successful. By contrast, the Natural-exceptions mode (row 4) yields stack information for the root configuration that is unhelpful; it identifies only `main`, which is already typed. Hence, this trail immediately gets stuck.

In short, this figure concretely demonstrates how the rational programmer behaves in different modes. In this case, the behaviors differ from each other in five of the six modes (the two Transient blame modes behave the same). The reader may keep the illustration in mind for the following discussion of the numeric results.

9.1 Interpreting the Results

The results of the experiment suggests a number of high-level conclusions about blame strategies in the gradual typed world. First, run-time type checks have a large positive impact, regardless of whether these checks assign blame or throw plain exceptions. Second, error messages with blame assignments are more helpful than those without. The results also indicate, though, that blame is not critical in a majority of cases, and therefore they suggest investigating whether blame tracking is worth the performance cost. Third, the Natural approach fares better than the Transient approach, but only by a small margin. Since Natural offers complete and sound path-based blame while Transient offers incomplete but sound heap-based blame [Greenman et al. 2019a], the results call for a study concerning the relative strengths of the two models of blame. Fourth, given that Transient's sound but shallow run-time type checks do not seem to hamper debugging, a language that supports *both* Natural and Transient might help reduce the number of wrappers and thus address the well-known performance issues of sound gradual typing [Greenman 2020, chapter 6]. Fifth, the fact that both modes of the Transient rational programmer are equally successful suggests that returning the whole blame sequence may not be beneficial. If so, Reticulated Python could limit the size of blame sequences to attempt to mitigate its serious performance problems (see below).

9.2 What Are the Threats to Validity?

The validity of these conclusions is threatened in two distinct ways. The first set of threats concerns aspects of the experimental setup discussed in preceding sections: (i) the representativeness of the benchmarks; (ii) the relation between the mutations and real programming mistakes; and (iii) the sampling strategy. Although the experimental setup attempts to mitigate these threats, the reader must keep these limitations in mind when drawing conclusions.

The second set of threats questions four rather different aspects. To start with, the realism of the rational programmer itself is questionable (sec. 9.3), as is the definition of "interesting scenarios" (sec. 9.4). The remaining threats are about the accuracy and cost of Transient blame, respectively (secs. 9.5 and 9.6).

9.3 Threat: Is the Rational Programmer Realistic?

Like *homo economicus*, which idealizes the actual behavior of a participant in an economy for the sake of mathematical modeling, the model of a rational programmer idealizes the actual debugging behavior of a software developer for the sake of a systematic, large-scale analysis. This idealization comes with advantages and disadvantages. In the economic realm, mathematical models have provided some predictive insights into the market's behavior; but as behavioral economics has shown more recently [Henrich et al. 2001], the mathematical abstraction of a rational actor makes predictions also quite unreliable in some situations.[8] Just like an ordinary consumer or producer, an actual software developer is unlikely to stick to the exact strategy proposed here. When this happens, the predicted benefits of blame assignment may not materialize. Indeed, the authors' personal experience suggests such deviations, and it also suggests that deviating often leads to dead-ends. To make a true judgment of the usefulness of the rational-programmer idea, the community will need much more experience with this form of evaluation and relating the evaluation to the behavior of working programmers.

Relatedly, the experimental setup hides how a rational programmer ascribes types to extend a trail. When the run-time checks signal an impedance mismatch in the real world, a programmer does not have a typed module ready to swap in. Instead, the programmer must come up with the next set of types, which means making choices. It is usually possible to consistently assign

[8] It has misled economists to focus on just mathematics, though this problem is not relevant here—tongue firmly in cheek.

types to variables in a module in several different ways. The maintenance of the benchmarks over many years has driven home this lesson but, fortunately, it has also shown that the types are in practice somewhat canonical. The authors therefore conjecture that different real-world programmers would often come up with equivalent type assignments during debugging sessions.

9.4 Threat: Is the Definition of Interesting Scenarios Reasonable?

Section 6.3 defines criteria for interesting mutations, one of which limits the scenarios under consideration to those with mistakes that raise a run-time error under Erasure. In other words, the experiment is Erasure-biased: it only considers the usefulness of blame when the safety checks of the underlying language alone are sufficient to detect the mistake. In reality, some mistakes require run-time type checks to be detected [Greenman et al. 2019a], and it is possible that blame has more to offer for these kinds of mistakes. If that is the case, then the results of the experiment on a population of scenarios including such mistakes should be different.

In fact, a variation of the experiment provides some preliminary evidence that this difference may be significant. Thus far, the variation of the experiment covers only three of the benchmarks but broadens the selection of scenarios to include any that raise run-time errors under Natural, regardless of other semantics. The results from this small experiment suggest that without Erasure-bias, Natural blame may be much more useful than all of the other modes.

