How to Evaluate Blame for Gradual Types, Part 2

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

Equipping an existing programming language with a gradual type system requires two major steps. The first and most visible one in academia is to add a notation for types and a type checking apparatus. The second, highly practical one is to provide a type veneer for the large number of existing untyped libraries; doing so enables typed components to import pieces of functionality and get their uses type-checked, without any changes to the libraries. When programmers create such typed veneers for libraries, they make mistakes that persist and cause trouble. The question is whether the academically investigated run-time checks for gradual type systems assist programmers with debugging such mistakes. This paper provides a first, surprising answer to this question via a rational-programmer investigation: run-time checks alone are typically less helpful than the safety checks of the underlying language. Combining Natural run-time checks with blame, however, provides significantly superior debugging hints.

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. 2023. How to Evaluate Blame for Gradual Types, Part 2. *Proc. ACM Program. Lang.* 7, ICFP, Article 194 (August 2023), 28 pages. https://doi.org/10.1145/3607836

1 GRADUAL TYPES CAN BE AND OFTEN ARE WRONG

TypeScript is the most well-known and widely-used implementation of gradual typing, with over 500k dependent packages on GitHub.[1] It adds a syntax for optional type annotations to JavaScript and a type checker for those annotations. Importantly, TypeScript programs mix seamlessly with JavaScript libraries due to the DefinitelyTyped repository,[2] which supplies crowd-sourced *type interfaces* for thousands of JavaScript libraries. More precisely, the repository contains declaration files for the types of the exports of libraries, which the type checker employs to confirm the (type-)consistency between the main program and its libraries.

Unsurprisingly, the authors of these type interfaces, who are often not the authors of the corresponding libraries, make mistakes. Indeed, academic researchers have published a fair number of results identifying, analyzing, and cataloging these mistakes [Cristiani and Thiemann 2021;

[1] https://github.com/microsoft/TypeScript/network/dependents?dependent_type=PACKAGE
[2] https://github.com/DefinitelyTyped/DefinitelyTyped

Authors' addresses: Lukas Lazarek, PLT @ Northwestern University, Evanston, Illinois, USA, lukas.lazarek@eecs.northwestern.edu; Ben Greenman, PLT @ Brown University, Providence, Rhode Island, 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.

2475-1421/2023/8-ART194
https://doi.org/10.1145/3607836

Feldthaus and Møller 2014; Hoeflich et al. 2022; Kristensen and Møller 2017b; Williams et al. 2017]. The problem is not unique to TypeScript. Typed Racket, a language with a similar type system plus a crowd-sourced set of type interfaces for libraries, suffers from similar mistakes, even in the run-time library [St-Amour and Toronto 2013].

This situation raises a natural question:

> *How well does a gradual type system assist developers with diagnosing errors due to* mistakes in type interfaces *for untyped libraries?*

Formulated this way the question points once again to the differences between industrial uses of gradual types and academic research. While the first insists on erasing types when the program runs, the second has investigated various approaches to run-time checking the boundary between typed and untyped pieces of code.

TypeScript, as an industrial product, erases types and thus does not offer any special support to help programmers in the face of wrong type interfaces. The point is to keep type annotations from interfering with performance, or as the TypeScript website advertises, "TypeScript becomes JavaScript via the delete key" [Microsoft [n. d.]].

By contrast, academic implementations of gradual typing, (e.g., Typed Racket [Tobin-Hochstadt and Felleisen 2006, 2008, 2010; Tobin-Hochstadt et al. 2017] and Reticulated Python [Vitousek et al. 2014, 2019, 2017]) compile types to run-time checks that aim to discover *impedance mismatches* between types imposed on untyped code and the latter's actual behavior. Moreover, when such run-time checking systems catch an impedance mismatch, they *blame* the boundary where a type interface and an untyped value (closure, object, class) are out of sync. The question is whether this blame information offers useful hints, that is, hints that describe the cause of the mismatch and thus assist the developer with the debugging task.

These explanations suggest refinements of the above research question:

(1) How often do the run-time checks of the academic semantics (without blame) help with the diagnosis of mistakes in type interfaces?
(2) How often does blame assignment reduce the debugging work?
(3) Or, how often do the run-time safety checks of the underlying language suffice to sort out such problems?

This paper provides some first answers to these questions.

Answering the questions means (1) investigating a pragmatics concern and (2) doing so with an apples-to-apples comparison. Pragmatics is about the relationship between semantics and its entailed value for a working programmer in a specific context, here, debugging type interfaces in a gradually typed language. For this specific context, we need to use a gradually typed syntax that supports assigning different semantics to the same program.

The answer rests on the pioneering work of Lazarek et al. [2021, 2020], which offers a two-step road map for just such questions. First, it introduces an empirical method for investigating pragmatics, dubbed the *rational programmer*. Roughly speaking, a rational-programmer investigation imagines a programmer as an algorithm that relies on the semantics of the language features it uses to solve a task scenario. In essence, the method simulates different rational programmers, one per semantics, on a large collection of task scenarios, plus an analysis of their respective effectiveness.

Second, besides this investigative framework, Lazarek et al.'s work supplies an open-source platform that enables an apples-to-apples comparison: Typed Racket equipped with the three major semantics of gradual typing [Greenman 2020; Greenman et al. 2022], that is, the academic *Natural* and *Transient*, plus the industrial *Erasure* semantics. Typed Racket is also a good match for this investigation because its type system closely resembles the one of TypeScript (minus type `Dynamic`) meaning the results may apply to this industrial language; see section 7.5.

Technically speaking, the investigation presented herein applies the rational-programmer method to the concrete question:

> *If a language (Typed Racket) comes with several different semantics for the same syntax and type system, is one of them better suited than the others for debugging mistakes in type interfaces?*

Since Lazarek et al. [2021] use the method to investigate the pragmatics of debugging mistakes in *code only*, the new emphasis on type-interface mistakes demands an adaptation of their investigative method. Concretely, the investigation of debugging type interfaces asks for the construction of a new corpus of debugging scenarios tailored to this specific context.

In a nutshell, this paper makes two major contributions:

- At the object level, its results suggest that the checks of Natural combined with its blame mechanism significantly outperform other semantics at debugging type interfaces. In fact, Natural with blame is the *only* one that significantly outperforms the industrial approach. Surprisingly, the investigation reveals that the run-time checks of the academic approaches on their own, including Natural, do not detect significantly more type interface mistakes, and moreover they seem to offer less help with debugging such mistakes than the run-time safety checks of the underlying language. That is, Erasure is a competitive semantics for gradual types when it comes to debugging incorrect type interfaces.
- At the meta level, the design of the rational programmer investigation provides the next piece of evidence concerning the applicability of the method. In particular, the rational programmer investigation of this paper mostly manages to reuse the design of the investigation of Lazarek et al. in a new context by swapping in only one significant component, the corpus of debugging scenarios; see section 4 for additional, minor differences. Our approach to constructing this corpus should offer guidance to other researchers on how to reapply this method to other languages or yet-different contexts.

The remainder of the paper is organized as follows. Section 2 illustrates concretely the landscape of gradual typing checks and error reporting mechanisms, and introduces with examples the context of debugging mistakes in interface types. Section 3 reviews the rational programmer method and instantiates its key ideas in the setting of the paper. Section 4 provides a detailed account of our experimental design while section 5 explains how we overcome the challenge of creating a new suitable corpus of debugging scenarios. Section 6 presents the results of the investigation and section 7 discusses their implications and limitations. The paper concludes with a survey of related work in section 8, and a few closing thoughts in section 9.

2 ONE TYPE INTERFACE MISTAKE, THREE FLAVORS OF GRADUAL TYPING

This rational-programmer investigation uses three different approaches to gradual typing: (1) *Erasure*, which discards types entirely and runs the program as-if untyped; (2) *Natural*, which uses higher order contracts to enforce behavioral properties of types [Findler and Felleisen 2002]; and (3) *Transient*, which rewrites typed code to inline type assertions. Each of these three approaches provide significantly different information to a developer working with an incorrect type interface.

Figure 1 sketches a program that illustrates the differences among the three semantics. The program is organized with a *client-interface-library architecture*: the top third is the client side, the bottom third is the library side, and there is a type interface in the middle. Specifically,

(1) `client/main` (top left) is the untyped entry point of the program. It uses a library to restructure some JSON user data and then summarizes part of the data.
(2) `client/summarize` (top right) is a typed component that implements one helper function, `summarize-ages`. It works with the types that the type interface declares.

(3) `json-unpack-interface` (middle) is the type interface (like those in DefinitelyTyped) that declares types for an untyped library.
(4) `json-unpack-lib` (bottom) is the untyped library.

The type interface mistakenly declares that the result type of `json-unpack` is a list of hash tables. A close look at the library—specifically the purpose statement of `json-unpack-lib`—shows that the function returns a list of associations. The client, however, has been programmed using the interface type because the client programmer has no knowledge about the (possibly large) implementation of the library. That is, `summarize-ages` accesses `user-data` as a list of hash tables.

With the *Erasure semantics*, the impedance mismatch causes the program to crash. A safety check in the runtime fails while applying `hash-ref` in `client/summarize`, and the resulting error informs the developer that `hash-ref` received something other than a hash table. That error also carries a stacktrace to help the developer understand where it happened; it has `client/summarize` at the top, followed by `client/main`. Thus the error information suggests to the developer that there is a problem with the client.

client/main : racket

```
(require json)
(require json-unpack-interface)
(require client/summarize)

;; read user data from files, analyze
(define user-data
  (json-unpack (read-json "profiles")
               "details"))
(define summary
  (summarize-ages user-data))
....
```

client/summarize : typed/racket

```
(provide summarize-ages)

(define-type UserInfo (HashTable String Any))

(: summarize-ages ((Listof UserInfo) -> Integer))
(define (summarize-ages user-data)
  (define ages : (Listof Integer)
    (for/list ([user-info (in-list user-data)])
      (define age (hash-ref user-info "age"))
      (cast age Integer)))
  (apply max ages))
```

json-unpack-interface : typed/racket

```
(require/typed/provide json-unpack-lib
  [json-unpack (JSExpr String ->
                  (Listof (HashTable String JSExpr)))])
```

json-unpack-lib : racket

```
(provide json-unpack)

;; Find JSON objects mapped by `key`, and convert them into
;; association lists
(define (json-unpack a-json key)
  (define selected-objects (find-objects-with-key a-json key))
  (map json-object->assoc selected-objects))

;; json-object->assoc : JSExpr -> (Assoc String Any)
(define (json-object->assoc j) ....)
....
```

The type in `json-unpack-interface` does not match `json-unpack`'s actual type; see comments in the latter.

Fig. 1. One program with an incorrect type interface, three interpretations.

