NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2024-10

June, 2024

An Investigation of the Pragmatics of Debugging with Contracts and Gradual Types

Lukas Lazarek

Abstract

This dissertation demonstrates that a new empirical method, called the Rational Programmer, can examine the pragmatics of contracts and gradual typing in the context of debugging at scale and in an automated manner. The method begins with a hypothesis about how a specification technique's error information assists in locating bugs. Then, the method calls for designing an algorithm that simulates an idealized programmer using the technique to locate bugs. We can then test the algorithm on a variety of scenarios in a large-scale automated experiment, the results of which provide evidence to either support or refute the hypothesis.

Concretely, this dissertation provides data from evaluations of two techniques: contracts and gradual typing. In the case of contracts, the rational programmer helps reveal that while carefully-designed blame information does live up to its hypothesized debugging benefits, primitive stacktrace information appears to do so nearly as well. In the case of gradual typing, the evaluation results suggest that when mistakes occur in code, academic approaches' specialized error information provide marginally better debugging help; when mistakes occur in type annotations, however, the results do suggest that the special error information offers valuable debugging information. These results demonstrate the value of the rational programmer and point to several directions for future investigations.

Keywords

programming languages, language design, debugging, contracts, gradual typing

NORTHWESTERN UNIVERSITY

An Investigation of the Pragmatics of Debugging with Contracts and Gradual Types

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

In the Field of Computer Science

By

Lukas Lazarek

EVANSTON, ILLINOIS

June 2024

© Copyright by Lukas Lazarek 2024

All Rights Reserved

ABSTRACT

This dissertation demonstrates that a new empirical method, called the Rational Programmer, can examine the pragmatics of contracts and gradual typing in the context of debugging at scale and in an automated manner. The method begins with a hypothesis about how a specification technique's error information assists in locating bugs. Then, the method calls for designing an algorithm that simulates an idealized programmer using the technique to locate bugs. We can then test the algorithm on a variety of scenarios in a large-scale automated experiment, the results of which provide evidence to either support or refute the hypothesis.

Concretely, this dissertation provides data from evaluations of two techniques: contracts and gradual typing. In the case of contracts, the rational programmer helps reveal that while carefully-designed blame information does live up to its hypothesized debugging benefits, primitive stacktrace information appears to do so nearly as well. In the case of gradual typing, the evaluation results suggest that when mistakes occur in code, academic approaches' specialized error information provide marginally better debugging help; when mistakes occur in type annotations, however, the results do suggest that the special error information offers valuable debugging information. These results demonstrate the value of the rational programmer and point to several directions for future investigations.

ACKNOWLEDGEMENTS

There are many people I want to thank for making my PhD experience great.

First of all I want to thank my advisor Christos Dimoulas for his guidance, patience, and support. He taught me how to focus my interests, and modeled a rigorous yet pragmatic approach to learning and research that I continuously aspire to. Beyond scholarly mentorship, I must also thank Christos for always being empathetic and understanding that the PhD was a part of my life rather than the other way around. I am fortunate to be his student.

I'd like to thank my committee members and collaborators for their feedback, insightful commentary, and assistance: Robby Findler, Jessica Hullman, Matthias Felleisen, Ben Greenman, and Alexis King.

I'd like to thank Jay McCarthy for introducing me to PL, and encouraging me to come to graduate school in the first place.

I'd like to thank my lab mates, current and former, for sharing the Northwestern PL journey with me, in all its particularity: Spencer Florence, Shu-Hung You, Dan Feltey, Alex Owens, Tochukwu Eze, Chenhao Zhang, Bangyen Pham, Nathaniel Hejduk, Hakan Dingenc, Joshua Hoeflich, Peter Zhong, and Caspar Popova.

I'd like to thank my friends across the rest of the department and university for making the journey not only fun, but possible – I couldn't have made it here without you all. In particular, I want to thank: Enrico Deiana and Ettore Trainiti, who were among the first to welcome me to Northwestern and have been such great friends since; Madhav Suresh, who has been a constant wellspring of conversation, fun, and ideas, and from whom I have learned more than many formal mentors; Hyeok Kim, who has probably spent nearly as much time in my office as me, which ultimately helped me not only have fun but also improve my work—

including this document; Michalis Mamakos, who has never failed to provide a healthy dose of sarcasm; Tommy McMichen, who I could always rely upon to stop by and chat, and helped me see a way of talking about research without being too serious; Nick Wanninger, who always welcomed me into his office, sometimes even with a serenade; Stephanie Jones, Sanchit Kalhan, and Vishesh Kumar, who have always been a pleasure to see but never around enough; Leif Rasmussen, Chenhao Zhang, and Can Gürkan, who found rolling dice equally entertaining. Though I have many reasons to thank each of the following people, for the sake of brevity, let me just say thanks also to: Abhraneel Sarma, Suman Bhandari, Nathan Greiner, Federico Sossai, Atmn Patel, Brian Homerding, and Karl Hallsby.

Finally, I'm grateful to my family, the Rieders, and Amrina Rosyada for their love and encouragement throughout this journey.

TABLE OF CONTENTS

Acknowledgments 4

List of Figures 12

List of Tables 13

Chapter 1: Introduction 14

1.1 A First Taste of Contracts and Gradual Types 14

1.2 Debugging with Contracts and Gradual Types 17

1.3 Pragmatics 18

1.4 The Rational Programmer 19

1.5 Outline 19

Chapter 2: The Rational Programmer Framework at a High Level: Linking Semantics and Pragmatics 20

Chapter 3: From the Rational Programmer Framework to a Method for Debugging with Contracts and Gradual Types 23

3.1 Making Automatable Procedures 23

3.2 Designing a Rigorous Experiment 25

3.3 Obtaining Scenarios for an Experiment 27

Chapter 4: Experiment 1: Behavioral Contracts and Behaviorial Bugs in Code 30

4.1 Background: Contracts and Blame 30

4.2 The Blame Shifting Hypothesis 39

4.3 The Blame Shifting Procedure 40

4.4 The Experiment in Precise Terms 45

4.5 Obtaining Debugging Scenarios for Contracts 48

4.6 Results 58

4.7 Lessons Learned 69

4.8 Summary 71

Chapter 5: Experiment 2: Gradual Types and Type-level Bugs in Code . . 72

5.1 Background: Gradual Types 72

5.2 Challenges 84

5.3 The Hypothesis for Gradual Types 86

5.4 The Procedure for Gradual Types 87

5.5 The Experiment in Precise Terms 92

5.6 Obtaining Debugging Scenarios for Gradual Types 96

5.7 Results 104

5.8 Lessons Learned . . . 111

5.9 Summary . . . 119

Chapter 6: Experiment 3: Gradual Types and Bugs in Type Annotations . 121

6.1 Background: Gradual Types Can Be and Often Are Wrong . . . 121

6.2 The Hypothesis for Type Interface Mistakes . . . 129

6.3 The Procedure for Type Interface Mistakes . . . 130

6.4 The Experiment in Precise Terms . . . 134

6.5 Obtaining Debugging Scenarios with Type Interface Mistakes . . . 136

6.6 Results . . . 147

6.7 Lessons Learned . . . 152

6.8 Summary . . . 160

Chapter 7: Related Work . . . 161

7.1 Contracts and Gradual Typing . . . 161

7.2 Evaluations of Debugging Information and Strategies . . . 166

7.3 Methodological Inspirations for Scenario Generation . . . 168

Chapter 8: Conclusion . . . 171

References . . . 190

Appendix A: Stratified Proportion Estimation . . . 191

Appendix B: Revisiting Experiment 2 with the Natural Bias 193

Appendix C: Revisiting Experiment 3 with the Erasure Bias 199

LIST OF FIGURES

1.1 Comparative view of specification tools in software development. 15

2.1 Rational programmer framework overview. 20

4.1 The Sieve of Eratosthenes, with a bug pointed out by the comment. 32

4.2 A precision progression of contracts for (a) `sieve` and (b) `sift`. 37

4.3 Definition of `sieved-stream/c`. 37

4.4 Lattice for the Sieve program. 42

4.5 Blame trail for the Sieve program. 43

4.6 Experiment overview 49

4.7 Breakdown of interesting mutants by mutator, per benchmark. 54

4.8 Stratification groups for stratified random sampling when using mutation. . . 57

4.9 Percentage rates of success. 59

4.10 Head to head usefulness comparisons. 61

4.11 Trail length distributions per mode. 63

4.12 Simple program inspired by `dungeon` that defeats blame shifting. 65

4.13 The shape of paths generated by `mbta`. 67

5.1 One mixed-typed program, three interpretations. 77

5.2 A simplistic debugging scenario. 82

5.3 Experimental questions and their relevant modes. 95

5.4 The experimental process for one mode of the rational programmer. 96

5.5 Example program using occurrence typing. 100

5.6 Breakdown of interesting mutants by mutator, per benchmark. 102

5.7 Percentage rates of success. 105

5.8 Head to head usefulness comparisons. 106

5.9 Blame usefulness analysis 107

5.10 Programmer effort 109

5.11 Effort comparisons 110

5.12 An example scenario from take5, with every mode's resulting trail. 112

6.1 One program with an incorrect type interface, three interpretations. 124

6.2 Type mistakes captured by final mutant population. 143

6.3 A simple program illustrating the need for adaptors. 144

6.4 Adapting the program of figure 6.3. 145

6.5 Percentage rates of success. 148

6.6 Head to head usefulness comparisons. 150

6.7 Trail length distributions per mode. 151

6.8 An example scenario from synth, with the trails that each mode explores. . . 153

6.9 Estimated percentage rates of bug detection (i.e. halting with an error). . . . 156

6.10 Estimated percentages of trails that succeed without typing library modules. 158

B.1 Percentage rates of success. 194

B.2 Head to head usefulness comparisons. 195

B.3 Blame usefulness analysis 196

B.4 Programmer effort 197

B.5 Effort comparisons 198

C.1 Percentage rates of success. 199

C.2 Head to head usefulness comparisons. 200

C.3 Trail length distributions per mode. 201

LIST OF TABLES

1.1 Error information comparison 18

3.1 Types of experimental questions. 27

4.1 Benchmarks summary 50

4.2 Summary of mutators 51

5.1 Summary 80

5.2 Summary of benchmarks 98

5.3 Summary of mutators 99

6.1 Summary of mutators 140

CHAPTER 1
INTRODUCTION

This dissertation provides arguments in support of the following thesis:

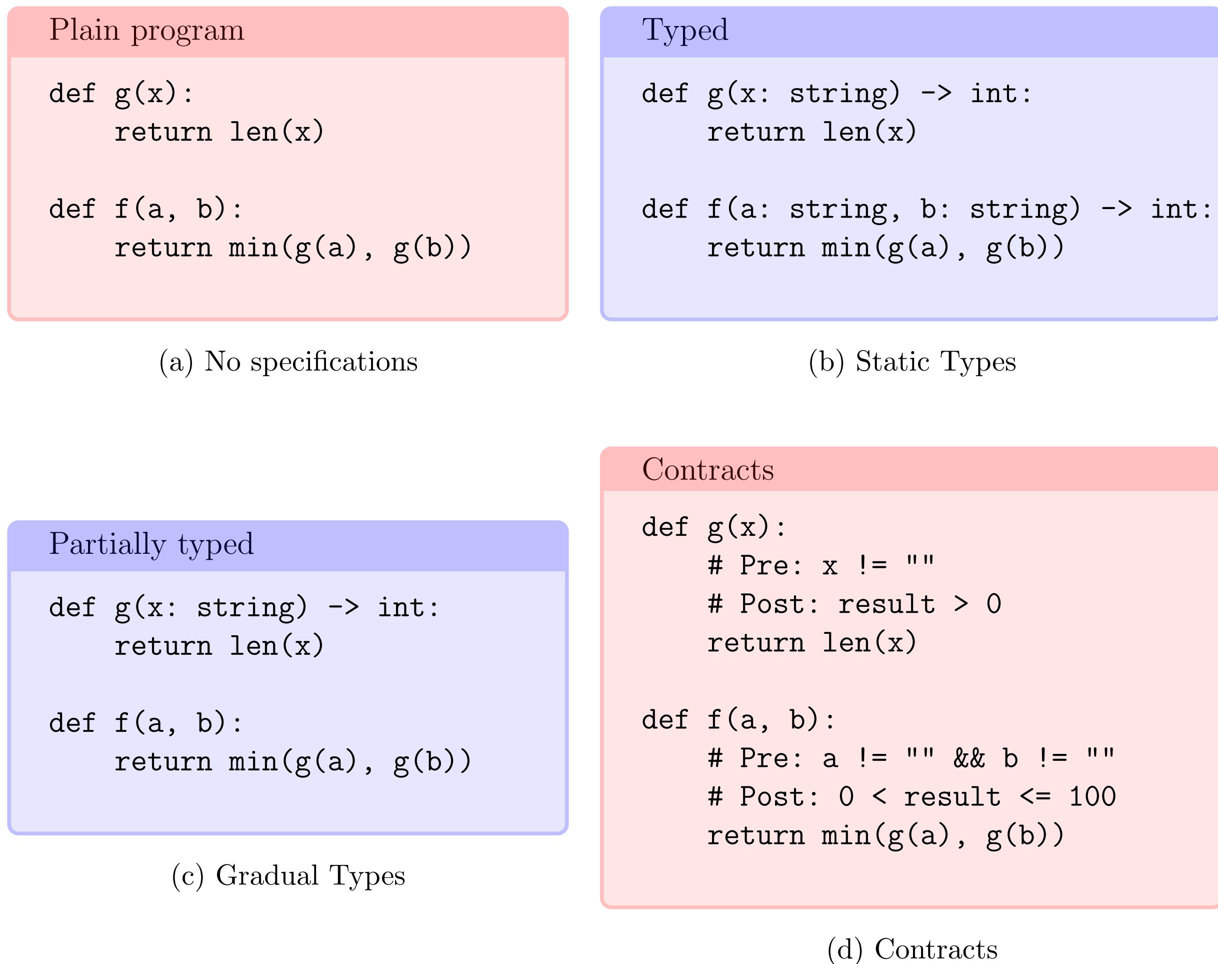
> Evaluating the pragmatics of debugging with contracts and gradual types, with the rational programmer, provides evidence that contract-based semantics and blame are useful for debugging.

In the rest of this chapter, I give a high-level explanation of the key pieces of this thesis before outlining how the remaining chapters dive into these pieces in detail.

1.1 A First Taste of Contracts and Gradual Types

All software has bugs; unfortunately, we usually do not know what or where they are. Specification techniques aim to help developers deal with this reality by allowing them to define a separate, detailed specification of what a program is intended to do and subsequently verify the actual program against that definition. There is a broad and varied array of specification techniques available to developers today, each offering different trade-offs in the kinds of specifications they support and how they are checked.

To illustrate the nature of these techniques and their different trade-offs concretely, consider the simple example of figure 1.1a. The figure presents a basic snippet of pseudocode consisting of two function definitions: one function `g`, performs a simple string-length operation on its input, and another function `f`, which accepts two inputs and applies `g` to each before calculating the minimum of the two results. Although extremely simple, this program



(a) No specifications

(b) Static Types

(c) Gradual Types

(d) Contracts

Figure 1.1: Comparative view of specification tools in software development.

snippet is sufficient to demonstrate the key ideas of the specification techniques that this dissertation focuses upon.

By far the most prevalent specification technique in practical use today is static types. Static types require developers to annotate their code with *types* describing what datatypes it uses. For example, figure 1.1b is the example program with type annotations specifying that `g` accepts a string and returns an integer, and that `f` accepts two strings and returns an integer. A type checker can then use these annotations to catch situations where the annotations of different pieces of code don't match up, which probably corresponds to a mistake in the code. A significant weakness of type checking is its conservative nature, though, meaning it occasionally rejects correct code. While this dissertation does not investigate static types (sec. 7.2.1 identifies a significant body of existing work in that context), it serves as a useful starting point from which to introduce gradual types and contracts.

Gradual types add a little flexibility to static types by allowing programmers to leave parts of their code un-annotated (or un-typed). Figure 1.1c illustrates one way this can look, with `g` annotated (typed) and `f` not (untyped). In the absence of complete types, the type checker checks as much as it can using what annotations there are in the program, making optimistic decisions in the face of insufficient information. In some approaches, gradually typed languages convert annotations into dynamic checks to compensate for the incomplete type information. For instance, the type checker may need to know that the result of `f` is an integer in order to typecheck some use of `f`, but may not verify that fact itself because `f` is untyped; in that case, some gradually typed languages add a run-time check that ensures `f`'s result is an integer after every time it is called. Gradual types cater to the practical realities of prototyping and incremental development, allowing developers to introduce types piecemeal into existing, untyped codebases.

Diverging significantly from types, contracts represent an entirely different specification approach. They enable programmers to write specifications in terms of pre- and post-condition assertions, which are verified dynamically as the program executes. As the example specifications of figure 1.1d illustrate, this approach supports highly expressive specifications; `g`'s contract, for instance, stipulates that its input should never be an empty string and that its output lies within the range 1–100. While thus supporting arbitrarily precise specifications, contracts are only checked as the program runs—often leading to performance overheads.

1.2 Debugging with Contracts and Gradual Types

The prior section demonstrates the differences between contracts and gradual types in terms of what kind of specifications programmers can write. Another major distinguishing factor between these options, however, is what happens when either technique does detect a problem in the program; in particular, the information they offer to help developers debug the problem varies dramatically between techniques.

To illustrate concretely, consider again the examples of the prior section. With static types, the type checker raises an error before the program runs that identifies one or more pieces of code where the type annotations don't match up. With contracts, the contract system halts the program with an error that describes which assertion failed, and the value that fails to satisfy the assertion; in addition, contracts provide a special mechanism intended to help with debugging called *blame*, which identifies one part of the program as being at fault for the violation. Finally, gradual types offer a range of different possible information, including type checker errors just like static types, blame information similar to that of contracts, or just plain stacktraces. Table 1.1 summarizes these differences compactly, with

one row naming the kind of information each technique provides, one row describing it, and a third row providing an example.

Table 1.1: Error information comparison

Static Types	**Gradual Types**	**Contracts**
Type checker error before running program	Type checker error before running program Blame like contracts Stacktraces	Contract violation with blame
Identifies mismatched types at some program location	*… see left and right …*	Identifies failed assertion, the witness value, and blames one component

`expected String,`
`found Bytes`
`on line 7`
	`stacktrace: f`
	`g`
		`101 fails postcondition`
		`0 < result ≤ 100,`
		`blaming f`

1.3 Pragmatics

The question this dissertation begins with is how useful the different kinds of information of the prior section are and how they compare to each other.

Unpacking this question: Semantics—i.e. formal models and their properties—provide precise understandings of what error information these techniques produce. However, semantic models and their properties don't make connections from that error information to practical outcomes like locating bugs. Hence, the question at hand is what those connections look like—or if they exist at all.

In this dissertation, I call connections of this nature the *pragmatics* of the techniques at hand, and I investigate them with a recently developed methodological framework called the rational programmer. This dissertation is a first application of that idea to understand the pragmatics of debugging with the run-time errors produced by contracts and gradual

types. At a meta-level, therefore, the dissertation can also be understood as a validation of the usefulness of the rational programmer framework.

1.4 The Rational Programmer

Chapter 2 provides a detailed overview of the rational programmer framework. At a very high level, in the context of debugging with specifications, the framework consists of four pieces. First, a hypothesis about how the semantics of a specification technique relate to pragmatic outcomes like locating bugs; second, the design of an automated procedure that leverages the hypothesized semantic-pragmatic connections to locate bugs; third, a large scale automated experiment that tests the procedure on real programs; and fourth, a set of data, resulting from that experiment, providing evidence that either supports the hypothesis or provides concrete examples refuting it.

1.5 Outline

The rest of this dissertation is organized as follows. Chapter 2 provides a high level overview of the rational programmer framework and its key pieces. Chapter 3 describes the key challenges involved in instantiating that framework in the specific context of debugging with contracts and gradual typing, and the key insights to overcome them. The following three chapters then apply those insights to design three rational programmer experiments evaluating contracts and gradual typing in the context of debugging; chapter 4 describes a rational programmer experiment for contracts, chapter 5 describes one for gradual typing—under one assumption about gradual types, and chapter 6 describes a followup experiment for gradual typing that flips the assumption of the former one. Finally, chapter 7 provides an overview of related work, and 8 concludes with a few directions for future work.

CHAPTER 2
THE RATIONAL PROGRAMMER FRAMEWORK AT A HIGH LEVEL: LINKING SEMANTICS AND PRAGMATICS

At a high level, the rational programmer is a framework for empirically investigating questions about how information provided by programming languages and tools relates to practical outcomes. The framework consists of four pieces.

Hypothesis. From these abstract questions, the framework concretely begins by formulating hypotheses about their answers. To illustrate the nature of these hypotheses, consider for example a hypothesis about locating bugs with blame information from contracts. A good hypothesis for this context posits a concrete relationship between the error information offered by the technique and the location of bugs, such as being able to translate the information into the location of a bug. Concretely, this idea can be formulated as the following hypothesis:

> blame from contracts can be translated into the location of a bug by iteratively adding stronger contracts to the blamed component.

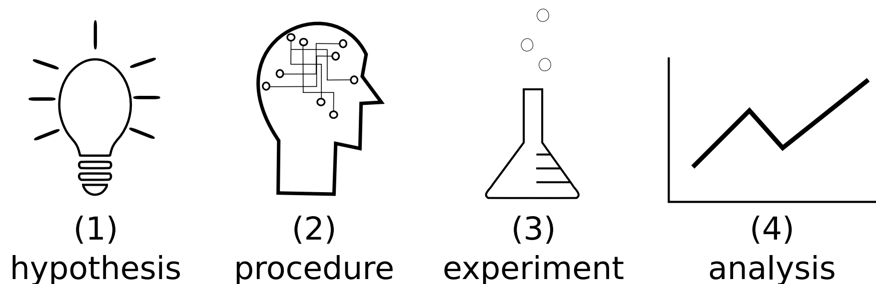


Figure 2.1: Rational programmer framework overview.

Automated procedure. The next piece of the rational programmer framework is to design an automated procedure that reifies or puts the hypothesis into action. That is, the procedure leverages the hypothesized relationship between semantic information and practical outcomes to accomplish a task. Continuing from the example hypothesis about contracts' blame, an example procedure could automatically respond to blame by adding stronger contracts to the blamed component and then re-running the program, repeating until it can no longer proceed—say, when the blamed component can not be given any stronger a contract.