9.5 Threat: Why Does Transient Lose Blame?

The execution of the experiment reveals that Transient produces empty blame sequences for 967 scenarios. An empty blame sequence means a lack of boundary crossings for the witness value. In theory, an empty sequence should not occur, because it means a typed module is blaming itself—something that can happen only if the type checker (or system) is unsound.

An investigation of these empty blame cases reveals problems with tracking blame for higher-order functions and conversely suggests three improvements for the Transient algorithm. To illustrate, consider the call `(filter f xs)`. The first suggestion is that the blame map should know that inputs to `f` may have come from the `xs` list; concretely, the blame-map entry for `f` should point to `xs` as a parent. Second, there should be two parents for `f` instead of one, because both `xs` and `filter` are responsible for sending correct input to `f`. Third, the blame map should work equally well in programs that rename `filter` or that replace the identifier with an expression. This third point suggests a need for type-like specifications that guide the construction of the blame map, instead of the identifier-based matching in Reticulated and Shallow Racket.

9.6 Threat: Is the Transient Blame Assignment Mechanism Realistic?

The results in section 8 also show that the cost of Transient blame is quite high. Under the Transient semantics, some of the debugging scenarios exceed the 4-minute timeout or the 6GB-memory limit. To put those limits into context, the fully typed and fully untyped benchmarks all normally complete in a few seconds with minimal memory usage. Furthermore, none of the mixed Natural configurations hit these limits, and with the blame map turned off, the Transient semantics also runs these programs in a short amount of time and well within the memory limit. In short, even though the Transient rational programmer appears to do well in the experiment, the implementation of the Transient blame strategy might be unrealistic.

At first glance, these measurements seem to contradict the results of Vitousek et al. [2017]. They report an average slowdown of 6.2x and a worst-case slowdown of 17.2x on the fully-typed Python benchmarks in Reticulated Python when the blame map gets enabled. Unfortunately, the average slowdown of 133x and the worst-case slowdown of 560x due to blame in Shallow Racket seems closer to the truth. There are at least three broad factors that skew Vitousek et. al.'s results:

(1) The 2017 implementation of Reticulated fails to insert certain soundness checks[9] and blame-map updates[10] from the paper.
(2) While Reticulated attempts to infer types for local variables, the impoverished nature of its type system does not allow the ascription of precise types and often resorts to type `Dynamic` [Greenman 2020, section 5.4.4]. Code with type `Dynamic` has fewer constraints to check at run-time—and much less information to track in the blame map.
(3) Vitousek et al. [2017] use small benchmarks. Four have since been retired from the official Python benchmark suite because they are too small, unrealistic, and unstable.[11] On the flip side, all the benchmarks in the GTP suite are larger than the official Python benchmarks. Reticulated Python runs the translation of the smallest GTP benchmark in approximately 40 seconds without blame but times out after 10 minutes with blame.

More work on Transient blame is needed to make an informed decision about its prospects as a viable production-level approach.

10 WHAT DOES PRIOR RESEARCH SAY ABOUT THIS PROBLEM

At a philosophical level, Lazarek et al. [2020] present the first empirical analysis *without* involving humans in this general area. While they do not spell out the notion of the rational programmer, they present many of the basic ingredients. At a technical level, the two pieces of work differ in many ways. First the experiment presented here involves three different modes of gradual typing, theirs a single notion of contract checking. Second, this paper also contributes the idea of creating three comparable rational programmers, with several modes. Hence it can answer whether blame adds value to a gradual type system and which gradual type system it benefits the most, while theirs establishes only a systematic relation between blame for contracts and behavioral bugs. Finally, this paper also contributes type-level mutators. As explained in section 6, almost none of Lazarek et al. [2020]'s mutators are useful in the context of gradual typing.

The literature on gradual typing presents many semantics beyond the three used here, and three additional blame strategies. Pyret (pyret.org) assigns fixed-size data types the Natural semantics and functions a Transient semantics. The Forgetful [Castagna and Lanvin 2017] and the Amnesic [Greenman et al. 2019a] semantics are similar to Transient but use wrappers instead of in-lined checks. Nom [Muehlboeck and Tate 2017] and other *concrete* semantics [Rastogi et al. 2015; Richards et al. 2017, 2015; Wrigstad et al. 2010] assume that every value comes with a type tag and use tag checks to supervise the interactions between typed and untyped code. The semantics derived with the Abstracting Gradual Typing technique [Garcia et al. 2016] are variants of Natural. The Monotonic semantics [Kuhlenschmidt et al. 2019; Rastogi et al. 2015; Siek et al. 2015; Swamy et al. 2014] differs from Natural in the treatment of mutable data. It associates every heap location with a type and rejects updates that lower the precision of types. Among these semantics, only Amnesic, Nom, and Monotonic present interesting blame strategies. The experiment excludes the first because it is merely a theoretical construction, the second because it imposes severe restrictions on programmers, and the third because it requires a re-engineering of the Racket runtime.