With the *Natural semantics*, the `json-unpack` function from the type interface is wrapped in a contract-proxy that enforces the interface-imposed types with dynamic checks and tracks responsibilities for those types [Tobin-Hochstadt and Felleisen 2008; Tobin-Hochstadt et al. 2017]. Since the function comes from an untyped component, the proxy assigns responsibility for its result type to the boundary between the type interface and `json-unpack-lib`. Analogously, since the function is exported to an untyped component, the proxy assigns responsibility to the boundary with `client/main` for supplying arguments of the appropriate types. Hence, when `json-unpack` returns from its application in `client/main`, the proxy checks that the result is a list of hash tables, which fails, and it blames the type interface/`json-unpack-lib` boundary.

Finally, with the *Transient semantics*, typed code is rewritten to verify the shapes of function arguments and results [Vitousek et al. 2017]. The shape roughly corresponds to the outermost constructor of a value; for example, a `(Listof String)` parameterized type turns into a check that the argument is a list. Compound data is deconstructed via function calls, so the contents of a value have their shape checked as the pieces are extracted. Thus `summarize-ages` is rewritten to assert at the function-entry point that `user-data` is a list and in the loop-entry point to assert that `user-info` is a hash table. The second check fails and blames the boundary between `client/main` and `client/summarize`—suggesting a problem in the client component.

Thus, when a developer debugs the code in figure 1, the chosen semantics matters, because three different semantics deliver three different hints. The following section describes how the rational programmer method can help us understand how the three semantics compare.

3 THE KEY IDEAS OF THE RATIONAL PROGRAMMER INVESTIGATION

In the abstract, the central idea of the rational programmer is gleaned from the long established scientific tradition of studying economic phenomena through so called rational agents [Henrich et al. 2001; Mill 1874]. Similar to an agent that abstracts over the behavior of participants in economic transactions to enable the study of the mechanisms, a rational programmer is a method for studying linguistic mechanisms through an *abstract model* of a programmer. Whereas economic agents are described with game theory and other mathematical tools, a rational programmer is described as an algorithm that prescribes how to use a language feature to *satisfice* [Simon 1947] the demands of a task. Human programmers can use the results of such a study to assess the benefits of rational behavior—that is, to decide whether to act like the algorithm when faced with a similar task.

Here the objective is to track down an incorrect type interface. To this end, the rational programmer exploits feedback from the gradual type system and the error messages it produces in a satisficing manner, modifying the program in the process. The modifications aim to directly identify the incorrect type interface or to make a change that provides new information.

The modification strategy is based on the theory of gradual typing. The central blame theorem of gradual typing states that, assuming types are correct, a blamed component must always be untyped [Tobin-Hochstadt and Felleisen 2006; Wadler and Findler 2009]. Therefore equipping that component with types should either (1) allow the type checker to discover a mismatch between the type interface and the component or (2) result in a blame assignment of some other untyped component. In the first case, the process has uncovered a flaw in the type interface.

Hence the rational programmer adds type annotations to a blamed component; re-runs the resulting program if it type checks; and repeats this process until it obtains a program that does not type check. By testing this process on a large corpus of realistic scenarios, we can collect data about how blame information aligns with the theoretical predictions that inform its design. Those scenarios where blame information translates eventually to a static type error constitute evidence that validate the design rationale behind the semantics; scenarios where the rational programmer

does not obtain useful hints from blame about how to further modify the program form examples where blame information does not live up to its intended role.

To make this discussion concrete, take a second look at the program in figure 1. Under the Natural semantics, the program terminates with this error:

```
json-unpack: broke its own contract
  promised: hash?
  produced: '(("age" . 42) ....)
  in: an element of
      the range of
      (-> any/c any/c (listof (and/c hash? ....)))
  contract from: (interface for json-unpack)
  blaming: (interface for json-unpack)
   (assuming the contract is correct)
  at: json-unpack-interface
```

The key to deciphering the error message is the phrase "interface for json-unpack." It says that the Natural semantics has discovered an impedance mismatch between the untyped `json-unpack` and the declaration of its type in `json-unpack-interface`, the type interface of `json-unpack-lib`.

The information clearly identifies the problem: `json-unpack`'s result type doesn't match the values it actually returns. A human programmer may deduce that the type in `json-unpack-interface` must be wrong, for instance by knowing that the underlying, untyped library has been in use for a long time. To analyze the issue and conclude for sure what is going on, the programmer may attempt to construct the function's correct type from the source of `json-unpack-lib` or submit a bug report to the developers of the type interface.

The rational programmer indirectly simulates this process. Specifically, it acts on the phrase "assuming the contract is correct" in the blame assignment issued by the Natural semantics and temporarily gives the interface the benefit of the doubt. That is, the rational programmer assigns blame to `json-unpack-lib`. In response, it adds types to this library, mimicking a programmer that attempts to provide a type interface for the library. Of course, equipping `json-unpack-lib` with type annotations allows the type checker to statically identify the mismatch between the correct type in `json-unpack-lib` and the incorrect one in the interface.

If the rational programmer were to use the Transient semantics instead, the process would follow the same pattern, only using Transient's flavor of blame instead. With Transient, the original program terminates with blame on the boundary between `client/main` and `client/summarize`. The rational programmer therefore equips `client/main`, the untyped of the two, with types and runs the resulting program. That, in turn, terminates with blame on the boundary between `json-unpack-lib` and the type interface. Again the rational programmer gives the interface the benefit of the doubt, annotates `json-unpack-lib`, and reaches a type error.

The rational programmer using Erasure is out of luck. The Erasure semantics exclusively relies on the safety checks and failure messages of the untyped language, mostly stacktraces. The rational programmer must therefore interpret the trace as blame assignment, which we operationalize by selecting the topmost untyped module to annotate. The Erasure semantics of the original program is a stack identifying first `client/summarize` and then `client/main`. Equipping `client/main` with types does not produce a type error, and since the types are simply erased, the additional annotations do not change the exception information. In short, the rational programmer using Erasure is stuck at this point.

An Experiment Sketch. The rational programmer represents a systematic process for chasing down an impedance mismatch, which can be implemented and tested on realistic scenarios. With a large number of such *debugging scenarios*, the rational programmer's aggregate results offer a

big picture view of the helpfulness of the blame information for locating impedance mismatches. Furthermore, by performing the same process on the same scenario for each of the three semantics, we can compare how often and under what circumstances one approach is better than another. The large scale collection of this data constitutes a rational programmer experiment.

The results of such an experiment, i.e. how well the rational programmer is able to locate bugs in real programs, offer insight into the validity of the design rationale underlying a gradually-typed semantics. In particular they help to understand if these tools offer practical benefits in accomplishing a task when used in a systematic way. Of course, human programmers do not always use tools in systematic ways. Hence, this kind of investigation shows how well "tools" work in real programs, not how people work with tools.[3]

Realizing the experiment requires overcoming two difficulties:

(1) The first is the adaption of the experimental framework of Lazarek et al. [2021] to this new setting of incorrect type interfaces. A foundational assumption of that prior work is that types do not contain mistakes, but the premise here is that such mistakes are not only possible but common. As it turns out, the high-level *protocol* of Lazarek et al. can be reused as-is with only a few adjustments (sec. 4).
(2) The second concerns the component of Lazarek et al.'s setup that cannot be reused. Studying mistakes in type interfaces requires the construction of a new, large and diverse corpus of interesting debugging scenarios. Summing up the intuition from section 2, interesting debugging scenarios have a client-interface-library architecture, where the type interface describes an incorrect type for some export(s) of the library, and the client is programmed to that type interface. No collection of such programs is available; the only option is to construct one, which is a challenging task demanding novel insights (sec. 5).

4 FROM THE KEY IDEAS TO A RATIONAL PROGRAMMER EXPERIMENT

An implementation of the rational programmer experiment requires the precise definition of how the rational programmer reacts to error messages. In essence, section 3 sketches a rational programmer that uses run-time type-error information to migrate a program component-by-component in order to debug it. However, despite recent progress [Campora et al. 2017; Garcia and Cimini 2015; Kristensen and Møller 2017a; Migeed and Palsberg 2019; Miyazaki et al. 2019; Phipps-Costin et al. 2021; Rastogi et al. 2012], gradual type migration remains largely an open problem. Following Lazarek et al. [2021], we circumvent this problem by pre-constructing the type migration lattices [Takikawa et al. 2016] for the program corpus of the experiment. Hence, with the help of this domain knowledge, the rational programmer becomes a search algorithm that starts from a flawed mix-typed version of a program, dubbed a *debugging scenario*, and attempts to find a path, dubbed a *blame trail*, through the program's migration lattice to a version that identifies the cause of the problem.

Following the sketch of section 2, we define several *modes* of the rational programmer. Each mode acts upon different sources of error information, so they may chart different blame trails for the same debugging scenario. Collecting and analyzing the differences between these trails for a large number of scenarios is the central idea of the experiment. The remainder of the section makes the experimental idea precise. First, we define two key terms: (1) the migration lattice of a program and (2) a debugging scenario (sec. 4.1). Second, we characterize the search for blame trails according to each mode (secs. 4.2 to 4.4). With these definitions in hand, we formalize the experimental questions from section 1 and the experimental procedure to answer them (sec. 4.5).

[3]Understanding the interactions of programmers with tools in the wild demands a separate study involving people.

4.1 Migration Lattices and Debugging Scenarios

From the perspective of our rational programmer experiment, a program P is a set of components. Intuitively, some of these components implement a library that the rest use. Independent of their role, some of the components are untyped, i.e., they do not have type annotations, and some are typed. In particular, one of the typed components, $\mathcal{I}$, plays the role of the (wrong) type interface between the library and its clients as described in sections 1 and 2.

The typed components of a program P, excluding $\mathcal{I}$, constitute a configuration s of P; conversely, a configuration s determines a syntactic variant of P with some typed and some untyped components. The configurations of P are ordered by the subset relation and form a lattice $\mathcal{L}[\![P]\!]$ with $2^{|P|-1}$ elements—the *migration lattice*. The bottom of $\mathcal{L}[\![P]\!]$ is the empty set, the top one consists of typed versions of all components in P (except $\mathcal{I}$). The configurations in between these two extremes determine the mixed-typed variants of P.

In the context of $\mathcal{L}[\![P]\!]$, a blame trail is simply an ascending chain of configurations of P starting at a debugging scenario. Any configuration s_0 of $\mathcal{L}[\![P]\!]$ can be a debugging scenario. If running a scenario s_i detects an impedance mismatch, i.e., a run-time type error, the rational programmer uses the error information to decide which component to equip with types next. This choice shifts the attention of the rational programmer to a scenario s_{i+1}, and extends the blame trail.