Experiment. With the procedure in hand, the next piece is to test it out on real programs in a large scale automated experiment. In terms of the running example, the idea of the experiment is to run the procedure on a large corpus of buggy programs and recording for each such test whether the procedure succeeds at locating the program's bug.

Analysis. The result of that experiment is a set of data about the procedure's performance with respect to its goal. In particular, that data consists of two parts: one, the tests for which the procedure succeeds, provides evidence in support of the original hypothesis; the set of tests for which the procedure fails, on the other hand, are concrete counter-examples refuting the hypothesis. Besides helping to understanding why the hypothesis doesn't hold, these counter-examples can also serve as test-cases with which to improve the design or implementation of the language technique or tool under evaluation. To make things concrete in terms of the running example, the data for that experiment provides (1) evidence that blame can be translated into the location of bugs, and/or (2) example programs where blame leads the procedure astray.

Figure 2.1 summarizes this overview of the rational programmer framework pictorially. The first component is the hypothesis positing a specific relationship between error infor-

mation and bug-finding. The second is a procedure reifying that hypothesis. The third is a large-scale experiment testing the procedure on real programs. The final component is the data resulting from the experiment.

CHAPTER 3
FROM THE RATIONAL PROGRAMMER FRAMEWORK TO A METHOD FOR DEBUGGING WITH CONTRACTS AND GRADUAL TYPES

Figure 2.1 outlines four essential components to the rational programmer framework. This chapter describes how I instantiate this framework into a concrete method for investigating the pragmatics of contracts and gradual typing in the context of debugging, which requires refining and overcoming challenges in most of those components. The following sections consider each challenge in turn, expanding upon what each entails and sketching how I overcome them in my work. This chapter lays the foundation for the following three chapters, which illustrate how to build three concrete experiments in the image of this outline.

3.1 Making Automatable Procedures

The second component of the framework is a procedure reifying the hypothesis that the experiment examines, putting it into action—to try to locate bugs, in this specific context. This piece presents an essential challenge. The trouble is that most interesting ways of responding to debugging information (e.g. adding correct contracts to the blamed component) are not automatable. To overcome this, the key insight I use in my work is to pre-bake possible responses for the programs under test.

To illustrate the challenge and solution, consider the example hypothesis from the prior chapter (2) about translating contracts' blame into the location of bugs. A procedure reifying that hypothesis must be able to automatically add precise and correct contracts (or increase their precision) for an arbitrary component that is blamed. The reality, however,

is that crafting contracts is a fundamentally creative task that requires an understanding of the component at hand, what it is intended to do, how it is intended to fit in with other components in the program, and how those pieces lead to a succinct description of correct behavior. Indeed, even in the context of plain type-level specifications this remains a largely open problem [Campora, Chen, Erwig, 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, Chaudhuri, et al. 2012].

Instead, suppose we start from a set of (buggy) programs that don't have contracts. A human can look at any of those programs and perform the reasoning necessary to come up with suitable contracts for every component in the program. If we record these manually-written contracts in copies of the components, then we obtain two parallel versions of the components in a program: the original ones without contracts, and corresponding versions with our manually-written contracts. With this construction, we can select one of the two versions for every component in the program to get a *configuration* of the program — a concrete instantiation of these possible choices in a single program that we can run. And if running such a configuration results in blame identifying some component in the program, it is trivial to construct an automatic procedure to add contracts to that component—simply swap in the version with a pre-written contract to get a new configuration. Of course, the manual labor required by this solution raises significant scalability questions; section 3.3 describes how I manage that problem.

As a final note, the essence of this idea comes from Greenman [2023], Greenman, Takikawa, et al. [2019], and Takikawa, Feltey, et al. [2016], who introduce the GTP benchmark suite. Essentially, the suite provides parallel typed and untyped versions of each component in every program of the suite, enabling automated experiments that investigate the performance

effects of adding types to various components in a program.

3.2 Designing a Rigorous Experiment

The third component of the framework is an experiment that tests how successfully the procedure is able to locate bugs in real programs. Obtaining meaningful results from such an experiment requires careful design of the experimental setup. Testing the procedure of the prior subsection on some corpus of scenarios allows answering two kinds of experimental questions:

1. How reliably does the procedure succeed at the debugging task (locating the bug)?

2. What does the process looks like?

The first question quantitatively captures the pragmatic value of the technique under evaluation for locating bugs. The second question offers a secondary lens to understand *how* the technique helps the rational programmer succeed (or fail). It supports asking, for example, how quickly the rational programmer typically succeeds.

However, answers to these two questions alone are not usually sufficient to provide a solid basis for understanding the comparative pragmatics of different competing techniques. In particular, it is unclear how the answers to the experimental questions offer insight into whether, for instance, Typed Racket's blame offers better debugging information than Erasure. Consider for example if the results show that some procedure reliably succeeds using Typed Racket, and that it also reliably succeeds using Erasure; in that case, we have little information about how the two compare directly (e.g. do they succeed in different scenarios?), nor do we have information about whether blame is responsible for Typed Racket's success. In other words, the experimental questions are not themselves comparative in nature. Truly

understanding the value offered by a technique requires comparison, at the very least against established baselines.

The rational programmer framework can enable such comparisons through a mechanism dubbed *modes*. Modes are variations of the original procedure under examination, which embody the same original hypothesis but using a different technique or source of information. Each mode can be tested in independent sub-experiments to obtain answers to the two kinds of experimental questions alone, and then all the answers together can be synthesized into a fuller understanding of the comparative pragmatics of different techniques. In doing so, it is of course critical to control all the experimental variables so that only the point of comparison changes across each sub-experiment, so that differences in the answers to the experimental questions can be accurately attributed.

In more concrete terms, the multiple modes of an experiment embody both the various techniques under evaluation and baselines against which to compare them all. One such baseline that is useful for all evaluations of debugging information is a mode that acts randomly, using no information at all, which captures a sort of null hypothesis. This random mode provides a baseline against which to establish the non-random nature of the debugging information offered by a technique, and what benefits a technique offers over random chance.

Hence, with modes enabling comparative evaluation, the experimental questions of chapter 4-6's experiments additionally include some number of comparative versions of the first question; that is, how reliably is one mode better than another mode (i.e. mode A succeeds but mode B fails for the same scenario)? In cases where both modes succeed, answering these comparison questions may also require considering the debugging process of the two modes to decide which is better (e.g. mode A succeeds faster than B). That leaves in total a space of experimental questions, summarized in table 3.1. There is one experimental question per

Table 3.1: Types of experimental questions.

	per technique	**per pair of techniques**	**per mode (optional)**
Question	Better than baseline?	A better than B?	What does process look like?

technique of interest asking whether the corresponding mode is better than its baseline, one question per pair of techniques asking if either of the two are reliably better than the other, and optionally another question per mode asking what the debugging process looks like for that mode (e.g. is it typically quick, or lengthy?).

3.3 Obtaining Scenarios for an Experiment

The experiment component poses another essential challenge for practical implementations of the rational programmer framework. A rigorous experiment needs buggy test programs with significant and reliable information about their bugs: their nature, their location, and strong confidence that there are not more unknown bugs. While there is a wealth of buggy software available in online repositories, there is no curated collection with this key information about the bugs represented, so obtaining such programs is a challenge.

To overcome this challenge, the insight I use in my work is to obtain such programs by injecting bugs ourselves using mutation analysis [DeMillo, Richard J. Lipton, et al. 1978; Y. Jia and Harman 2011; Richard J Lipton 1971]. The essential idea of mutation is to make small syntactic modifications to a program, such as swapping a `+` with a `-` or a `1` with a `0`. Each such modification is dubbed a *mutation* of the original program, creating a *mutant* which has a single known potential bug.

This approach has several advantages, but also comes with drawbacks. The principal advantage of this approach is that by injecting the bugs ourselves, we have all the informa-

tion we need about them. Furthermore, by starting with a seed set of programs which is well-tested, we can have confidence that the only bug in the program is the one we injected. The drawbacks are that not all mutations actually introduce interesting bugs, and that the relationship between mutations and real bugs that programmers actually make is unclear. To address the first drawback, we develop criteria describing the suitability of mutants and ensure that we create a large and diverse population of mutants satisfying those criteria (see secs. 4.5.2.1, 5.6.3, and 6.5.2.1). To address the second drawback, we use or design mutators that have been carefully selected to fit the kinds of bugs under study in each of the three experiments (see in particular sec. 6.5.2.1). Furthermore, there is some empirical evidence that mutations effectively simulate real faults in the context of test suite evaluation [Andrews et al. 2005; Just et al. 2014] and fault localization [Papadakis and Le Traon 2015], despite clear differences in the syntactic nature of real faults [Gopinath, Jensen, et al. 2014] and the fact that many real faults may not even be attributable to a single location in a program [Thung et al. 2012] With all that in mind, the second drawback ultimately constitutes a threat to validity of this approach (see secs. 4.7.1, 5.8.2, and 6.7.3).

Mutation introduces a secondary challenge, however, because it typically produces more mutants than are feasible to actually test. To overcome this challenge in my work, I sample from the population of all possible scenarios to obtain a smaller set that is feasible to run. In particular, I use a stratified random sampling approach that trades a significant amount of computational work for a modest reduction in the statistical confidence of the experimental results. The key idea of this approach is to divide the population of scenarios induced by mutation into groups, and then sub-groups, and sub-sub-groups, and so on, based on characteristics of the tests—e.g. which source program the mutant corresponds to, which mutator created the mutant, etc. Since we know that these groups each have different

characteristics, we can select a few representatives from each group to obtain a sample; it will be a diverse sample of the overall population that is representative of its variety in those characteristics, and that representativeness translates to better confidence. While this is an abstract summary, section 4.5.3 describes one such concrete stratification in detail, and all the other chapters essentially replicate the same sampling strategy.

CHAPTER 4
EXPERIMENT 1: BEHAVIORAL CONTRACTS AND BEHAVIORIAL BUGS IN CODE

This chapter demonstrates how to instantiate the rational programmer framework to evaluate the pragmatics of contracts in the context of debugging. It is an adaptation and extension of Lazarek, A. King, et al. [2020], including a full reproduction of the experiment with some significant structural changes. As such, this is joint work with Alexis King, Samanvitha Sundar, Robby Findler, and Christos Dimoulas.

The chapter begins with the essential background on contracts and blame (sec. 4.1) before instantiating the pieces of the framework (secs. 4.2-4.5), describing the results of the experiment (sec. 4.6), and discussing them (secs. 4.7-4.8).

4.1 Background: Contracts and Blame

The origins of contracts and blame in higher-order languages [Findler and Felleisen 2002] can be traced to an apocryphal story.[1] Once upon a time, a young PhD Student embarked on the mission of building a programming environment for a newly-hatched higher-order language. The road to success was (and still is) strewn with vicious bugs, and the Student fought for days, months and years to weed out as many of them as possible. Some times though, the battle was impossible to win... The Student had to deal with havoc-causing faulty callbacks and other powerful values from other people's code. All the Student could

[1] This story is a work of fiction. Names, characters, business, events and incidents are the products of the authors' imagination. Any resemblance to actual persons, living or dead, or actual events is purely coincidental.

do was labor hard to trace where the values came from, and try even harder to convince the authors' of that other code that the problem was on their end and their responsibility. After repeating this process again and again, the Student finally made a wish; "I wish there was a way to say what values others should give to my code, and if they do not comply then they get blamed!" And so contracts and blame came to be. Happily ever after, contracts caught all the stray values, blame showed where they came from, and the Student had to worry no more about which piece of code was at fault.