11 WHAT TO DO NEXT

The interviews of Tunnell Wilson et al. [2018] suggests that programmers prefer the run-time checking of Natural over other soundness methods. But, the opinion of a random set of programmers does not mean that blame assignment adds value. Similarly, researchers and language designers

[9]Missing check: https://github.com/mvitousek/reticulated/issues/36
[10]Missing cast: https://github.com/mvitousek/reticulated/issues/43
[11]Release notes: https://pyperformance.readthedocs.io/changelog.html

have implicitly answered this question one way or another without evidence for the blame-strategy dimension. The experiment presented in this paper provides some justification for the programmers' leanings and helps language creators revisit their decisions. Of course, the design choice remains a trade-off along several dimensions, and the presented experiment sheds light on only one of them.

The paper does *not* address a problem in the gradually typed world that was pointed out early on by practical researchers [Feldthaus and Møller 2014; St-Amour and Toronto 2013; Williams et al. 2017] and that has recently received theoretical attention [Campora and Chen 2020; Greenman et al. 2019a]: mistakes in type annotations themselves. Developers use gradual typing to move an untyped code base into the typed realm, and to this end, they need typed APIs for the vast repositories of already-existing libraries. Instead of converting the libraries themselves, language implementors merely create facade modules that import untyped functions and export them with type annotations, like `typed-pack-lib` in figure 1. With those facades, the compiler can type-check typed modules, but these retroactive additions of types to a library may result from a misunderstanding of the code. *In short, any retroactively ascribed type may thus be a mistake itself.*

The cited evidence suggests that this scenario is quite common and largely unadressed. A future evaluation must develop mutators that produce incorrect type annotations without breaking the code itself. Some preliminary work suggests that such type mutators are even more difficult to develop than the type-level code mutators presented here. Based on the gradual typing literature, the Natural semantics should discover such mistakes. In contrast, the Transient and Erasure semantics cannot help with such mistakes at all; indeed, we expect that these latter two raise misleading exceptions or produce wrong answers without warning. Only additional experimental work can confirm or reject these conjectures.

ACKNOWLEDGMENTS

Felleisen and Greenman were partly supported by NSF grant SHF 1763922. Greenman also received support from NSF grant 2030859 to the CRA for the CIFellows project. Dimoulas and Felleisen wish to thank Max New for extensive discussions of the early ideas in this paper. Thanks also to Robby Findler and Northwestern PLT for their valuable feedback at various stages of this work. This paper was supported in part by the NSF grant CNS-1823244.

REFERENCES

John Peter Campora and Sheng Chen. 2020. Taming Type Annotations in Gradual Typing. *PACMPL* 4, OOPSLA, 191:1–191:30. https://doi.org/10.1145/3428259

Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. *PACMPL* 1, ICFP (2017), 41:1–41:28. https://doi.org/10.1145/3110285

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. https://doi.org/10.1109/C-M.1978.218136

Asger Feldthaus and Anders Møller. 2014. Checking Correctness of TypeScript Interfaces for JavaScript Libraries. In *OOPSLA*. 1–16. https://doi.org/10.1145/2660193.2660215

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *ICFP*. 48–59. https://doi.org/10.1145/581478.581484

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *POPL*. 429–442. https://doi.org/10.1145/2837614.2837670

Rahul Gopinath and Eric Walkingshaw. 2017. How Good Are Your Types? Using Mutation Analysis to Evaluate the Effectiveness of Type Annotations. In *ICSTW*. 122–127. https://doi.org/10.1109/ICSTW.2017.28

Ben Greenman. 2020. *Deep and Shallow Types.* Ph.D. Dissertation. Northeastern University.

Ben Greenman and Matthias Felleisen. 2018. A Spectrum of Type Soundness and Performance. *PACMPL* 2, ICFP (2018), 71:1–71:32. https://doi.org/10.1145/3235045

Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2019a. Complete Monitors for Gradual Types. *PACMPL* 3, OOPSLA (2019), 122:1–122:29. https://doi.org/10.1145/3360548

Ben Greenman, Lukas Lazarek, Christos Dimoulas, and Matthias Felleisen. 2021. A Transient Semantics for Typed Racket. Submitted for review.

Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. 2019b. How to Evaluate the Performance of Gradual Type Systems. *Journal of Functional Programming* 29, e4 (2019), 1–45. https://doi.org/10.1017/S0956796818000217

Joseph Henrich, Robert Boyd, Samuel Bowles, Colin Camerer, Ernst Fehr, Herbert Gintis, and Richard McElreath. 2001. In Search of Homo Economicus: Behavioral Experiments in 15 Small-Scale Societies. *American Economic Review* 91, 2 (2001), 73–78. https://pubs.aeaweb.org/doi/pdf/10.1257/aer.91.2.73

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. https://doi.org/10.1109/TSE.2010.62

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. Toward Efficient Gradual Typing for Structural Types via Coercions. In *PLDI*. 517–532. https://doi.org/10.1145/3325989

Lukas Lazarek, Alexis King, Samanvitha Sundar, Robert B. Findler, and Christos Dimoulas. 2020. Does Blame Shifting Work? *PACMPL* 4, POPL (2020), 65:1–65:29. https://doi.org/10.1145/3373113

Richard J Lipton. 1971. *Fault Diagnosis of Computer Programs.* Technical Report. Carnegie Mellon University, Pittsburgh, PA.

Jacob Matthews and Robert Bruce Findler. 2009. Operational Semantics for Multi-Language Programs. *TOPLAS* 31, 3 (2009), 1–44. https://doi.org/10.1145/1498926.1498930

John Stuart Mill. 1874. *Essays on Some Unsettled Questions of Political Economy.* Longmans, Green, Reader, and Dyer.

Fabian Muehlboeck and Ross Tate. 2017. Sound Gradual Typing is Nominally Alive and Well. *PACMPL* 1, OOPSLA (2017), 56:1–56:30. https://doi.org/10.1145/3133880

Hari Prashanth and Sam Tobin-Hochstadt. 2010. Functional Data Structures for Typed Racket. In *SFP*. 8–14. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.308.8444

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *POPL*. 167–180. https://doi.org/10.1145/2676726.2676971

Gregor Richards, Ellen Arteca, and Alexi Turcotte. 2017. The VM Already Knew That: Leveraging Compile-Time Knowledge to Optimize Gradual Typing. *PACMPL* 1, OOPSLA (2017), 55:1–55:27. https://doi.org/10.1145/3133879

Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete Types for TypeScript. In *ECOOP*. 76–100. https://doi.org/10.4230/LIPIcs.ECOOP.2015.76

Jeremy Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015. Monotonic References for Efficient Gradual Typing. In *ESOP*. 432–456. https://doi.org/10.1007/978-3-662-46669-8_18

Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. 2012. Typing the Numeric Tower. In *PADL*. 289–303. https://doi.org/10.1007/978-3-642-27694-1_21

Vincent St-Amour and Neil Toronto. 2013. Experience Report: Applying Random Testing to a Base Type Environment. In *ICFP*. 351–356. https://doi.org/10.1145/2500365.2500616

Nikhil Swamy, Cédric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. 2014. Gradual Typing Embedded Securely in JavaScript. In *POPL*. 425–437. https://doi.org/10.1145/2535838.2535889

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead? In *POPL*. 456–468. https://doi.org/10.1145/2837614.2837630

Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual Typing for First-Class Classes. In *OOPSLA*. 793–810. https://doi.org/10.1145/2384616.2384674

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: from Scripts to Programs. In *DLS*. 964–974. https://doi.org/10.1145/1176617.1176755

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *POPL*. 395–406. https://doi.org/10.1145/1328438.1328486

Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical Types for Untyped Languages. In *ICFP*. 117–128. https://doi.org/10.1145/1863543.1863561

Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. 2017. Migratory Typing: Ten Years Later. In *SNAPL*. 17:1–17:17. https://doi.org/10.4230/LIPIcs.SNAPL.2017.17

Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. 2018. The Behavior of Gradual Types: a User Study. In *DLS*. 1–12. https://doi.org/10.1145/3276945.3276947

Michael M. Vitousek, Andrew Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *DLS*. 45–56. https://doi.org/10.1145/2661088.2661101

Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. 2019. Optimizing and Evaluating Transient Gradual Typing. In *DLS*. 28–41. https://doi.org/10.1145/3359619.3359742

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 *POPL*. 762–774. https://doi.org/10.1145/3009837.3009849

Philip Wadler and Robert Bruce Findler. 2009. Well-typed Programs Can't Be Blamed. In *ESOP*. 1–15. https://doi.org/10.1007/978-3-642-00590-9_1

Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. 2017. Mixed Messages: Measuring Conformance and Non-Interference in TypeScript. In *ECOOP*. 28:1–28:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.28

Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Ostlund, and Jan Vitek. 2010. Integrating Typed and Untyped Code in a Scripting Language. In *POPL*. 377–388. https://doi.org/10.1145/1706299.1706343