While extending the trail, the rational programmer eventually encounters a scenario s_n that is the end of the trail. There are three such cases:

(1) When the rational programmer reaches a scenario s_n where the type checker rejects the program, the rational programmer has managed to identify the source of the impedance mismatch. The trail ends in success. See section 3 for an example.
(2) Due to the actual implementation of the experiment, the rational programmer may succeed in a different way. Namely, running s_n may terminate with a run-time type error that identifies a boundary between the type interface $\mathcal{I}$ and itself. This situation may arise because the implementation realizes type interfaces as three modules: two typed ones surrounding an untyped adapter module. Section 5.3 explains this design and its rationale in detail.
(3) When the trail ends because the run-time error from s_n does not identify one of the untyped components of P (nor the components of $\mathcal{I}$), the rational programmer has failed. In essence, the trail goes cold and provides no further hints about how to migrate the program in order to get additional information about the impedance mismatch.

4.2 The Natural Rational Programmer

The Natural semantics produces two kinds of run-time error information: blame information due to a failed run-time type check, and stacktrace information due to a raised exception. The two are not equivalent in a strict sense. On one hand, Natural's blame information identifies at most one boundary between a typed and an untyped component, and implies that the untyped side of the boundary is responsible for an impedance mismatch. This information is a direct hint to the rational programmer about how to extend the blame trail: swap out the untyped component with its typed counterpart and obtain a new configuration. On the other hand, the stacktrace information points to all the components that happen to be on the call stack of the underlying language when an exception is raised. In the absence of blame information, the Natural rational programmer can fall back on the stacktrace information and interpret it as blame information in order to continue the debugging session: swap out the first untyped component in the stacktrace with its typed counterpart and obtain a new configuration.

Mode definition: Natural blame
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*

$$s_{i+1} \setminus s_i = \begin{cases} \{blame[\![P, s_i]\!]\} & \text{if (the program for) } s_i \text{ produces blame} \\ \{exception_{Natural}[\![P, s_i]\!]\} & \text{otherwise} \end{cases}$$

where

(1) $blame[\![P, s]\!]$ *denotes the component (of* P*) that* s *blames under the Natural semantics, and*
(2) $exception_{Natural}[\![P, s]\!]$ *denotes the first untyped component in the stacktrace produced by* s *under the Natural semantics.*

As the Natural rational programmer extends a blame trail it may encounter a scenario that does not type-check or blames $\mathcal{I}$ in P. Both situations mean that the rational programmer has located the source of the bug in P. The blame trail ends in success. In contrast, a Natural blame trail ends in failure if the rational programmer reaches a scenario that does not reveal the bug statically, yet its terminating exception also does not point to an untyped module (either as blame information or as part of the stacktrace information). Thus the rational programmer has no further hints on how to continue the search for the bug.

A Natural blame trail $s_0, \ldots s_n$ *in a lattice* $\mathcal{L}[\![P]\!]$ *is* successful *iff* $error[\![P, s_n]\!] \equiv \mathcal{I}$ *or (the program for)* s_n *does not type check,*
where $error[\![P, s_n]\!]$ *is the component identified either by blame or exception information produced by* s_n *under the Natural semantics.*

A Natural blame trail $s_0, .., s_n$ *in a lattice* $\mathcal{L}[\![P]\!]$ *is* failing *iff* s_n *type checks and the trail cannot be extended further.*

Since every exception results in a stacktrace, including those from failed run-time type checks, a mode of the rational programmer that ignores blame and always extends its trail based on stacktrace information can serve as the baseline for determining the actual role of blame in the successes of the Natural rational programmer.

Mode definition: Natural 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* $s_{i+1} \setminus s_i = \{exception_{Natural}[\![P, s_i]\!]\}$.

With this baseline, the usefulness of Natural blame boils down to the comparison between Natural blame trails and Natural exception trails that start at the same scenario s_0.

Given a program P *and a debugging scenario* 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.*

4.3 The Transient Rational Programmer

Formulating the definition of a Transient blame trail is more complex than that for Natural blame trails. The Transient semantics assigns blame to a sequence of components instead of one side of a single boundary. Specifically, the Transient blame sequence, denoted $multiblame[\![P, s]\!]$, says that the value-witness of the impedance mismatch may have crossed the boundaries between neighboring components in the sequence, and that the same run-time type check could have detected the mismatch upon each crossing.

To resolve this ambiguity, we pick two different ways of interpreting blame sequences as blame information. The first choice is that the rational programmer isolates the untyped component added

to the blame sequence first. The second isolates the component added last. The intuition for the two choices is that the first aims to find the earliest point in the evaluation of a program that could have detected the impedance mismatch, while the second interprets the blame sequence as a stack. Besides these two different interpretations of blame sequences, the definition of Transient blame trails largely follows that for Natural blame trails.

Mode definition: Transient first blame
A Transient-first blame trail *is a sequence of scenarios* $s_0, \ldots s_n$ *of* P *such that for all* $0 \leq i \leq n-1$, $s_i \subset s_{i+1}$ *and*

$$s_{i+1} \setminus s_i = \begin{cases} \{first[\![multiblame[\![P, s_i]\!]]\!]\} & \text{if } s_i \text{ produces blame} \\ \{exception_{Transient}[\![P, s_i]\!]\} & \text{otherwise} \end{cases}$$

where

(1) $first[\![multiblame[\![P, s]\!]]\!]$ *is the first untyped module that Transient adds to the blame sequence for* s *under the Transient semantics, and*
(2) $exception_{Transient}[\![P, s]\!]$ *denotes the first untyped component in the stacktrace produced by* s *under the Transient semantics.*

Mode definition: Transient last blame
A Transient-last blame trail *is analogous to a* Transient-first blame trail*, but selects the last untyped module from* $multiblame[\![P, s_i]\!]$ *that Transient adds to the blame sequence rather than the first.*

Also similar to the Natural rational programmer, a *Transient exception trail* can serve as a baseline for isolating the usefulness of Transient-first and Transient-last blame. The definitions of Transient exception trails and the usefulness of the two interpretations of Transient blame are analogous to those formulated for Natural.

Mode definition: Transient exceptions
A Transient exception trail *is analogous to a* Natural exception trail*, but using the Transient semantics rather than Natural.*

4.4 The Erasure Rational Programmer

In contrast to the Natural and Transient semantics, the Erasure semantics produces no blame information. The only kind of error information from Erasure is a stacktrace, so the Erasure rational programmer has a single exception mode.

Mode definition: Erasure
An Erasure trail *is analogous to a* Natural exception trail*, but using the Erasure semantics rather than Natural.*

4.5 The Experimental Questions

The definitions of when blame is useful in the context of the different modes of the rational programmer offers the vocabulary for stating the research question from section 1 in precise terms:

Q_1 Is blame information useful in the context of Natural for type interface mistakes?
Q_2 Is first-blame useful in the context of Transient for type interface mistakes?
Q_3 Is last-blame useful in the context of Transient for type interface mistakes?
Q_* Is blame in the context of X more useful than blame in the context of Y for type interface mistakes (where X, Y in [Natural, Transient, Erasure])?

Answering Q_1 requires comparing the success of Natural blame and Natural exception trails for all debugging scenarios. If there are any scenarios for which the Natural blame trail succeeds but

the Natural exception trail does not, then Q_1 has a positive answer. Those scenarios are evidence that Natural's blame information improves the debugging effectiveness of the rational programmer over the information available from just Natural's stacktrace information. That said, if there are no such scenarios, Natural blame may still be useful if it reduces the number of modules the rational programmer needs to inspect in order to locate a bug. That number is equal to the length of the blame trail, so comparing the length of Natural-blame and Natural-exceptions trails provides a secondary metric of utility. Along those lines, we can also compare the length of both mode's trails to those of a mode that randomly types modules to understand in some absolute sense if the information from Natural translates to shorter trails than selection by chance. The answers to Q_2 and Q_3 are similar, using the corresponding modes.

Q_* requires comparing the proportion of scenarios where one mode fares better than the other, and the inverse. Better here means that, for the same debugging scenario, one mode's trail succeeds and the other's does not. For example, to determine if the Natural blame mode fares better that Erasure mode, we compare the percentage of Natural blame trails that succeed but the corresponding Erasure trail does not, and vice versa. These two proportions may be similar, in which case the answer to Q_* may not be clear cut; trail length may again offer more information to understand the trade offs between the two modes in that case.

Thus the process to answer the experimental questions boils down to the following plan:

(1) Create a large and diverse corpus of debugging scenarios;
(2) Collect the blame trails for each mode of the rational programmer;
(3) Compare the successes and failures of each mode's blame trails.

5 THE CHALLENGE OF A LARGE AND DIVERSE CORPUS OF SCENARIOS

While there are plenty of wrong type interfaces for untyped libraries in the wild, they are not a suitable basis for a corpus of debugging scenarios. In addition to a library and a wrong type interface, a debugging scenario consists of clients that interact with the library as if the type interface were correct and in such a way that the impedance mismatch manifests itself. However, no curated collection of such buggy programs with client-interface-library architecture exists.

To create a corpus of such debugging scenarios, we proceed in four steps. First, we identify a diverse set of fully-typed correct Racket programs as the seed for the scenario corpus (section 5.1). These programs can be naturally split into components that implement a library, a thin component that plays the role of the library's type interface, and the library's clients that interact with the library through the interface. This architecture matches the needs of our experimental design. Second, we mutate each seed program to inject mistakes into its type interface. Historically, though, mutation analysis does not provide mutators for types. We therefore invent type mutators and validate their effectiveness (section 5.2). Third, we add dynamic adaptors to each mutated program so that client components interact with the program's library according to the mutated type interface rather than the original one (section 5.3). Importantly, all these adapted mutants have the same migration lattices because they all share the same type-able components. Moreover, these lattices can be computed in a straightforward manner from the type annotations of their corresponding fully-typed seed program. Finally, we sample the extensive space of generated debugging scenarios to obtain a sufficiently large and diverse but computationally feasible corpus for the rational programmer experiment (section 5.4).

5.1 The Seed of the Scenario Corpus

Our starting point is Greenman et al. [2019b]'s GTP collection of Typed Racket programs. Originally created as a benchmark suite for evaluating the performance of gradual typing with Typed Racket,

GTP contains fully typed, correct programs authored by various people for various practical purposes. The programs vary in size, style, and complexity; they employ a variety of Typed Racket features; and they are all deterministic. For details, see Greenman et al. [2019b]. Finally, each program in GTP comes with interchangeable typed and untyped versions of every component, which renders the construction of its migration lattice straightforward.

Table 1. Summary of the seed GTP programs.

name	description	author	loc	mod.
`acquire`	object-oriented board game implementation	M. Felleisen	2332	9
`gregor`	utilities for calendar dates	J. Zeppieri	2641	13
`kcfa`	functional implementation of 2CFA for λ calculus	M. Might	406	7
`quadT`	converter from S-expression source code to PDF	M. Butterick	7813	14
`quadU`	converter from S-expression source code to PDF	B. Greenman	7558	14
`snake`	functional implementation of the Snake game	D. Van Horn	305	8
`synth`	converter of notes and drum beats to WAV	V. St-Amour	1060	10
`take5`	mixin-based card game simulator	M. Felleisen	761	8
`tetris`	functional implementation of Tetris	D. Van Horn	445	9
`suffixtree`	algorithm for longest common subsequences between strings	D. Yoo	718	6

Table 1 provides an overview of the ten GTP programs chosen to serve as generators of debugging scenarios. These are the programs in the suite with the densest dependency graphs. We exclude programs with sparse dependency graphs because they exhibit limited run-time interactions between their components and thus result in simplistic debugging scenarios.