To understand the basics of modern higher-order contracts, consider figure 4.1. It depicts a snippet of a Racket program that calculates an infinite stream of prime numbers using the Sieve of Eratosthenes. The snippet consists of three function definitions:

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

In this example, we will refer to top level definitions as components.

Three of the definitions come with contracts.[2]

- The contract for `start`, `integer?`, states that it is an integer. In detail, this *flat* contract is a predicate, and the contract system checks that contract by checking that `start` satisfies the predicate when the definition of `start` is evaluated.

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

sieve : racket

```
;; ... dependencies omitted ...

;; `sift n st`
;; Filter all elements in `st` that are
;; divisible by `n`. Return a new stream.
(define/contract (sift n st)
  (-> integer? stream? stream?)
  (define-values (hd tl) (stream-unfold st))
    (cond [(= 1 (modulo hd n)) ;; <- a bug
           (sift n tl)]
          [else
           (make-stream hd (λ () (sift n tl)))]))

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

(define/contract start integer? 2)

;; stream of prime numbers
(define/contract primes
  (streamof (and/c integer? prime?))
  (sieve (count-from start)))
```

Figure 4.1: The Sieve of Eratosthenes, with a bug pointed out by the comment.

- The contract for `sift`, `(-> integer? stream? stream?)`, states that it is a function that consumes an integer and a stream and produces a stream. Unlike flat contracts, *higher-order* contracts like function contracts can not be completely checked immediately upon evaluating the definition of `sift`. The contract system can check that `sift` is a function (and, in Racket, that it accepts two arguments) — so-called *first-order* properties — but it is unclear how to know from the function value alone whether it will return a stream when given an integer and a stream. When the function eventually gets applied with some particular inputs, however, the contract system can check the input contracts and the function's result; in other words, the contract system needs to delay (most of the) checks for higher-order contracts.

- The contract for `primes`, `(streamof (and/c integer? prime?))`, states that it is a stream of integers that are also prime numbers. Streams are also higher-order values, because they are essentially a pair containing the head of the stream, and a function that returns the rest of the stream.

When a contract check fails, the contract system raises an error with information intended to help programmers determine the cause of the problem. As basic pieces of information, these error messages identify which contract failed, the value that failed the check, and a stacktrace describing where in the program the check failed; this information may be sufficient to determine the cause of simple contract failures (e.g. for purely first order contracts like that of `start`).

For higher-order contracts, however, that information may not be sufficiently helpful, so contracts also provide specialized debugging information called *blame*. The trouble stems from the delaying necessary to check higher-order contracts; consider that in languages like Racket, functions are first-class values, so they can flow through a program to places that have

no direct connection to the point where they are defined (or created, or acquire a contract). In large programs with many components and complex value flows, it may therefore be difficult for a programmer to understand which component actually produced the value which fails to satisfy the broken contract. To address this problem, Findler and Felleisen [2002] describe contracts as establishing obligations between the component providing a value and components that use that value. For example, `sift`'s function contract establishes the obligations that components which call `sift` are responsible for providing an integer and a stream as inputs, and it is `sift`'s own responsibility to return a stream. By tracking these obligations as the program runs, the contract system can identify which component is to blame for supplying the value causing a contract violation.

4.1.1 Debugging with Blame

Twenty years on, stories like those at the start of the section sustain a folklore belief in the potency of blame for helping programmers find software bugs.[3] Papers about higher-order contracts contain claims in the vein of "blame kicks off the debugging process in the right direction" or "blame narrows down the search for the bug", despite a lack of systematic supporting evidence (e.g. [Dimoulas, Findler, and Felleisen 2013; Dimoulas, New, et al. 2016; Strickland and Felleisen 2009b; Waye et al. 2017]). This dissertation examines whether the reputation of blame is justified in Racket's contract system.

Getting down to specifics, the contracts community has an established programming practice for dealing with blame that we call *blame shifting*. It goes like this: if a programmer is convinced that a blamed component does not contain a bug, then the programmer increases the precision of the contracts between the component and other components in an attempt

[3] See for example `https://beautifulracket.com/jsonic-2/contracts.html` (accessed November 2019).

to detect faulty values the component received. That is, the programmer attempts to *shift the blame* to some other part of the program as a means of digging down to the root cause of the problem.

This blame shifting practice serves as the main source of inspiration for the hypotheses and procedures tested in the rest of this dissertation, so it is useful to walk through an example. Consider again figure 4.1; these contracts are sufficient to uncover a bug we planted in the implementation of the Eratosthenes sieve. In detail, when we run the program and attempt to inspect the first two elements of `primes`, the contract system complains that the stream's second element is `4`, an integer that is definitely not prime. Thus it fails the `prime?` part of the contract of `primes`. Together with the information about which value failed which contract, the contract system provides *blame* information that identifies the component responsible for the problem. In this case, blame points to `primes`, which promised to be a stream of primes.

However, even a cursory inspection of `primes` suggests that the problem is not actually there. As the comment in `sift` of figure 4.1 shows, the problem is with `sift`. In contrast to what it is supposed to do, `sift` fails to remove from its `st` argument elements that are multiples of its `n` argument. Unfortunately the contract of `sift` is not precise enough to detect this discrepancy, and `sieve` does not have a contract at all. This reflects a fundamental aspect of the design of contract systems; programmers can choose the level of precision of the contracts of their components and the contract system reports only a mismatch between the contracts and the program's behavior. Hence, in the absence of precise contracts, blame points to the component whose contracts detect that it handles a faulty value. In fact, this value may have reached the blamed component from somewhere else in the code under contracts that are insufficient to detect the bug (if there are any at all). Specifically in our

example, `primes` ends up getting blamed because it has blindly trusted these two components to produce values about which `primes` makes promises in its own contract [Dimoulas and Felleisen 2011; Dimoulas, Findler, Flanagan, et al. 2011].

The above justification of blame is the source of the key insight for evaluating blame: if we make the contract of `primes` more precise, then the contract system should be able to detect the problem and give us blame information that is more accurate with respect to the location of the bug. Specifically, the contract system should detect that `primes` received a faulty value. In general terms, heeding blame and increasing the precision of the contracts in a program should eventually lead to the identification of the component that contains the bug.

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

Figure 4.2 shows three candidate contracts for `sieve` ordered by increasing precision. The first, (a.1), states properties of the tags of the argument and result of `sieve`. Of course, this is insufficient to change the behavior of the program; `sieve` does indeed produce a stream when given a stream. Thus, attempting to inspect the first two elements of `primes` with this new contract results in exactly the same contract error that blames `primes`. The second contract, (a 2), is also insufficient; the result stream does contain integers, just not the right ones.

[4]This is a subtle point of the design of Racket's contract system. Even though we refer to the contract of a component as a single entity that regulates all its interactions with any other component in a program, even those may not control, Racket's contract system pushes programmers to split the contract into multiple contracts spread across a number of components. For simplicity, however, we treat all of these pieces as the single contract of a component. In fact, recent updates to Racket's contract system have introduced a mechanism (called `contract-in`) that make this treatment more natural.

sieve-contracts : racket

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

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

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

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

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

Figure 4.2: A precision progression of contracts for (a) `sieve` and (b) `sift`.

sieve-contracts : racket

```
(define sieved-stream/c
  (stream/dc integer?
             (λ (first)
                (-> (sieved-simple-stream-following/c first)))))
(define (sieved-simple-stream-following/c sieved-n)
  (and/c (streamof (and/c integer? (not/c (divisible-by/c sieved-n))))
         (stream/dc any/c
                    (λ (first)
                      (-> (sieved-simple-stream-following/c first))))))
```

Figure 4.3: Definition of `sieved-stream/c`.

Further increasing the precision of the contract requires considering the expected behavioral properties of `sieve` beyond "type-level" descriptions. In particular, `sieve` should produce a stream where no integer in the stream is divisible by any of its predecessors. To check this property, the last contract for `sieve` in figure 4.2, (a.3), replaces the range of the previous contract, (a.2), with the custom contract `sieved-stream/c`. The contract verifies that the stream's tail contains only numbers indivisible by its head; then it attaches itself recursively to the tail of that stream, thereby building up the property that no element of the stream is a factor of any subsequent element. For the interested reader, figure 4.3 shows the full definition, which uses a custom dependent stream contract combinator `stream/dc`; `stream/dc` accepts first a contract for the head of the stream and second a function which computes a contract for the tail thunk of the stream when given the head.

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

Since an inspection of `sieve` confirms that the bug is not there, the next step is to revisit the contracts of `sift`, the component from which `sieve` receives values. The bottom part of figure 4.2 shows how we can gradually enhance the precision of the contract of `sift` to obtain a contract that, similar to the last one for `sieve`, precisely describes the expected behavior of the function. This dependent contract, (b.2), uses the function contract combinator `->i` instead of `->`. The former supports naming the arguments and result values of a function to use them to construct other portions of the contract. In this case, the contract for the result of `sift` depends on the argument `n`, and uses it to enforce that the elements of the result are

not divisible by `n`.[5] Hence it is now sufficiently precise to detect the bug and blame finally shifts to `sift`, the definition that contains the bug.

In sum, based on an intuitive understanding of blame and established practice of debugging with contracts, we have translated blame into the location of a bug. This blame-shifting process consists of following blame through the program, strengthening the contracts on each blamed component to shift the blame to another one. Eventually, the shifting stopped on the buggy component in our example.

4.2 The Blame Shifting Hypothesis

Section 4.1 describes how, based on an intuitive understanding of blame, a programmer can translate blame from contract systems into the location of a bug by shifting blame. This blame shifting process consists of following blame through the program, strengthening the contracts on each blamed component to shift the blame to another one. Eventually, the shifting stops on the buggy component.

This chapter describes a rational programmer experiment that essentially tests the hypothesis that this process is generally able to translate blame from contract systems into the location of a bug. In other words, the hypothesis is that

> blame shifting always results in blame settling on the faulty component.

To test this hypothesis, according to the outline of the method from chapter 2, we must lay out a procedure that precisely captures the blame shifting process. The next section distills the ideas of section 4.1 into an automated procedure.

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

4.3 The Blame Shifting Procedure

In informal terms, the procedure is as follows. For a given scenario, the procedure starts from the scenario's initial contract violation and attempts to repeatedly shift blame until it no longer can. Specifically, the procedure consists of the following steps:

1. run the program to get a contract violation blaming some component A;
2. try to shift blame by making A's contracts more precise, if possible, and go back to step 1;
3. otherwise, blame cannot be shifted any more, so check that A is the buggy component. Blame shifting succeeds if this is true, and fails otherwise.

The next step in the outline of chapter 2 is to construct a large scale automated experiment to test this procedure. In order to do that, however, we need to first overcome the challenge identified in section 3.1 of automating the addition of contracts (sec. 4.3.1). Subsequently, we revisit the informal procedure description and formulate it in precise terms (sec. 4.3.2) before moving on to the experiment design in the next section (sec. 4.4).