In their existing state, the ten chosen programs are not suitable for generating debugging scenarios. Specifically, they lack dichotomous client and library sides, with a type interface component between those. It is easy, however, to identify library and client portions in all of them and to modify them to consolidate the connections between the two in a new type interface component.

While a simple modification of the GTP programs thus suffices to obtain programs with a *client-interface-library architecture*, many of the resulting type interfaces lack a key feature of interesting type declarations. In particular, they cannot include data structure definitions, i.e., Racket's `structs`. The reason is that `structs` in Racket are by default generative. Hence, on one hand, the type interface cannot be their definition site because typically library components depend on the data type too, not just the library's clients. On the other hand, due to generativity, the type interface cannot duplicate the data type definitions. In sum, as Greenman et al. [2019b] describe, data types used by multiple components must reside in a so-called adaptor module. Greenman's adaptor modules make it impossible, however, to mutate the data type definitions, a kind of mutation that is an essential ingredient for the generation of non-trivial debugging scenarios. With this mutation, the library equipped with a type interface and its client components get different views of the same data type.

Fortunately, Racket offers a work-around that is applicable to most of the chosen programs.[4] The key is to change all `structs` to so called pre-fabricated `structs`. These are non-generative data types which are equivalent to any other pre-fabricated data type with the same structure. In other words, a pre-fabricated `struct` allows every component that uses instances of the data type to re-declare its type definition. Thus, the type interface can also contain definitions for the data types, which opens up their mutation for the creation of incorrect views for client components.

[4]Unfortunately, this change is not feasible for the `acquire`, `kcfa`, and `suffixtree` programs; contracts generated by Typed Racket for prefabricated `structs` result in impractical slowdowns for those programs. Hence, we use adaptor modules and do not mutate their data type definitions.

5.2 Mutating Interface Types

With suitable seed programs in hand, we use mutation to transform them into debugging scenarios. Recall that in a debugging scenario, the type interface ascribes an incorrect type to some value(s) that cross from the library to the client components. Therefore turning the seed programs into debugging scenarios requires mutating type annotations in their type interfaces.

Standard mutation operators are useless for this purpose, for they mutate code rather than types. Instead, we develop a new set of operators targeting the language of types. The goal of these new operators, listed in table 2, is to make small syntactic changes to a type interface so as to create an inconsistency between the mutated interface and the actual types of library components. The operators' design draws inspiration both from the authors' own experience in making mistakes in type specifications and their observations about mistakes students make in a variety of programming-oriented courses.

Table 2. Summary of mutators.

name	description	example
`base->Any`	swaps a base type with `Any`	`Integer` ⟶ `Any`
`composite->Any`	swaps a composite type with `Any`	`(List Player)` ⟶ `Any`
`arg-swap`	swaps two of a function's (or method's) argument types	`(A B C -> D)` ⟶ `(C B A -> D)`
`result-swap`	swaps two of a function's (or method's) result types	`(A -> (Values B C D))` ⟶ `(A -> (Values C B D))`
`struct-swap`	swaps two of a struct's field types	`(struct pair ([id : Natural] [content : String]))` ⟶ `(struct pair ([id : String] [content : Natural]))`
`class-swap`	swaps two of a class's field types	`(Class (field [id : Natural] [content : String]))` ⟶ `(Class (field [id : String] [content : Natural]))`

Table 2 presents the mutators:

- The first two capture the generic situation where the programmer has accidentally used the wrong type in some place, for example, ascribing `(Integer -> String)` to a function from `Integer` to `Integer`. Rather than arbitrarily picking an alternative type, these mutators use the type `Any` to generically represent some other (incompatible) type than the one originally in the same place at the interface.
- The second pair, `arg-swap` and `result-swap`, correspond to the specific mistake when the programmer forgets the proper order of positional arguments or results of a function and thus puts the types in the wrong order.[5]
- The last two swap fields in a structure type or class type definition.

The question is

whether these mutators create suitable mutants.

The answer has two distinct dimensions. The first is a philosophical dimension. It questions how these mutators correspond to the mistakes programmers actually make or encounter in type interfaces. The second is a technical one, namely, whether the mutators create variants of GTP

[5]In Racket, expressions may produce multiple values. Typed Racket's type language therefore supports describing the types of each result in function types.

programs whose type interfaces ascribe the wrong type to values such that a program-run signals a type-related exception.

Along the first dimension, these mutators effectively simulate the kinds of mistakes that, according to recent work by Hoeflich et al. [2022] and Williams et al. [2017], actually appear in DefinitelyTyped type interfaces. For instance, Hoeflich et al. [2022] found that one of the most common mistakes is the misspelling of field names in record types. This mistake means that clients attempt to access a non-existent field, only to find out that it is missing. In JavaScript this failed access results in `undefined`, a special and useless placeholder. The `base->Any` and `composite->Any` mutators simulate this scenario as they transform field types, thereby rendering the field unusable by client components. Similarly, the typical off-by-one function arity mistake is simulated by transforming the last argument of a function's type to the opaque `Any` type, because in JavaScript missing arguments are filled in with `undefined` opaque values.

Along the second dimension, it is necessary to run the mutants produced by the mutators in order to understand their quality. Unsurprisingly, our mutators do not always create type interfaces that cause impedance mismatches. For example, replacing a type with `Any` typically causes a type error, because `Any` is the top type encompassing all types, but not always. Fortunately, the GTP benchmarks make it easy to check whether a particular mutant is ill-typed. Specifically, if the mutation introduces an impedance mismatch, the type checker signals a static type error for the top `configuration` of the migration lattice for the mutant. This type error identifies the mismatch between the interface and the corresponding library component.

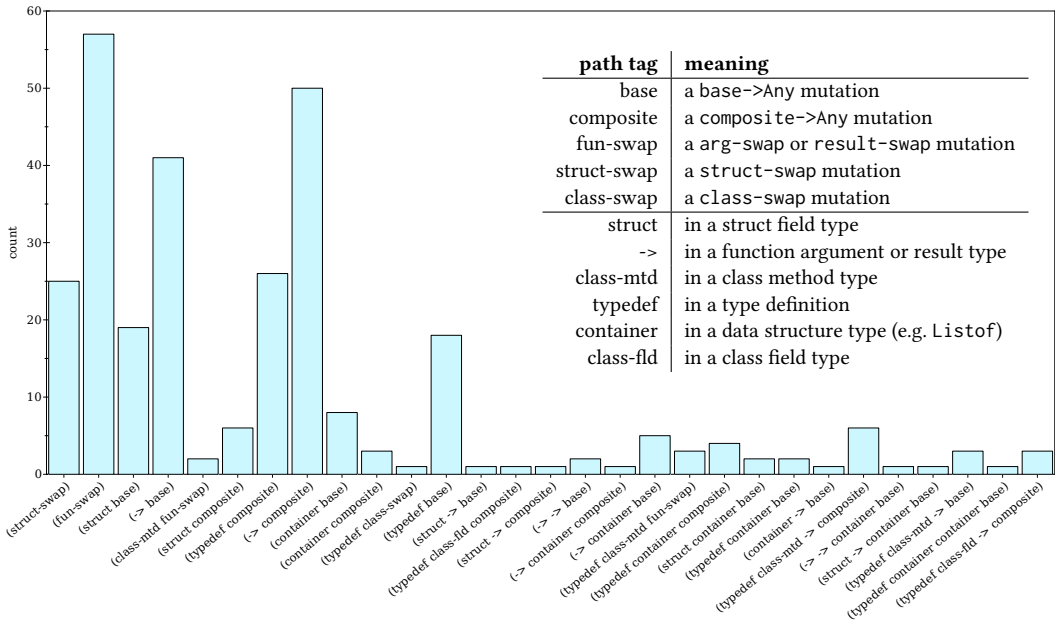
Once an ill-typed mutant is identified, the next step is to confirm its suitability as a source of debugging scenarios. An impedance mismatch alone may not change the run-time behavior of a program. For instance, the mismatch may be in the type of a function that is never used. The Natural semantics provides an appropriate filter for such mismatches. Since Natural is a complete monitor and signals strictly more errors than Transient or Erasure [Greenman et al. 2019a], it guarantees to signal an error if the impedance mismatch affects the program's behavior. We therefore further select only those mutants for which the Natural semantics signals an error in the bottom `configuration` of the migration lattice. After all, any error arising while running this configuration must be due to a contract resulting from the type interface, because those types are the only ones enforced. That said, this choice introduces a small degree of bias against the other semantics, which sections 6 and 7.1 quantify and discuss.

All told, the mutators create just under one thousand mutants from the ten selected GTP programs. Of those around 400 are ill-typed, and 294 have observable changes in dynamic behavior. Hence we end up with 294 suitable mutants for the rational programmer experiment.

Figure 2 illustrates that these mutants form a diverse population, capturing a wide array of mistakes in many different shapes of types. Each bar depicts the number of mutants where the mutation falls into the category named on the x-axis. The categories correspond to a path down the spine of the mutated type, from the outermost level down to the location and specific change introduced. For example, the category `(-> struct base)` collects mutations that change the base type (to `Any`) of a struct field which is the argument or result of a function type.

5.3 Adapting Mutants to Debugging Scenarios

Mutating the type interface of the GTP programs alone does not suffice to create interesting debugging scenarios. In particular, since the programs are correct with respect to the types of the original interface, mutating the interface creates a disconnect between the client components and the types described by the mutated interface. Figure 3 illustrates the problem with a simplistic example. While the interface has been mutated to swap the argument types of `f`, the client uses `f` according to its original type. An interesting debugging scenario requires, however, that the client's



path tag	meaning
base	a `base->Any` mutation
composite	a `composite->Any` mutation
fun-swap	a `arg-swap` or `result-swap` mutation
struct-swap	a `struct-swap` mutation
class-swap	a `class-swap` mutation
struct	in a struct field type
->	in a function argument or result type
class-mtd	in a class method type
typedef	in a type definition
container	in a data structure type (e.g. `Listof`)
class-fld	in a class field type

Fig. 2. Type mistakes captured by final mutant population.

code aligns with the mutated interface types. Fixing this mutation-induced discrepancy represents the key technical challenge for the design of our rational programmer experiment.

client : typed/racket

```
(require ti)

(f #\c 5)
```

ti : typed/racket

```
(require/typed/provide lib
  ;; arguments are swapped
  [f (Integer Char -> Integer)])
```

lib : racket

```
(provide f)
(define (f ch n)
  (+ (char->integer ch) n))
```

Fig. 3. A simple program illustrating the need for adaptors.

We solve this challenge with the introduction of *mutation adaptors*. Conceptually, a mutation adaptor adjusts the client to align with the mutated type interface. From an architectural perspective, it is an interposition layer between the mutated interface and the client side of the program, and it consists of a typed and untyped part:

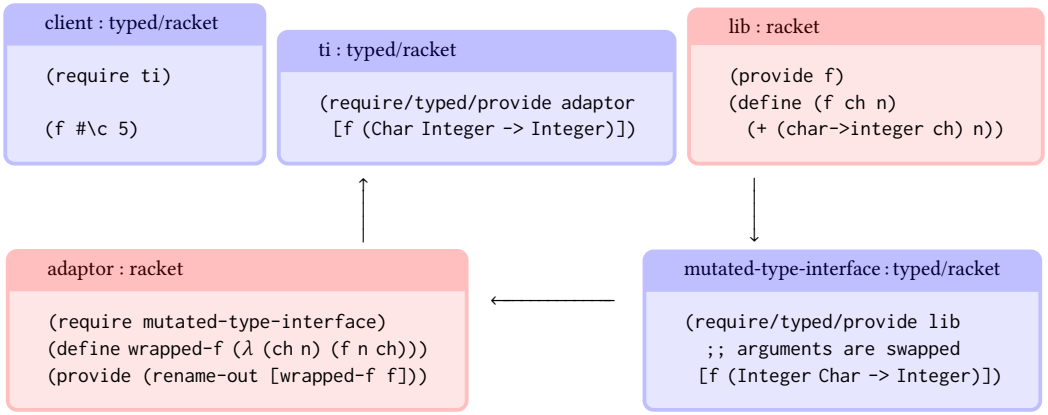
(1) a variant of the original type interface, with which the client interacts.
This variant imports the adaptor module and re-exports all elements at the original and correct type. It thus decouples the clients from the mutated interface. Specifically, this type interface ensures that all client components type-check according to their existing type annotations from the GTP benchmark suite.

(2) the flow adaptor, an untyped module between the original interface and the mutated one.
The flow adaptor adjusts the flow of values at run time from the original type signatures to the mutated ones, respectively. It simulates a client that uses an exported value according to the mutated type.

> The behavior of flow adaptors is specific to the mutation of the type interface. For swapping mutators, the flow adaptor swaps the values of concern. For mutators that replace types with `Any`, the adaptors replace the corresponding value with an opaque sealed value to represent a value of unexpected type.

Concerning the experiment, the faulty type interface in a debugging scenario therefore consists of three modules: the mutated type interface, the flow adaptor, and a variant of the original type interface.

Let us return to the example in figure 3. Adapting this program requires the injection of: a function that swaps the `Char` and the `Integer` argument to adjust the flow of values; the mutated type interface; and the modification of the original type interface to import values from the flow adaptor. The diagram in figure 4 represents the result of these program modifications.



The adaptor creates the wrapper function `wrapped-f` and exports it as `f`, so that the rest of the components are oblivious to the wrapper. The adaptor and the two interfaces (in blue) form the actual interface of the library in the debugging scenario for the purposes of the experiment.

Fig. 4. Adapting the program of figure 3.

While adaptors for the swapping mutators have a fairly obvious rationale, those for `base->Any` and `composite->Any` demand some explanation. The point of replacing a type with `Any` is to hand the library a value of a completely unknown and unexpected type. The existing library code cannot deal with such a value, and it signals an error when it uses any elimination operations on such a value. The flow adaptors therefore simulate this situation by placing the value of the mutated type in an opaque container, ensuring that the library is unable to inspect or use it in any way.

Implementing these program modifications—specifically the flow adaptors—is mostly straightforward. The experiment framework generates flow adaptors that boil down to either swapping or sealing. Technically speaking, the framework exploits Racket's support for interposition through impersonators and chaperones [Strickland et al. 2012] or makes adapted copies of values. The one exception to this approach are mutations that swap class and object fields. Since Racket does not provide a mechanism for interposing on field accesses, we manually changed the benchmarks to make all external field accesses go through new getter and setter methods for which Racket does offer interposition features. This manual change does not affect the behavior of the programs.

Finally, it is time to explain the remark (2) in section 4.1. The program modifications described here do not affect the generation of the migration lattice. Recall that the lattice of a GTP program is built from the components that are either typed or untyped. And, as specified in section 4.1,

the construction of the lattice ignores the type interface, which in this experiment consists of the mutated type interface, the flow adaptor, and the variant of the original type interface. Consequently, the lattices for all mutants are exactly the same as the lattices of the corresponding original program.

5.4 Sampling Debugging Scenarios

The 294 usable mutants across the selected GTP programs yield over two million different mutant × configuration pairs, which in turn, are the debugging scenarios that the rational programmer can explore. Hence, to perform the experiment within a feasible time-frame, we must sample this scenario space to obtain a reduced yet representative population. "Representative sampling" means that the results of the experiment on the sample generalize to the whole population. To ensure that the sample is representative, we use a stratified random sampling approach, breaking the population into strata based on source program, mutators, and mutants, in that order. This choice means we can generalize with high confidence, given that each of the stratification criteria captures relevant groupings within the population.

In more detail, within each suitable mutant's lattice of configurations, we sample from those configurations that do not correspond to trivial debugging scenarios. A trivial scenario is one for which the type checker directly identifies the impedance mismatch between the mutated interface and the library. Such trivial scenarios come about whenever the configuration uses the typed variant of a library component that provides a value whose interface type has been mutated. In this situation, the type checker discovers the conflict between the type annotations of the library and the interface. After filtering out the trivial scenarios, the interesting scenarios for a given mutant actually form a sub-lattice of the migration lattice, in which no configuration contains a typed variant of the library component with faulty interface types. As a result, the sub-lattice is computable. With the sub-lattice in hand, we uniformly sample 100 interesting scenarios per mutant — in other words, we make no assumptions about the difficulty of debugging scenarios based on their position in the lattice. Thus we arrive at a final population of 29,400 debugging scenarios for the rational programmer experiment.

While sampling scenarios lightens the computational burden on the rational programmer, it has consequences for our ability to generalize the results of the experiment to the full population of interesting scenarios. Statistical principles tell us that our results are representative of the full population with 95% confidence and within some margin of error, depending on the size of the sample. To remind readers of this uncertainty in generalization, the next section describes the results of our experiment with a note alongside each figure concerning the corresponding margins of error.

6 WHAT ARE THE RESULTS OF THE RATIONAL PROGRAMMER EXPERIMENT

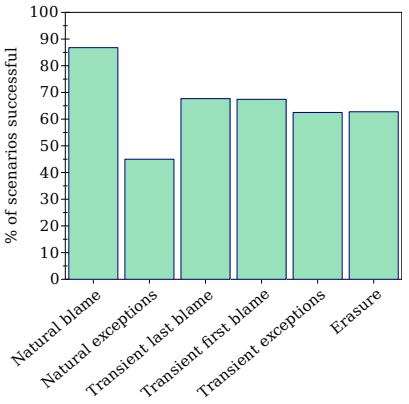
We run the experiment giving each debugging scenario a 10 minute timeout and a 6 GB memory limit. In aggregate, following all trails required thousands of compute hours.

Figure 5 shows the high level success rate estimates of each rational programmer mode for the debugging scenarios of the experiment. These success rates illustrate points that form the basis of the rest of our analysis.

First, the Natural blame mode far outperforms all other modes: the rational programmer that heeds blame information from the Natural semantics explores successful blame trails in nearly 90% of the scenarios. Second, the next closest modes, both at nearly 70% of the scenarios, are the two Transient blame modes. The Erasure mode follows close behind with just under 65%. Finally, the Transient exceptions mode performs ever so slightly worse than Erasure, around 5% worse than its corresponding blame modes. Nonetheless, they all far outpace the Natural exceptions mode that is only successful for about 45% of the scenarios. Clearly, there are significant differences in

the utility of the error information the rational programmer relies upon—across both the different semantics and sources of error information within each.

Digging deeper into the causes of failed trails for each of the modes offers some insight into these differences. In the Natural blame mode, the rational programmer fails to reach a static error in about 3,400 scenarios, all for the same reason, namely, running the scenario results in an exception from the underlying language rather than blame. In the absence of blame, the Natural blame mode falls back on stacktrace information to make progress; in these scenarios, however, the stack contains no untyped modules in the program, giving the rational programmer no indication of where to look next, so it is stuck.



The upper bound margin of error is 0.08%.

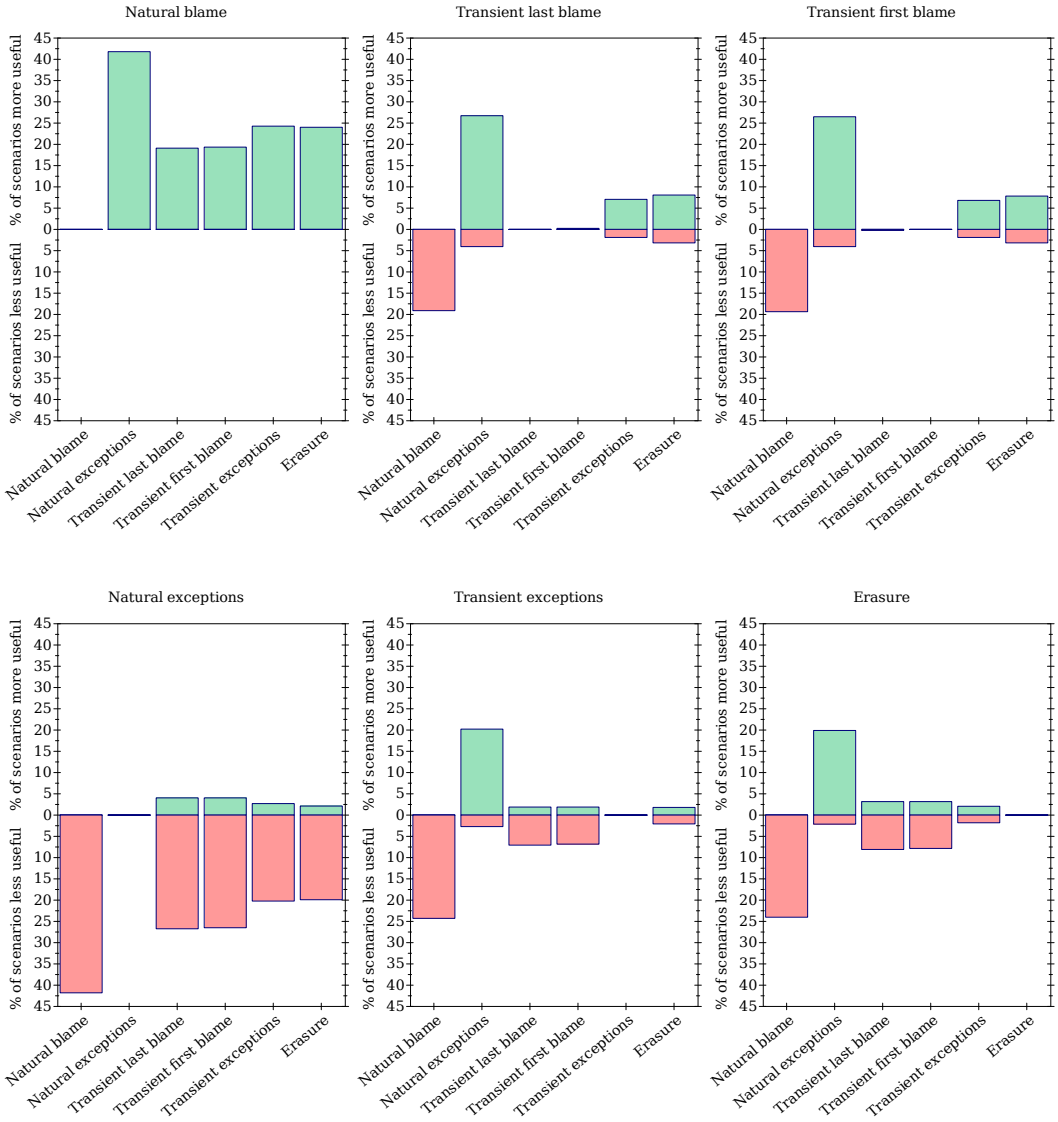
Fig. 5. Percentage rates of success.