4.3.1 Capturing Contract Choices with the Configuration Lattice

Inspired by Greenman [2023], Greenman, Takikawa, et al. [2019], and Takikawa, Feltey, et al. [2016], the blame shifting process can be understood as exploring a space of *configurations* of a buggy program, which are instances of that program with a particular choice of contract for all of its components. Specifically, alongside a program P, we define a *contract map*: a mapping from each component to an ordered sequence of possible contracts for that

component. For example, for some P with components A and B, we write

$$\{A \mapsto [c_{a1}, c_{a2}, c_{a3}], B \mapsto [c_{b1}, c_{b2}, c_{b3}]\} \tag{4.1}$$

to represent the contract map where each component has three possible contracts. Those contracts are sorted in ascending order of precision, such that $c_{a1} < c_{a2} < c_{a3}$ for instance.[6] A program configuration of P is therefore represented by a mapping from each component to a choice of one contract from the corresponding sequence, such as $\{A \mapsto c_{a2}, B \mapsto c_{b1}\}$. The set of all such configurations can be partially ordered by lifting the precision ordering on individual component contracts to configurations, forming a lattice $\mathcal{L}[\![P]\!]$. The top of the lattice is the configuration mapping all components to their most precise contracts, and the bottom configuration the inverse. Figure 4.4 illustrates the lattice for the example from section 4.1 with three components.

The contract map succinctly captures the pre-baked possible contracts (sec. 3.1) that enable automating the blame shifting procedure. With the map in hand, we can trivially increase the precision of contracts around any component, thereby sidestepping the challenge of automatically generating precise contracts.

4.3.2 The Blame Shifting Procedure, Formally

With the terminology of configuration lattices, we can describe the blame shifting procedure precisely. We describe the entire process of blame shifting with a blame *trail*. A blame trail is an ascending path through the configuration lattice (i.e. a sequence of configurations) starting from any configuration in the lattice that raises an error; we call the starting config-

[6] A contract c_2 is more precise than another contract c_1, written $c_1 < c_2$, iff a program using c_2 instead of c_1 signals a contract violation whenever the program using c_1 does so. Intuitively, c_2 should check everything that c_1 checks, and possibly more.

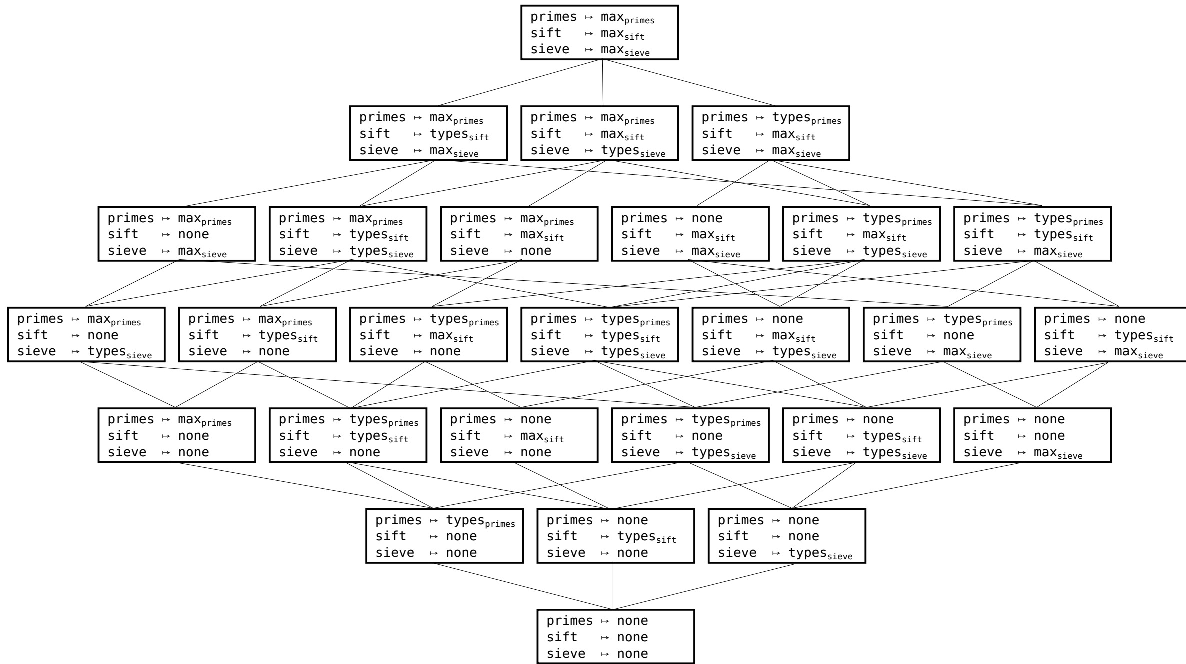


Figure 4.4: Lattice for the Sieve program.

uration of a trail a *debugging scenario*. Every configuration in a trail has exactly one *elevated component* with respect to the previous configuration, which is the component blamed by the previous configuration. An elevated component is a component that is mapped to a more precise contract in the successor configuration than in the predecessor. The elevated component may increase its contract precision by only one step from those listed in the contract map. An increase in precision of more than one step or of more than one component is not valid in a blame trail. Thus blame trails consist of a sequence of steps with one elevated component per step, thereby forming a path through the lattice. Figure 4.5 illustrates the blame trail followed in the Sieve program example from section 4.1 in these terms.

Formally, we define the blame shifting procedure as a mode of the rational programmer. There is one wrinkle in this definition that the prior section's example does not cover. It may be that none of the contracts in some debugging scenario are strong enough for the

Figure 4.5: Blame trail for the Sieve program.

contract system to detect the problem, and so running the scenario results in the program producing an error from some runtime safety check rather than a contract violation error (which has blame). In such situations, all that a programmer receives alongside the error message (e.g. "division by zero") is a stacktrace. The stacktrace is of a fundamentally different nature than blame—it points to a sequence of all the components that happen to be on the call stack of the program when it crashed—but it can be interpreted similarly (and typically is, in practice) as pointers for where to look to debug the problem. Thus the blame shifting procedure in our definition can fall back on this interpretation of the stacktrace in the absence of blame, in order to make progress.

Mode definition: Blame

A Blame trail *is a sequence of configurations* $s_0, ...s_n$ *of a program* P *such that for all* $0 \leq i \leq n-1$, $s_i < s_{i+1}$ *and*

$$elevated[\![s_{i+1}, s_i]\!] = \begin{cases} \{blame[\![P, s_i]\!]\} & \text{if (the program for) } s_i \text{ produces a contract violation} \\ \{exception[\![P, s_i]\!]\} & \text{otherwise} \end{cases}$$

where

1. $blame[\![P, s]\!]$ *denotes the component (of* P*) that the contract violation from running* s *blames, and*

2. $exception[\![P, s]\!]$ *denotes the first component in the stacktrace produced by* s *that can be elevated.*

4.4 The Experiment in Precise Terms

4.4.1 Success, Failure, and Usefulness

Blame trails as defined in 4.3.2 terminate when they cannot be extended any further, i.e. the component identified by the error in the last configuration cannot be elevated. Once terminated, trails can be either *successful* or *failing* depending on whether the component identified by that last error contains the bug or not.

> **Definition**: *A blame trail $s_0, ...s_n$ in a lattice $\mathcal{L}[\![P]\!]$ for which component $\mathcal{M}$ contains a bug is* successful *iff it cannot be extended further and error* $[\![P, s_n]\!] \equiv \mathcal{M}$ *where error* $[\![P, s_n]\!]$ *is the component identified either by blame or exception information produced by s_n.*
>
> *A blame trail $s_0, .., s_n$ in a lattice $\mathcal{L}[\![P]\!]$ is* failing *iff it cannot be extended further and error* $[\![P, s_n]\!] \not\equiv \mathcal{M}$.

While a successful blame trail indicates that it pays off to heed blame—when available—while debugging a scenario, it does not answer whether blame is a critical piece of the rational programmer's process. For instance, contract violations carry stacktrace information as well as blame, so ignoring the blame and using the simpler stacktrace information alone might be as useful as using blame. In other words, to understand the importance of blame we need to compare blame trails against a corresponding mode that ignores blame information. We therefore define a second mode of the rational programmer that uses exception information

only to establish a baseline.

Mode definition: Exceptions

An exception trail *is a sequence of configurations* $s_0, ...s_n$ *of a program* P *such that for all* $0 \leq i \leq n-1$, $s_i < s_{i+1}$ *and elevated*$[\![s_{i+1}, s_i]\!] = \{exception\,[\![P, s_i]\!]\}$.

With this baseline, we define the usefulness of blame by comparing between blame trails and exception trails that start at the same scenario s_0.

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

4.4.2 Debugging Effort

In addition to success and failure information for the different modes of the rational programmer, there is a different kind of information about trails that we can inspect to understand the blame shifting process. The length of the trails, dubbed *debugging effort*, offers a secondary metric of comparison to usefulness. For instance, effort can shed light on the difference between two modes' trails which start at the same debugging scenario and both succeed—one may reach success faster, which is obviously preferable.

Besides serving as a kind of tie-breaking comparison, measuring effort can also reveal whether the observed effectiveness of the rational programmer is an artifact of pure chance. In particular, the effort distribution for one mode can be compared with that of the random mode that ignores error information entirely and instead selects which component to elevate randomly. Since each mutant has a finite number of components, random mode trails are always successful. However, the random mode's effort distribution should be thinly spread

out across the range of trail lengths possible in the set of debugging scenarios. In contrast, the effort distribution of other modes should be quite different if their effectiveness is not coincidental.

4.4.3 Experimental Questions

Blame trails and their properties provide the tools for a rigorous examination of blame. In terms of the kinds of questions outlined in table 3.1 (page 27), this examination collects data to answer questions corresponding to the first and third columns. The second column is not relevant here, because there is only one technique under evaluation: blame from contracts.

In the first column, which in this context refers to questions about whether blame is better than its baseline, there are two related flavors of experimental question:

Q_1 Is blame information useful?

Q_2 How much more (or less) useful is blame compared to exception information?

Q_1 has a positive answer if there are any trails for which blame is more useful than exceptions. More useful here refers to the definition in section 4.3.2. Such trails constitute evidence that there exist interesting scenarios that the rational programmer manages to debug because of blame information. Hence, answering it is a simple matter of checking for the existence of any such a trail.

Answering Q_2, however, requires a more nuanced head-to-head comparison of the percentage of scenarios where blame is more useful than exceptions and the inverse. From that comparison a judgment of degree can then be made. However, the two proportions may be similar, in which case the answer to Q_2 may not be clear cut.

Drawing from the third column of table 3.1, with the debugging effort metric, provides a

way to break ties in Q_2. In particular, if the proportion of trails where blame is more useful than exceptions is similar to that where exceptions are more useful than blame, blame may still be useful if it reduces the number of modules the rational programmer needs to inspect in order to locate a bug.

In total, 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 trails for each mode of the rational programmer;
3. Compare the successes and failures of each mode's trails.

4.5 Obtaining Debugging Scenarios for Contracts

Putting the rational programmer experiment described in the prior section to work in the context of Racket demands several steps. Foremost of those is obtaining debugging scenarios—i.e. buggy programs with contract maps—on which to test the rational programmer.

Following the insight of chapter 3, we use mutation to inject synthetic bugs in correct programs. Using mutation, the process entails generating many mutants and turning those into debugging scenarios. The process must start with a suitable collection of representative programs, with contract maps (sec. 4.5.1). Then, we apply mutators to inject synthetic faults (sec. 4.5.2). That unfortunately results in too many debugging scenarios to test them all, so we sample the space (sec. 4.5.3). As a reference along the way, figure 4.6 provides an overview of the resulting experimental process.

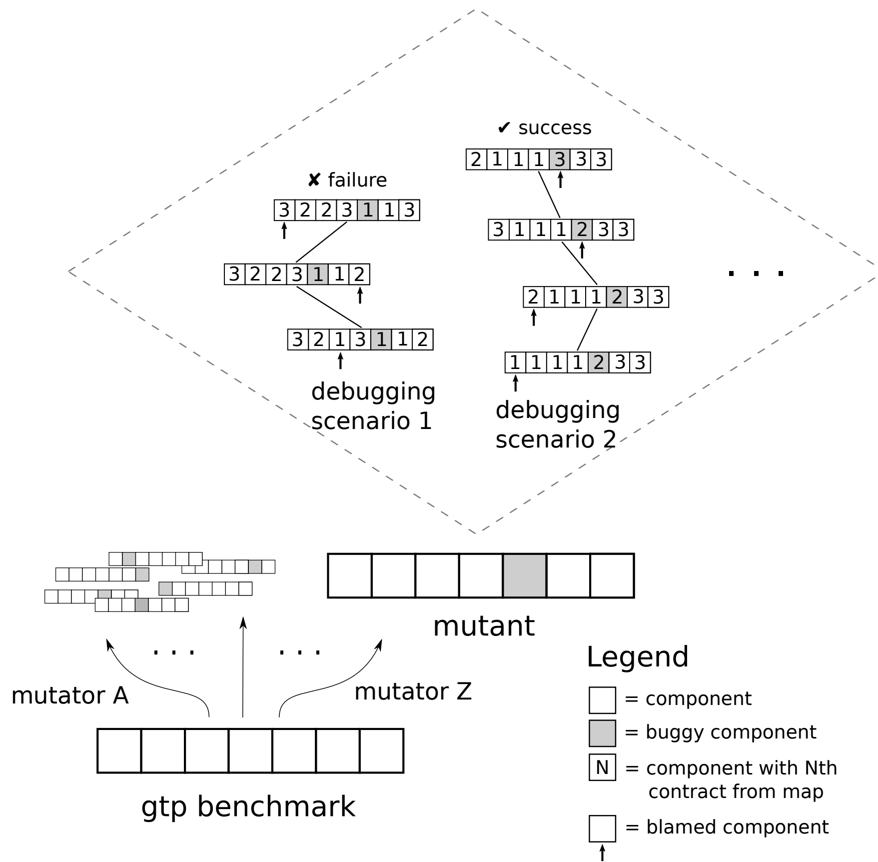


Figure 4.6: Experiment overview

Table 4.1: Benchmarks summary

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

Table 4.2: Summary of mutators

name	**description**	**example**
`constant`	swaps a constant with a similar value	0 → 1
`arithmetic`	swaps arithmetic operators	`+` → `-`
`relational`	swaps relational operators	`<` → `<=`
`logical`	swaps logical operators	`and` → `or`
`conditional`	negates conditional test expressions	`(if (= x 0) t e)` → `(if (not (= x 0)) t e)`
`statement`	deletes expressions from sequences	`(begin x y z)` → `(begin x z)`
`argument`	swaps function argument order	`(f x y)` → `(f y x)`

`hide-method`	hides public methods	`(class object%`
		`(define/public (m x)`
		`x))`
		→ `(class object%`
		`(define/private (m x)`
		`x))`

4.5.1 Starting Programs from the GTP suite

We begin with a set of seven programs from the gradual typing performance benchmarks of Greenman [2023], Greenman, Takikawa, et al. [2019], and Takikawa, Feltey, et al. [2016], selected from the full set of twenty to representatively capture the diversity of their features. Some of those programs are highly imperative, some are functional, and others follow an object-oriented design. Furthermore, the programs combine a wide range of Racket constructs such as first class functions, classes, objects, and mutable data. These programs therefore exercise a correspondingly diverse set of Racket's contract system features. Each benchmark comes with an included driver that runs the program on pre-determined inputs, which do a good job of covering the programs; according to Racket's coverage tool, the inputs achieve at least 90% expression-coverage. Table 4.1 summarizes the choice of benchmarks.

For this experiment, we define components to be the modules of the benchmarks. With this definition, the benchmarks provide configuration lattices ranging in size from 9 configurations to over 65,000. **Note**. This is a different choice than Lazarek, A. King, et al. [2020], who treat top-level definitions as components, for two reasons. First of all, Racket encourages programmers to treat modules as components, and Racket's contract system is largely designed around that perspective. Second, modules are the more standard choice among Racket programmers.

Finally, we construct the contract maps manually with three levels of precision per component. The first level, `none`, is the trivial correctness property that holds for any component; we have implemented it using Racket's `any/c`, the contract that accepts all values. The second level, `type`, captures type-like properties. For this level, since the programs originate from Typed Racket's performance evaluation benchmark suite, we have translated the Typed Racket types of their definitions into contracts. The third level, `max`, aims for partial functional correctness; it consists of the most precise specification we can express for each definition using Racket's contract DSL of combinators and predicates, and without duplicating the component's implementation as much as possible.

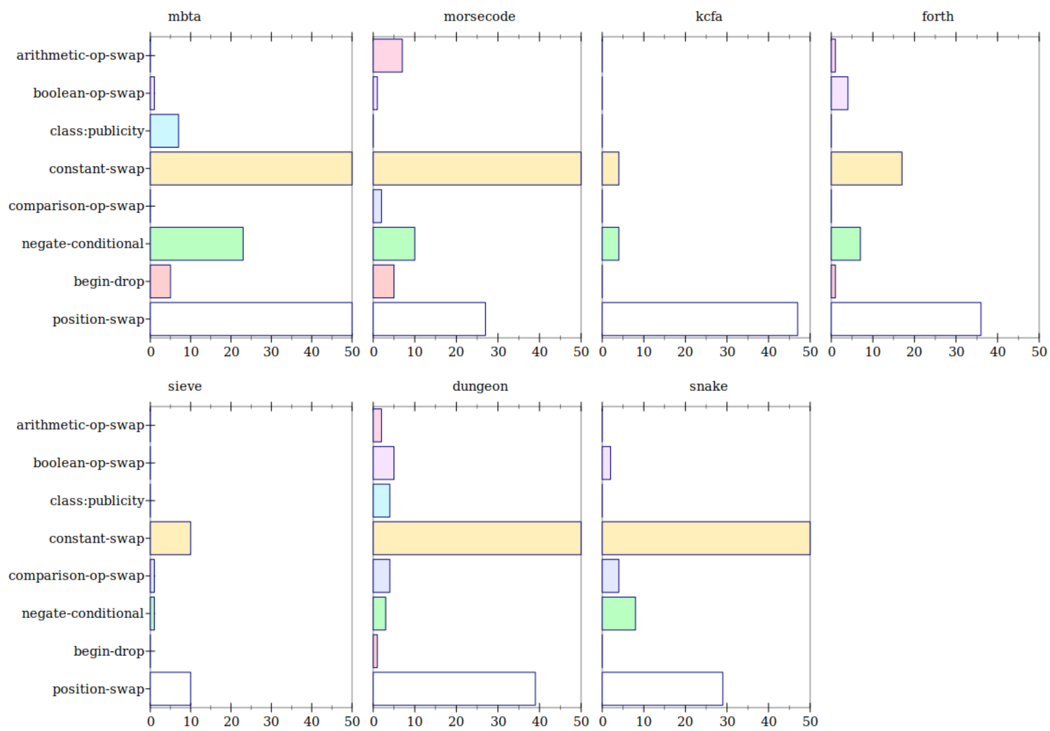
The `max` level of contracts aims to capture the strongest specification that satisfies some reasonable constraints or criteria. As such, this level does not aim to be the maximum precision contract that can possibly be implemented, nor do we require them to be so. For instance, we did not implement any contracts that use state to monitor extra-functional properties such as how many times a function is invoked. Instead, our selection of contract levels reflects our effort to understand blame in Racket when programmers take full advantage of its contract DSL to express functional specifications, and adhere to common-sense engineering and design principles. Avoiding duplication of the specified component's imple-

mentation is the most prominent such principle; although at one level it is a straightforward way to specify exactly the original, assumed-correct behavior of the component, especially when that behavior is highly nuanced, this kind of repetition is problematic and unrealistic. Such contracts offer no practical value in the real-world for bug detection—because bugs in the implementation end up in the spec—and the duplication of code creates obvious maintenance problems.

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

4.5.2 Injecting Bugs with Mutation

Following section 3.3, we draw inspiration from mutation testing to inject synthetic bugs in the GTP suite. This provides a convenient and partially automatic method to obtain a large corpus of debugging scenarios, where each has a single known bug. In this experiment, we use a standard set of mutation operators from the mutation testing literature, summarized in table 4.2. Those operators produce thousands of mutant programs that *may* make a suitable starting point for the experiment. The determining question is whether the mutant programs capture an interesting and diverse set of bugs.



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

Figure 4.7: Breakdown of interesting mutants by mutator, per benchmark.

4.5.2.1 Are These Mutators Interesting?

While mutation is well-known for producing huge numbers of mutants, a meaningful experiment requires mutants that are interesting. A mutation is *interesting* if running the mutant with all contracts removed raises a run-time error. Mutants that fail to meet this condition may not contain a bug at all (equivalent mutants); mutants that do, however, clearly change the behavior of the program, since all GTP benchmarks normally complete without errors. Thus, this condition provides a convenient filter to ensure that the experiment only involves mutants that we are certain change program behavior.

Of course, the filter is a conservative one, also removing from consideration mutants that change program behavior without causing it to crash. Since contracts may be able to detect such mutants (and raise a contract violation), this choice leaves some portion of decidably non-equivalent mutants off the table for the experiment. Section 4.7 quantifies and discusses what this choice means for interpreting the experimental results.

The definition of interesting mutants creates a powerful filter. All together, the listed mutators produce 3,329 mutants, of which 360 are interesting; see figure 4.7 for an overview. Broken down by benchmark, the mutators produce at least 50 interesting mutants for every benchmark (except `sieve`, which is quite small), and these mutants originate from at least three different mutators per benchmark. Thus, the mutators result in a sizable and diverse population of scenarios for every benchmark. Furthermore, every mutator contributes interesting mutants in at least one benchmark. Some mutators apply only to a few benchmarks, because they target rather specific features; for instance, the class-focused mutators are mainly effective in a program that makes extensive use of object-oriented features.

4.5.3 Sampling to make the experiment feasible

When using mutation to obtain debugging scenarios, the configuration lattices and number of mutants can both become so large as to make the full experiment computationally infeasible. With 360 interesting mutants and up to dozens of components in each, mutation provides nearly 500,000 interesting debugging scenarios for this experiment—far too many to try them all in a reasonable time frame. Following section 3.3, we use stratified random sampling of debugging scenarios to make the experiment feasible.

With the specifics of this experiment in hand, we can describe stratified random sampling in concrete terms. Figure 4.8 depicts an overview of the groups we use to stratify the full population of debugging scenarios described thus far. The first level of grouping that arises naturally from mutation is to group all of the debugging scenarios generated from the same source program. Then, within those program-groups, we group by the mutator that injected each scenario's bug. Within those mutator-groups, we finally group by mutant, making each final group the set of all scenarios in the configuration lattice for a given mutant. We perform standard uniform random sampling within those final groups.