Similarly the stacktrace information does not help the Natural exceptions mode in about 15,000 scenarios. Of those, 3,400 scenarios are the same as those that stymie the Natural blame mode. In around 11,500 additional scenarios the Natural run-time type checks do signal an impedance mismatch, but the Natural exceptions mode ignores the blame information, and the stack is unhelpful. This is not altogether surprising, however, because the checks likely occurred while a value of incorrect type passed across the boundary of the type interface; at that point, the only modules likely to be on the stack are client components. None of the client components (in any of the benchmarks) can ever cause a mismatch to be statically detected, since the mismatch is by construction between the interface and one or more library components. Thus, in the setting of this experiment, the Natural checks produce unhelpful stack information most of the time.

The two Transient blame modes fail in the same ways, spread over a few broad causes. Principal among them is unhelpful stack information, accounting for just under 6,500 failures. More interestingly, over 1,000 failures occur because Transient checks fail to detect the mismatch at all: the program completes (most probably with incorrect results). Finally, around 600 scenarios end in failure when Transient checks signal an error, but Transient blame is unhelpful. Specifically, the blame sequence is empty. The corresponding scenarios are instances where Transient's collaborative blame algorithm fails due to a fundamental limitation in tracking blame for built-in higher order functions. Lazarek et al. [2021] report the exact same problem; the interested reader may wish to consult that paper for further details.

Finally, the Erasure mode fails in two ways: either the stacktrace information available from the exceptions of the underlying language are unhelpful, or the program terminates without any error. Unhelpful stacktrace information account for 8,700 of the Erasure mode's failures, and the program terminates with no error information in 1,200 of the scenarios (again, likely with incorrect results).

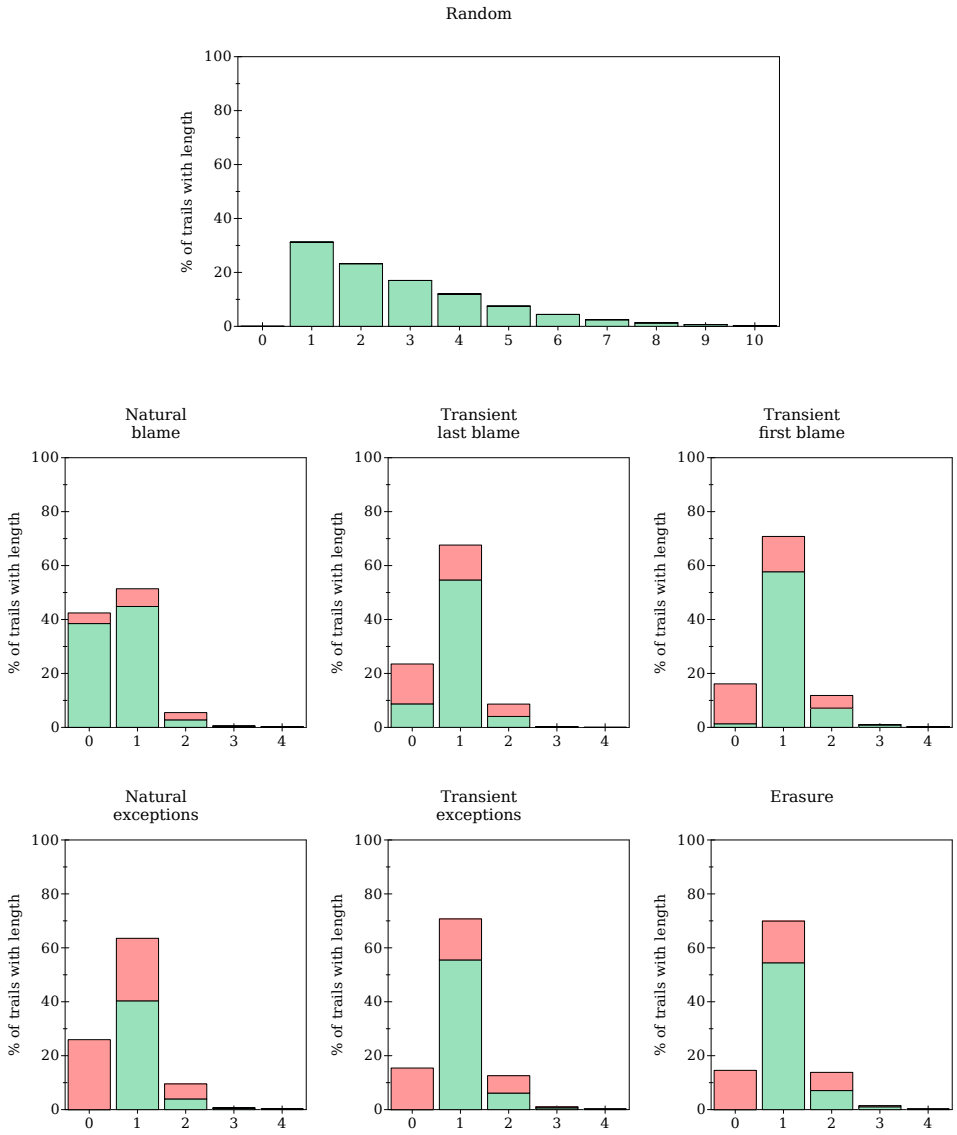
Figure 6 gives a head-to-head account of the success rates of the modes to shed light on the comparative utility of the sources of error information available to the rational programmer. Specifically, the figure names one plot per mode, where the plot compares the estimated percentage of scenarios where the named mode uses more (and less) useful information than each mode named along the x-axis. For instance, the top left plot illustrates that there are no scenarios where any of the other modes have more useful information than Natural blame mode for the same scenario. And while the Natural exceptions mode performs the worst in terms of overall success rates, the bottom left plot clarifies that there are in fact scenarios where the Natural exceptions mode is more successful than each of the other modes except for the Natural blame mode.



Each plot compares the mode named above the plot to every other mode. The green bars above 0 depict the estimated percentage of scenarios where the named mode has more useful information than the other. The red bars below 0 conversely depict the estimated percentage where the named mode has less useful information. The upper bound margin of error is 0.08%.

Fig. 6. Head to head usefulness comparisons.

These results offer answers to the experimental question from section 4.5. Concretely, we can answer question Q_1 in the affirmative: blame is useful in the context of Natural. There are a wealth of scenarios where the Natural blame mode improves over the Natural exceptions mode, and none to the contrary; indeed, the same is clear for the Natural blame mode compared to all others,



Each plot depicts the distribution of trail lengths for the mode named above. The proportion of successful trails (bottom of each stacked bar) and failed trails (top) are also indicated by color (green for success and red for failure). The upper bound margin of error is 0.01%.

Fig. 7. Trail length distributions per mode.

answering the Q_* questions concerning the Natural blame mode as well. Questions Q_2 and Q_3 are similarly answered in the affirmative, though there is a tiny proportion of scenarios where Transient exceptions improve over each interpretation of Transient blame. Both Transient blame modes improve over Erasure in a small proportion of scenarios, and the converse is only true in a tiny proportion. Thus the Q_* questions concerning Transient and Erasure can be answered in

favor of Transient's blame, though not by much. However, neither Transient blame mode appears preferable over the other.

The length of successful trails helps to clear some of that uncertainty. Figure 7 depicts the distribution of trail lengths for each mode, where each bar is also colored according to the proportion of successful and failing trails. The main takeaway from this data is that the Q_* questions about Transient first and last blame can be answered slightly in favor of the last blame interpretation, since it has a significantly higher proportion of successful trails with length zero.

7 WHAT CAN PROGRAMMERS LEARN FROM THE RATIONAL PROGRAMMER

An intuitive understanding of the rational programmer's workings is instrumental to interpreting the aggregate results of the previous section. Figure 8 provides a detailed account of one scenario from the GTP program `synth`, which offers a useful illustration of how each mode of the rational programmer works. The top left of the figure illustrates the program's dependency graph, and the rest of the figure details the trails that each mode explores.

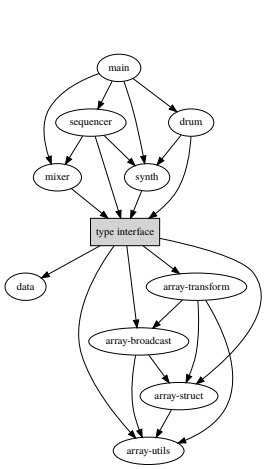
In this scenario, the type interface has been mutated so that the type of an `Integer` field in an `Array` data structure definition is replaced with `Any`. Locating this mistake takes the modes of the rational programmer on five different paths through the migration lattice of the (adapted) mutant, illustrated in the top right of the figure.

The table in the middle of the figure details how each of those paths play out, step by step. Each row of the table corresponds to a mode. Each column describes a point in the trail, starting from the root debugging scenario, with the result of running the corresponding configuration. The following column to the right then describes the configuration the rational programmer examines next in response to those results, and the results of that new configuration respectively; and so on. Finally, the **OK?** column summarizes whether the trail ends in success or failure.