Random sampling makes the experiment computationally much lighter, but it requires extra care for comparing the results across modes. Since the results for each mode now generalize with some confidence and margin of error to the entire corpus, directly comparing the aggregate results of two modes is complex to reason about. Instead, a simpler to understand and statistically sound approach is to standardize the sample of debugging scenarios (i.e. select the same sample for all modes) and make the desired comparison on a per-scenario basis. The methods of answering the experimental questions described in the prior section capture this approach.

Concretely in terms of this experiment, we perform stratified random sampling with one

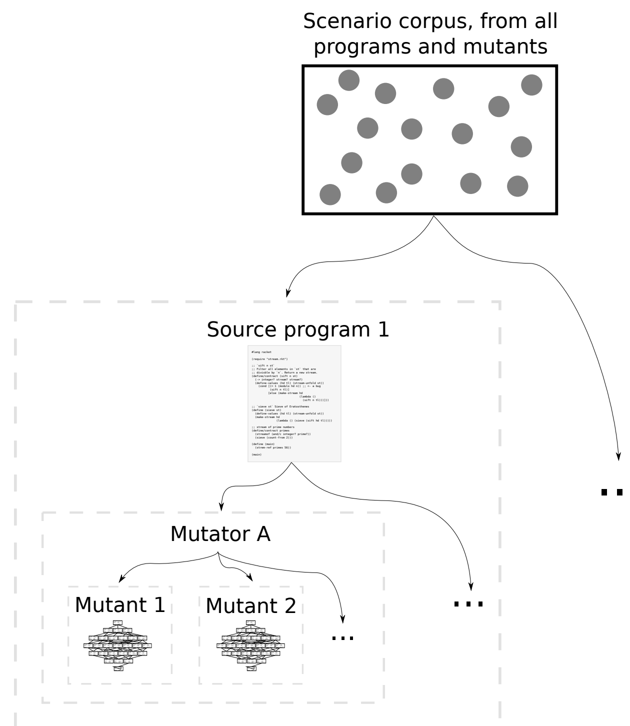


Figure 4.8: Stratification groups for stratified random sampling when using mutation.

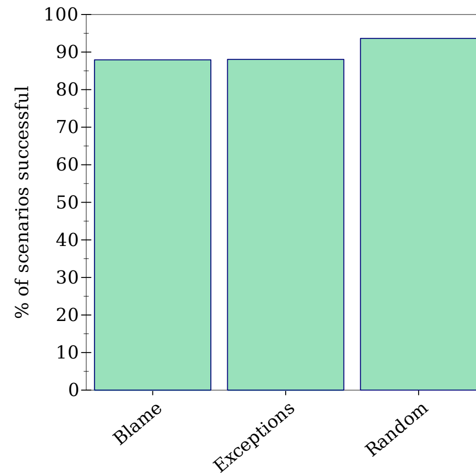
level of stratification across the population of scenarios corresponding to the 360 interesting mutants, grouping scenarios by mutant. Within each group we sample one hundred debugging scenarios to explore, and thereby obtain a sample corpus of over 45,000 debugging scenarios. This sample size is enough to obtain a confidence of 0.95 (with margin of error 0.05) about the answers to our experimental questions. See appendix A for the details of the calculations for the sufficient size of samples for this estimate. Exploring more of the lattice would yield higher confidence in the generalizability of our results to the entire population of mutants' debugging scenarios, but our choice of random sample size reflects the (informal) standard practice of estimating results to 95 percent or higher confidence.

4.6 Results

We ran 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 4.9 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. The blame and stack modes have roughly the same rate of success at just under 90% of the scenarios, and the null mode has a slightly higher rate at just over 90%. While this may at first glance seem surprising, an understanding of the causes of failed trails for each of the modes clarifies these results.

In the blame mode, the rational programmer fails to locate the bug in just under 5,000 scenarios, for two reasons. The first reason is at first glance straightforward: running the scenario results in an exception from language safety checks rather than blame. In the absence of blame, the blame mode falls back on stacktrace information to make progress. In these failing scenarios, however, the stack consists entirely of components already configured



The upper bound margin of error is 0.03%.

Figure 4.9: Percentage rates of success.

with max-level contracts. The rational programmer therefore has no indication of where to look next, so it is stuck. Unhelpful stack traces account for just over 3,800 of the blame mode's failing trails.

The remaining roughly 1,100 failures, on the other hand, all correspond to one underlying problem with Racket's contract system. In particular, all of the corresponding mutants have bugs which affect the order in which different components of the `dungeon` GTP benchmark call a function that produces a stream of numbers; the order of such calls turns out to be critical for the functional correctness of the program. For the sake of coherence, we defer an in-depth discussion of the problem to subsection 4.6.1; the interested reader can either jump to the subsection and then return, or continue to finish the overview of the results before digging into the causes of problems like this one.

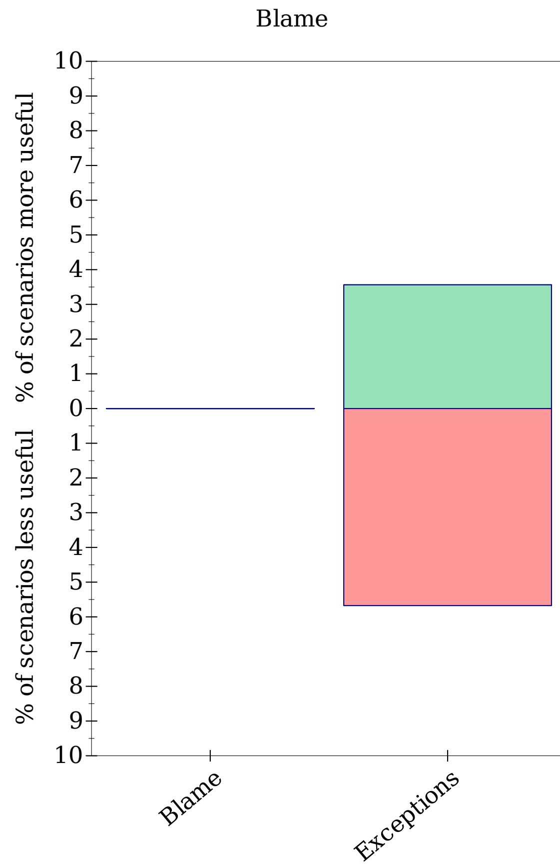
Turning now to the other two modes, it is straightforward to describe the reasons for their failures. All of the stack mode's approximately 5,100 failures are due to unhelpful stack traces of the same nature as the blame mode's first 3,800—the additional roughly 1,200 are trails where blame was available but ignored. And the null mode's approximately 2,500 failures

are captured also primarily by unhelpful stack traces in the final (top) configuration, as well as 600 trails that are thwarted by the aforementioned problem in the `dungeon` benchmark, and 131 trails that hit resource limits.

The prevalence of unhelpful stack traces thwarting even the null mode of the rational programmer in these results suggests that a significant number of the mutants in our corpus have bugs that the contract system does not detect at all, even in the topmost configuration. Indeed, that is the case for roughly two-thirds of the mutants; correspondingly, roughly two-thirds of even the blame mode's successful trails see no blame at all.

There are two plausible reasons why a mutant may cause a runtime error but not a contract violation: one is that these mutations cause the programs to crash before they affect any inter-module interactions mediated by contracts, and the other is that our max-level contracts are too weak to detect them. The first case accounts for 2,200 of the failing null mode trails. Closer investigation of the remaining trail failures reveals that they all belong to two mutants, which exhibit an interesting problem in the formulation of contracts that are strong enough to shift blame in the GTP benchmark `mbta`. That challenge is summarized in subsection 4.6.2.

In order to dig into the actual effect of blame information, the rest of the section will drill down to the one-third of the mutants (corresponding to 13,800 trails) for which the contract system is able to detect the bug. It is worth noting that the original iteration of this experiment [Lazarek, A. King, et al. 2020] did not identify these mutants, despite using essentially the same set of mutators, programs, and contracts, because the scenario construction of that experiment filtered them out a-priori; interesting mutants were defined there to be only those for which the top configuration has blame, as opposed to any runtime error.



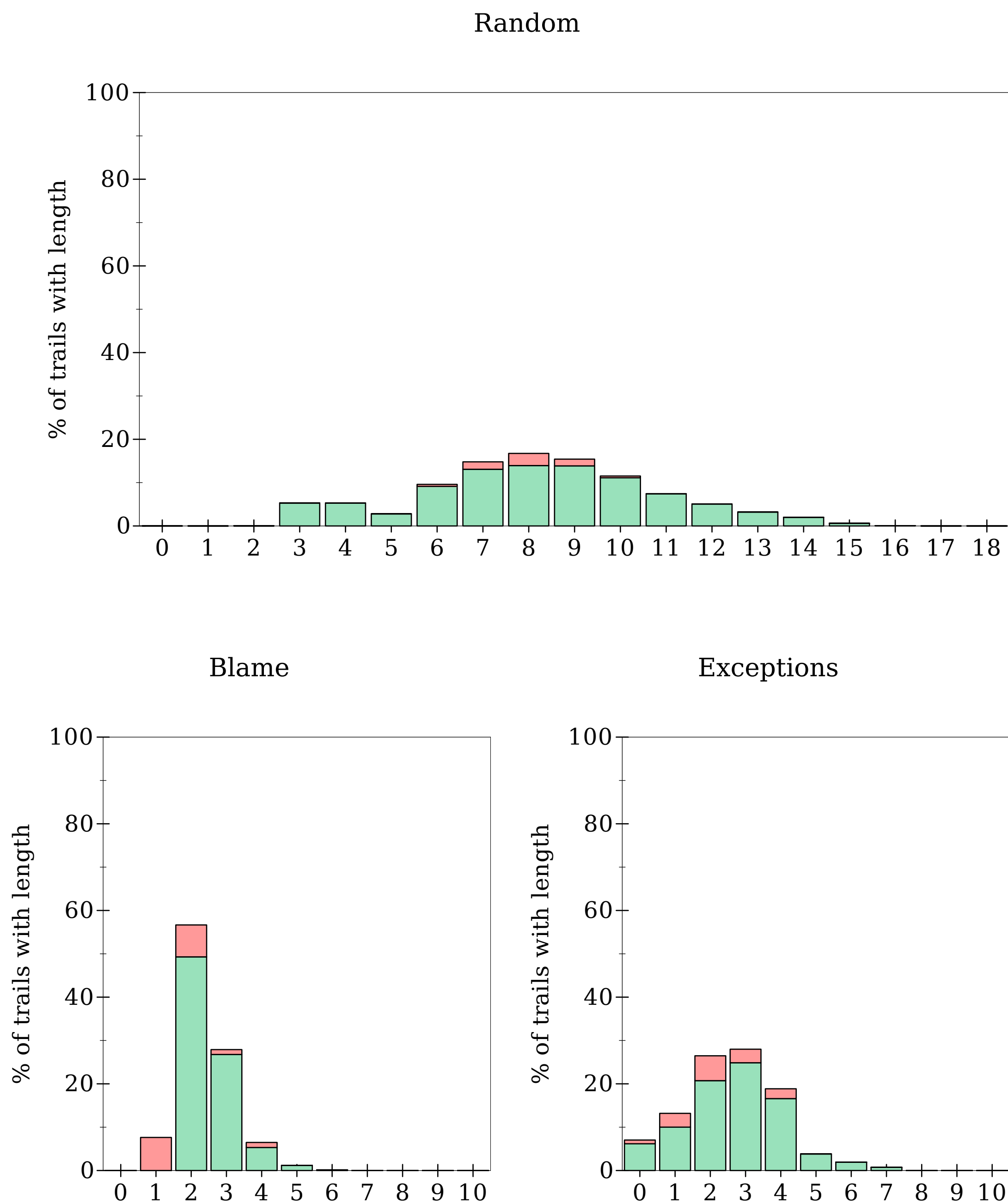
The plot compares the blame mode (named above the plot) to every other mode; in this case, the only other interesting mode is the exceptions mode (see chapters 5 and 6 for the general case of this figure structure). The green bars above 0 depict the estimated percentage of scenarios where the named mode 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 0.11%.