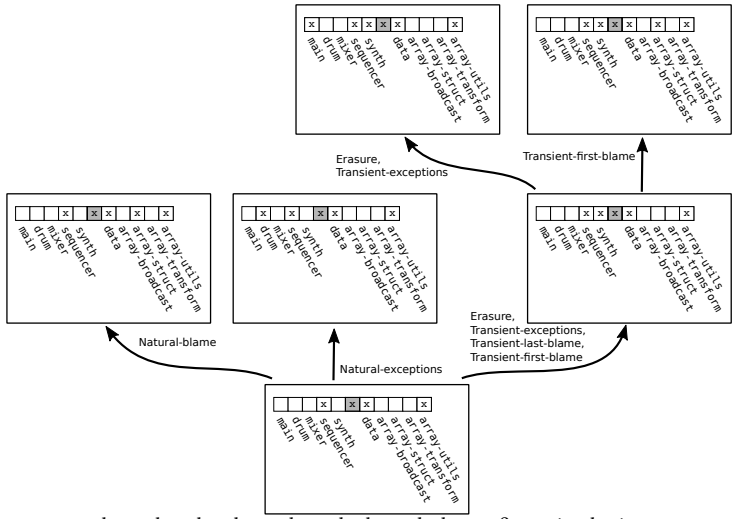
For instance, compare the first and third rows of the table. The first row, for the Natural blame mode, shows that the root configuration results in blame on the `array-struct` module. So the rational programmer types that module to obtain the configuration in the next column, which does not type check. In contrast, the Transient-last-blame mode's row shows that the root configuration does not result in blame but in stacktrace information, where `synth` is the top module. The rational programmer types that module, and the result of that new configuration is blame on the type interface. Readers familiar with the Transient semantics may wonder how blame can land on the interface, because it is a typed module. In fact, due to the adaptation described in section 5.3, the interface really consists of two typed modules sandwiching the untyped flow-adaptor module. This latter component is what Transient blames, and we interpret that as successfully identifying the interface. In practice, this situation corresponds to one where there is an untyped library module in between the buggy type interface and the typed library module that detects the mismatch, which would be blamed, and which, once annotated, would make the mismatch apparent to the type checker. Thus the two modes take different paths to success in this scenario.

7.1 Interpreting the Results

The experimental results suggest a few takeaways about the value of blame when types are mistakenly ascribed in gradually-typed programs. First, the information from run-time type checks—sans blame—is on the whole less helpful for the rational programmer than the information that would have been available from (possibly later) exceptions from the underlying language. This stands in contrast with Lazarek et al. [2021]'s finding that gradual run-time type checks offer the rational programmer comparable value to the regular safety checks of the underlying language. Of course, in practice working programmers won't know a-priori if they have made a mistake in



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	OK?
Natural -blame			drum array-struct		$\tau_\times$				✓
Transient -first-blame			synth main			synth main		$\tau_\times$	✓
Transient -last-blame			synth main			synth main			✓
Erasure			synth main			synth main			X
Natural -exceptions			drum			drum			X
Transient -exceptions			synth main			synth main			X

Legend

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

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

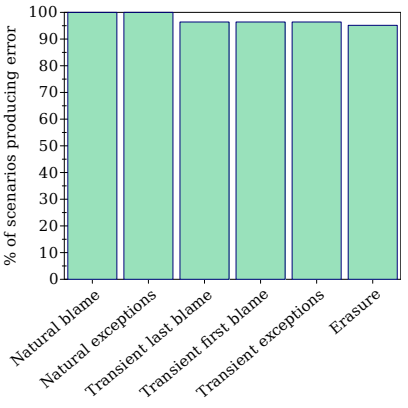
Fig. 8. An example scenario from synth, with the trails that each mode explores.

types or code, so the contrast raises the question of whether run-time type checks without blame offer debugging value for working programmers.

Unlike run-time type checks without blame, those with blame offer clearly valuable information, across all semantics. However, specifically in the context of mistakes in interface types, Natural blame outpaces that of Transient significantly. Indeed, figure 6 shows that Natural blame offers better information than all other modes in large proportions of the scenarios. In contrast, Transient's

blame information improves over Erasure's stacktrace information on some occasions and on others is worse, making it overall a marginal improvement over Erasure.

While Natural with blame thus appears the most useful in terms of the debugging information it offers, its high overhead is well-known to be prohibitive for use in deployment. At the same time, the more performant options that perform type checks at run time but without blame do not appear to offer debugging benefits over Erasure. So what should a working programmer do? The results suggest a dual strategy: use Erasure for deployment, and—if available—a Natural blame debugging mode during reproduction and debugging of mistakes discovered in deployed software. This strategy requires that not too many impedance mismatches go entirely unnoticed when using Erasure, and the data in figure 9 suggests that is probably the case.



The upper bound margin of error is 0.01%.

Fig. 9. Estimated percentage rates of bug detection (i.e. halting with an error).

7.2 Threats to Validity

The validity of these conclusions are subject to two categories of threats. The first category of threats concern the experimental setup. Some of those are described in preceding sections, namely: (i) the GTP programs we use may not be truly representative of all programs in the wild; (ii) our synthetic type mistakes may not be truly representative of all mistakes programmers make in ascribing types; and (iii) our adaptation of client-side behavior does not match exactly the reality of program behavior with clients programmed against incorrect type interfaces. While the design of the experiment attempts to mitigate these threats with the careful design and analysis of the scenario generation (sec. 5), the reader must keep them in mind when drawing conclusions.

The second category consists of external threats due to the philosophical underpinnings of the experimental design. Most fundamentally, the rational programmer itself does not necessarily reflect the way real programmers use gradual types or debug mistakes in type interfaces (section 7.3). At a more technical level, the experiment design assumes that the rational programmer can inspect and annotate library components, which real programmers may not be able to do (section 7.4). Finally, the experiment aims to answer the research questions in the restricted context of a single language with one syntax and type system but different semantics. While this is necessary for the apples-to-apples comparison of a scientific experiment, it also raises the question of how the results of the experiment transfer to other linguistic settings (section 7.5).

7.3 Threat: The Rational Programmer is not a Human Programmer

Programming language researchers know quite well that despite their simplified nature, models have an illuminating power. Consider Standard ML, the language with the most rigorous, extensive formal definition [Milner et al. 1998, 1990]. The model simplifies the language to an extremely small kernel, excluding most of what programmers find useful (e.g., the libraries, the runtime). Yet, many theory papers use models like this to prove theorems about their designs and thus guide language evolution (think Classic Java [Flatt et al. 1998], Featherweight Java [Igarashi et al. 2001]). Similarly, empirical PL research has also relied on highly simplified mental models of program execution for a long time. As Mytkowicz et al. [2009] report, ignorance of these simplifications can produce

wrong data—and did so for decades. Despite this problem, the simplistic model acted as a compass that helped compiler writers improve their product substantially over the same time period.

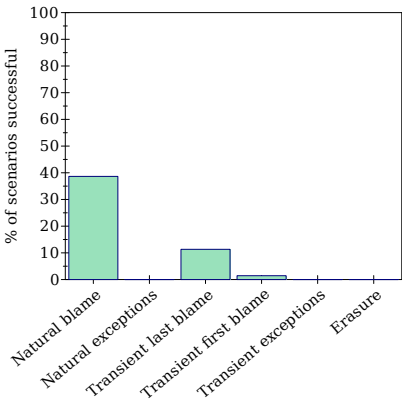
Like such models, the rational programmer is a simplified one. While the rational programmer experiment assumes that a programmer takes all information into account and sticks to a well-defined, possibly costly process, a human programmer may make guesses, follow hunches, and take shortcuts. Hence, the conclusions from the rational-programmer investigation may not match the experience of working programmers. Further research that goes beyond the scope of this paper is necessary to establish a connection between the behavior of rational and human programmers.

That said, the behavioral simplifications of the rational programmer are analogous to the strategic simplifications that theoretical and practical models make, and like those, they are necessary to make the rational programmer experiment feasible. Despite all simplifications, section 6 demonstrates that the rational programmer method produces results that offer a valuable lens for the community to understand some pragmatic aspects of the semantics of blame and gradual type checking, and it does so at scale and in a quantifiable manner.

7.4 Threat: Typing Library-side Modules

In the experiment, the rational programmer opens up and ascribes types to library components in the process of hunting down an impedance mismatch. When working programmers find themselves in the same situation, however, it is far from clear that they would be willing or able to do the same. This is especially relevant in settings like DefinitelyTyped, where the library in question is some third-party package on npm. In that case, the programmer relies on the authors of the type declaration file or the package to respond to a bug report and pick up the search of the bug. While anecdotal evidence suggests that it is common for programmers to issue bug reports, and type declaration and package authors to respond with fixes quickly [Hoeflich et al. 2022], assuming that they do so all the time is an experimental simplification.

Hence, the simplification naturally raises the question of what the results would look like if the rational programmer only modified client components. Figure 10 offers some indication of the answer to this question based on the data already available. It depicts the estimated overall success rates of each mode where the criteria for extending a blame trail excludes adding types to library components. That is, the rational programmer fails when error information points to a library component as the next point of focus of the investigation.



The upper bound margin of error is 0.04%.

Fig. 10. Estimated percentages of trails that succeed without typing library modules.

This data draws a significantly different picture. While the Natural blame mode remains by far the most successful, Transient-last-blame emerges here as the best alternative information, and none of the modes using exception information, including Erasure, have any success. This is not altogether surprising because, as discussed in section 6, even if stacktrace information points to client components, adding types to client components can never turn the impedance mismatch into a static type error.

This filter on the data does not tell the whole story, however. While it does suggest that Natural blame offers the best debugging information in this setting too, and by a significant margin, a followup experiment is necessary to see if that suggestion bears out for true client-side rational programmer modes. For instance, a true client-side version of each mode would simply filter library

components from stacktrace information and pick the next client component instead of failing when the top of the stack is a library component. Such modes model programmers that question the correctness of type declarations and third-party libraries as a last resort, and only after exhausting all possibilities that the problem stems from their code.

7.5 Threat: Different Languages, Different Types, Different Checks

While the type systems of Typed Racket and TypeScript are quite similar, their run-time safety checks differ significantly. The former is well-known for its informative run-time error messages and stacktrace information (due to its origins in education); the latter is a derivative of JavaScript, which famously ignores run-time errors as much as possible and often produces sparse stack traces (if any). Hence, the results for an analogous study of TypeScript may make the Natural and Transient semantics look much stronger than the Erasure semantics. An attempt to replicate the experiment in the context of Typescript is needed to clarify whether the conclusions of this paper transfer from one linguistic setting to another. This paper offers a blueprint and techniques to researchers that would like to take up this challenge. While the ideas and techniques we use should be useful for replication in any linguistic context, details such as the specific mutators that are relevant and the adaptor implementation approach will vary across contexts.

8 WHAT DOES PRIOR RESEARCH SAY ABOUT THIS PROBLEM

Lazarek et al. [2021] present the only prior work directly related to this one. They introduce the rational programmer method and evaluate the effectiveness of blame information when debugging code-originated impedance mismatches in a gradually typed language; they leave open the question of how to study impedance mismatches that are due to mistakes in type interfaces. This paper validates that the rational-programmer method can be adapted to understand debugging mistakes in type interfaces (sec. 2), but doing so demands two innovations: a novel approach to create interesting debugging scenarios (sec. 5) as well as careful adaptation of the key notion of blame trails (sec. 4). This success may also be a guide for others looking to apply this evaluation method.

There are three significant bodies of adjacent work. First, a number of papers investigate the prevalence of mistakes in gradual types, their theoretical underpinnings, and ways to mitigate them [Campora and Chen 2020; Cristiani and Thiemann 2021; Feldthaus and Møller 2014; Greenman et al. 2019a; Hoeflich et al. 2022; Kristensen and Møller 2017b; St-Amour and Toronto 2013; Williams et al. 2017]. Second, others aim to help programmers debug type errors in *statically typed* settings [Becker et al. 2016; Chen and Erwig 2014; Pavlinovic et al. 2014; Seidel et al. 2016, 2018; Zhang and Myers 2014], including some that find a significant portion of those errors arise from incorrect type annotations [Wu and Chen 2017]. Third, there is an extensive area of active research on developing and applying human-centered approaches to understand the practical aspects of type system design [Brown et al. 2018; Coblenz et al. 2020, 2021; Denny et al. 2021; du Boulay and Matthew 1984; Hanenberg 2010; Hanenberg et al. 2014; Lubin and Chasins 2021; Spiza and Hanenberg 2014; Wexelblat 1976]. Within this area, Tunnell Wilson et al. [2018] survey programmers about their general preferences between semantics for gradual typing.

Finally, our debugging scenario corpus incorporates techniques from the software engineering research world, particularly mutation testing [DeMillo et al. 1978; Lipton 1971] and software component adaptation [Keller and Hölzle 1998; Mätzel and Schnorf 1997].

9 WHERE TO GO FROM HERE

When it comes to *detecting* type interface mistakes, all semantics are essentially equally good, at least for these programs. When it comes to *locating* those mistakes, however, the Natural-with-blame mode is the clear winner. In fact, it is the only combination that seems to provide a significant

edge over industry's Erasure semantics. All other academic semantics with blame offer limited benefits over Erasure at providing debugging hints. And notably, academic semantics *without blame* fare no better, or even worse, than Erasure.

Combining these observations with the results of Lazarek et al. [2021] suggests that in industrial gradually typed languages, such as TypeScript, Erasure seems to suffice for deployment. But, these languages would also significantly benefit from a Natural-with-blame development mode.

Despite the strong similarities between the type systems of Typed Racket and TypeScript, it remains open whether the insights concerning the former apply to the latter, too. Confirming them will require a new backend for TypeScript and another rational-programmer investigation.

Finally future work should refine the cost aspect of the rational-programmer investigation, specifically cost as in developer time. The rational programmer, as instantiated in both this work and Lazarek et al.'s prior work, does not account for the actual time spent on detecting and locating bugs. That is, the rational programmer makes no distinction between identifying the bug in ten seconds or ten hours. Instead the investigations crudely approximate developer time with the number of type-annotation steps, which in particular hides the reality that some components are easy to annotate and others are not. Furthermore, they do not consider how early in a program's execution a mistake is surfaced, despite the common wisdom that reporting mistakes early rather than late in a long-running program has significant practical benefits. In short, adding dimensions of time to a rational programmer investigation should become a high priority.

ACKNOWLEDGMENTS

Felleisen and Greenman were partly supported by NSF grant SHF 1763922. Greenman was partly supported by NSF grant 2030859 to the CRA for the CIFellows program. Lazarek and Dimoulas were partly supported by NSF Career Award 2237984. We also thank the anonymous ICFP reviewers for their constructive feedback.

REFERENCES

Brett A. Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. 2016. Effective Compiler Error Message Enhancement for Novice Programming Students. *Computer Science Education* 26, 2-3 (2016), 148–175. https://doi.org/10.1080/08993408.2016.1225464

Neil C. C. Brown, Amjad AlTadmri, Sue Sentance, and Michael Kölling. 2018. Blackbox, Five Years On: An Evaluation of a Large-scale Programming Data Collection Project. In *ICER*. 196–204. https://doi.org/10.1145/3230977.3230991

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

John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2017. Migrating Gradual Types. *PACMPL* 2, POPL (2017), 15:1–15:29. https://doi.org/10.1145/3158103

Sheng Chen and Martin Erwig. 2014. Counter-Factual Typing for Debugging Type Errors. In *POPL*. 583–594. https://doi.org/10.1145/2535838.2535863

Michael Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. 2020. Can Advanced Type Systems Be Usable? An Empirical Study of Ownership, Assets, and Typestate in Obsidian. *PACMPL* 4, OOPSLA (2020), 132:1–132:28. https://doi.org/10.1145/3428200

Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2021. PLIERS: A Process That Integrates User-Centered Methods into Programming Language Design. *ACM Trans. Comput.-Hum. Interact.* 4, Article 28 (2021), 53 pages. https://doi.org/10.1145/3452379

Fernando Cristiani and Peter Thiemann. 2021. Generation of TypeScript Declaration Files from JavaScript Code. In *International Conference on Managed Programming Languages and Runtimes*. 97–112. https://doi.org/10.1145/3475738.3480941

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

Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, Zachary C. Albrecht, and Garrett B. Powell. 2021. On Designing Programming Error Messages for Novices: Readability and its Constituent Factors. In *CHI*. 55:1–55:15. https://doi.org/10.1145/3411764.3445696

Benedict du Boulay and Ian Matthew. 1984. Fatal Error in Pass Zero: How not to Confuse Novices. In *Readings on Cognitive Ergonomics - Mind and Computers*, Vol. 178. 132–141. https://doi.org/10.1007/3-540-13394-1_11

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

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1998. Classes and Mixins. In *POPL*. 171–183. https://doi.org/10.1145/268946.268961

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *POPL*. 303–315. https://doi.org/10.1145/2676726.2676992

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

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. 2022. A Transient Semantics for Typed Racket. *Programming* 2, 6. https://doi.org/10.22152/programming-journal.org/2022/6/9

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. *JFP* 29, e4 (2019), 1–45. https://doi.org/10.1017/S0956796818000217

Stefan Hanenberg. 2010. An Experiment about Static and Dynamic Type Systems: Doubts about the Positive Impact of Static Type Systems on Development Time. In *OOPSLA*. 22–35. https://doi.org/10.1145/1869459.1869462

Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. 2014. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering* 19, 5 (2014), 1335–1382. https://doi.org/10.1007/s10664-013-9289-1

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://doi.org/10.1257/aer.91.2.73

Joshua Hoeflich, Robert Bruce Findler, and Manuel Serrano. 2022. Highly Illogical, Kirk: Spotting Type Mismatches in the Large despite Broken Contracts, Unsound Types, and Too Many Linters. *PACMPL* 6, OOPSLA (2022), 142:1–142:26. https://doi.org/10.1145/3563305

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *TOPLAS* 23, 3 (2001), 396–450. https://doi.org/10.1145/503502.503505

Ralph Keller and Urs Hölzle. 1998. Binary component adaptation. In *ECOOP*. 307–329. https://doi.org/10.1007/BFb0054097

Erik Krogh Kristensen and Anders Møller. 2017a. Inference and Evolution of TypeScript Declaration Files. In *FASE*. 99–115. https://doi.org/10.1007/978-3-662-54494-5_6

Erik Krogh Kristensen and Anders Møller. 2017b. Type Test Scripts for TypeScript Testing. *PACMPL* 1, OOPSLA (2017), 90:1–90:25. https://doi.org/10.1145/3133914

Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2021. How to Evaluate Blame for Gradual Types. *PACMPL* 5, ICFP (2021), 68:1–68:29. https://doi.org/10.1145/3473573

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.

Justin Lubin and Sarah E. Chasins. 2021. How Statically-Typed Functional Programmers Write Code. *PACMPL* 5, OOPSLA (2021), 155:1–155:30. https://doi.org/10.1145/3485532

Kai-Uwe Mätzel and Peter Schnorf. 1997. *Dynamic component adaptation*. Technical Report. Ubilab Technical Report 97.6.

Microsoft. [n. d.]. TypeScript. Retrieved February 23, 2023 from https://www.typescriptlang.org.

Zeina Migeed and Jens Palsberg. 2019. What is Decidable about Gradual Types? *PACMPL* 4, POPL (2019), 29:1–29:29 pages. https://doi.org/10.1145/3371097

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

Robin Milner, Robert Harper, David MacQueen, and Mads Tofte. 1998. *The Definition of Standard ML, Revised Edition*. MIT Press.

Robin Milner, Mads Tofte, and Robert Harper. 1990. *The Definition of Standard ML*. MIT Press.

Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. 2019. Dynamic Type Inference for Gradual Hindley–Milner Typing. *PACMPL* 3, POPL (2019), 18:1–18:29 pages. https://doi.org/10.1145/3290331

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing Wrong Data without Doing Anything Obviously Wrong! In *ASPLOS*. 265–276. https://doi.org/10.1145/1508244.1508275

Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2014. Finding Minimum Type Error Sources. In *OOPSLA*. 525–542. https://doi.org/10.1145/2660193.2660230

Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. 2021. Solver-Based Gradual Type Migration. *PACMPL* 5, OOPSLA (2021), 111:1–111:27 pages. https://doi.org/10.1145/3485488

Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The Ins and Outs of Gradual Type Inference. In *POPL*. 481–494. https://doi.org/10.1145/2103656.2103714

Eric L. Seidel, Ranjit Jhala, and Westley Weimer. 2016. Dynamic Witnesses for Static Type Errors (or, Ill-Typed Programs Usually Go Wrong). In *ICFP*. 228–242. https://doi.org/10.1145/2951913.2951915

Eric L. Seidel, Ranjit Jhala, and Westley Weimer. 2018. Dynamic Witnesses for Static Type Errors (or, Ill-Typed Programs Usually Go Wrong). 28 (2018), e13. https://doi.org/10.1017/S0956796818000126

Herbert A. Simon. 1947. *Administrative Behavior*. MacMillan.

Samuel Spiza and Stefan Hanenberg. 2014. Type Names without Static Type Checking Already Improve the Usability of APIs (as Long as the Type Names Are Correct): An Empirical Study. In *Modularity*. 99–108. https://doi.org/10.1145/2577080.2577098

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

T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. Chaperones and Impersonators: Run-time Support for Reasonable Interposition. In *OOPSLA*. 943–962. https://doi.org/10.1145/2384616.2384685

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

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

Richard L. Wexelblat. 1976. Maxims for Malfeasant Designers, or How to Design Languages to Make Programming as Difficult as Possible. In *ICSE*. 331–336. https://doi.org/10.5555/800253.807695

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

Baijun Wu and Sheng Chen. 2017. How Type Errors Were Fixed and What Students Did? *PACMPL* 1, OOPSLA (2017), 105:1–105:27 pages. https://doi.org/10.1145/3133929

Danfeng Zhang and Andrew C. Myers. 2014. Toward General Diagnosis of Static Errors. In *POPL*. 569–581. https://doi.org/10.1145/2535838.2535870

Received 2023-03-01; accepted 2023-06-27