Figure 4.10: Head to head usefulness comparisons.

Figure 4.10 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 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; that comparison illustrates that in about 4% of the scenarios, blame provides more useful information than stack traces, but in another (different) roughly 6% of the scenarios, the inverse is true.

These results offer answers to the experimental questions from section 4.4.3. Concretely, we can answer question Q_1 in the affirmative: blame is useful. There are hundreds of scenarios where the blame mode improves over the stack mode. That said, there is a similar proportion of scenarios where the inverse is the case, all due to Racket's missing support for protocols.

The answer to Q_2 however, is not clearly affirmative based on the data of figure 4.10. The proportion of scenarios where blame improves over the stack mode is similar to the inverse. In this situation, the length of successful trails helps to clear some of that uncertainty. Figure 4.11 depicts the distribution of trail lengths for each mode, where each bar is also colored according to the proportion of successful and failing trails. There are two main takeaways from this data. The first is that Q_2 can be answered slightly in favor of blame, since it has a higher concentration of mass on shorter trails (of length 2 or less). That said, even here the answer is not entirely clear, because the stack mode does have significant portions of trails for which it succeeds in 0 or 1 steps while the blame mode has no such trails. The second takeaway from figure 4.11 is that both blame and stack information clearly have non-random influence on the blame shifting process, because the distribution of trail lengths for each differ from that of the Random mode.



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 0.24%.

Figure 4.11: Trail length distributions per mode.

4.6.1 A Weakness of Racket: Missing Protocol Contracts

To make the discussion concrete, consider the simplified program inspired by `dungeon` in figure 4.12. Its `next-number!` function produces numbers from a pre-defined sequence and functions `asks-for-2-small-numbers` and `asks-for-1-small-number` use `next-number!`'s results to call `small-number-please`. The latter requires that its arguments are less than 10, and it has a contract that captures this constraint. The first function (`asks-for-2-small-numbers`) obtains numbers from the sequence and passes them to `small-number-please` in a loop that iterates twice; it does not verify that the numbers are appropriately sized. The second function (`asks-for-1-small-number`) does the same but only once. The original version of the program completes without issue because the sequence of numbers is constructed to start with three small numbers. `asks-for-2-small-numbers` provides the first two of those to `small-number-please`, and `asks-for-1-small-number` provides the third.

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

In effect, the program assumes a protocol specifying the number of calls of `next-number!`, and Racket's contract combinators cannot express that protocol. While it is possible to write contracts that communicate using shared state to enforce the protocol, Racket's combinators

example : racket

```
(define numbers '(1 2 3 30))

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

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

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

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

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

Figure 4.12: Simple program inspired by `dungeon` that defeats blame shifting.

provide no support for specifying such properties. As a result, the bug evades the contracts of the components of `dungeon` and eventually changes the functional behavior of some component unrelated to the bug. The contracts therefore do detect the deviation from functional correctness, but the contract system cannot trace it back to the faulty component.

4.6.2 Buggy or Ill-Structured Benchmarks?

The `mbta` benchmark exhibits particular behavior that poses a challenge for this experiment. At a high level, the benchmark exhibits irregular behavior in its output, and the structure of the benchmark prevents formulating contracts that both satisfy the constraints described in section 4.5.1 and are precise enough to fully describe that behavior. In detail, `mbta` (see table 4.1) generates paths telling a user how to take the various MBTA subway lines to travel from a starting station to a destination station. To make the path more easily interpretable, the program adds commentary explicitly identifying points in the path where the user must switch train lines. Figure 4.13 illustrates an example path containing this kind of commentary. An intuitive and simple contract describing the correctness of such paths specifies (among other things) that there must be such a message between all points in the path where the specific train line changes. The actual implementation of `mbta` fails to add this commentary, however, or adds apparently spurious commentary, under certain edge conditions. Critically, capturing those conditions requires essentially reproducing the private internal code that produces the commentary. As a result, `mbta` fails to live up to the intuitive contract, and a contract that is precise enough to specify its actual behavior violates our design constraints—namely, the requirement that contracts do not reproduce the computation they specify. The contracts we use for the experiment resolve this tension by specifying the commentary behavior at a high level, but weakly enough to be satisfied by

```
station A, take blue
station B, take blue
--- switch from blue to red
station C, take red
...
```

Figure 4.13: The shape of paths generated by `mbta`.

the benchmark's original behavior. Unfortunately a consequence of this weak specification is that there is a class of mutants that change the commentary behavior of `mbta`, causing test failures, but no amount of contract strengthening allows blame shifting to locate the injected bugs.

There are at least two reasonable perspectives for interpreting this problem. One interpretation is that `mbta` exhibits a problematic structure for thorough specification with contracts, where the behavior of a component may not support a high-level description of correctness independent from its implementation. That is, specifying the correctness of some components may truly demand reproducing some or much of the component's implementation itself, and/or restructuring the original program to expose private details. Because this kind of duplication and exposure is obviously problematic (see the end of sec. 4.5.1), the demand for it may be an indication that the component itself is poorly designed; if there is no high level, self-contained description of its correctness, then perhaps it represents a failure to design a good abstraction. In the case of `mbta` in particular, the complexity and irregularity of the commentary behavior, as well as the restrictiveness of the interface by which it is exposed to the rest of the program, support this perspective. In more detail, `mbta`'s annotation behavior is exposed to the program by a component with a bare-bones interface that transforms strings (start and destination station names) to a string (describing the path)—using the private, internal representation of the subway graph to compute the path and then

add appropriate annotations. That component is wrapped, however, in another one that also maps strings to strings by delegating to the first component, creating a scenario where the outer component lacks the information necessary to formulate a contract strong enough to support blame shifting without duplicating the inaccessible annotation computation. In simpler cases than `mbta`, however, this pattern of duplication between contracts and code for small and simple components may also indicate an opportunity for improvement in contract system design, in creating a way for contracts to be specified as part of or integrated with a component's implementation in a way that minimizes duplication.

Another interpretation of the problem is that the irregularity in `mbta` represents a bug in the program. One could reasonably argue that the program ought to live up to the aforementioned intuitive contract, and the fact that it doesn't directly indicates a problem with the program. From this perspective, the choice to manually write contracts for the benchmark programs provides an unforeseen additional benefit: the process serves as a check for the correctness of the original programs (which the experiment assumes). Indeed, this perspective could just as well be applied to the protocol problem described in the prior section; a reasonable judgment of figure 4.12's program (and the original benchmark) is that the users of `next-number!` are buggy because they fail to check or ensure the appropriateness of the number they supply to `small-number-please`. Ultimately, whether these problems indicate bugs or something else depends on perspective, but the discovery of these potential-bugs is in many ways a validation of the experimental design as a whole. This is a useful check in light of the experimental design's assumption that mutated programs contain only a single bug. Indeed, blame shifting in the context of multiple bugs cannot be expected to locate a particular one of those bugs, for blame may reasonably settle on any one of them.

4.7 Lessons Learned

The experimental results suggest a few takeaways about the value of blame in Racket's contract system. First, blame is useful for locating bugs via the process of blame shifting. However, plain stack trace information appears to be just as useful on the whole.

That said, the causes of failure for the blame mode of the rational programmer bring up a useful direction for improvement of Racket's contract system. In particular, the failures exemplify an expressiveness problem for Racket contracts. Racket's contract combinators cannot express protocols like the one identified in `Dungeon`, even though they are quite common in real programs. For example, file system APIs implicitly come with protocols about when operations can be applied to a file, and in what order: an open file cannot be re-opened, a closed file cannot be read or written to, and so on. Protocols do not only describe temporal properties, but also other restrictions on the proper use of components, such as security. Thus, protocols are a natural extension for Racket's contract system. In general, there have been some steps towards protocol contracts [Dimoulas, New, et al. 2016; Disney et al. 2011; Heidegger et al. 2012; Moore, Dimoulas, Findler, et al. 2016; Moore, Dimoulas, D. King, et al. 2014; Scholliers et al. 2015], including recent work adding them to Racket in the time since the original publication of this experiment [Moy and Felleisen 2023].

4.7.1 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 bugs may not be truly representative of all mistakes programmers make; and

(iii) our manually-written selection of contracts for the experiment may not be representative of all contracts that programmers write. While the design of the experiment attempts to mitigate these threats with the careful design and analysis of the scenario generation (sec. 4.5), 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 contracts and blame.

4.7.2 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 or R6RS Scheme, two languages with highly rigorous, extensive formal definitions [Milner, Harper, et al. 1998; Milner, Tofte, et al. 1990; Sperber et al. 2009]. Each 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, Pierce, 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 thesis 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 4.6 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 contracts, and it does so at scale and in a quantifiable manner.

4.8 Summary

This chapter describes a straightforward application of the rational programmer framework to evaluate blame in Racket's contract system. The results of that experiment support three different conclusions. First, they confirm that blame provides useful information for locating bugs via blame shifting. Second, they highlight a key weakness of Racket's current contract system (protocols) and emphasize the value of addressing it. Finally, they cast doubt on the value of blame overall, because stack trace information appears to be just as useful for locating bugs.

As a first application of the rational programmer framework, these results are promising. They offer new insights into the pragmatics of contracts in the context of debugging. A next natural question is how the method applies to the pragmatics of debugging in the neighboring field of gradual typing, which employs a range of checking techniques—some based on contracts, and others completely different. The next chapter answers this question.

CHAPTER 5
EXPERIMENT 2: GRADUAL TYPES AND TYPE-LEVEL BUGS IN CODE

This chapter demonstrates how to instantiate the rational programmer framework to evaluate the pragmatics of gradual typing in the context of debugging. In this setting, there are multiple competing semantics for the language, each providing its own kind of blame or other debugging information, which we compare head-to-head. Because several of the ingredients for this experiment are essentially the same as the prior chapter, this chapter focuses on describing the differences and new challenges involved in this one. The chapter is an adaptation of Lazarek, Greenman, et al. [2021] and joint work with Ben Greenman, Matthias Felleisen, and Christos Dimoulas.

The chapter begins with the essential background on gradual types (sec. 5.1) before describing the key challenges associated with this new setting (sec. 5.2), instantiating the pieces of the framework (secs. 5.3-5.6), describing the results of the experiment (sec. 5.7), and discussing them (secs. 5.8-5.9).

5.1 Background: Gradual Types

Gradual typing is an approach to merging the worlds of static and dynamic typing in the hopes of getting the best of both worlds. In the world of static typing, programmers annotate their program with type annotations describing the type of data it uses and produces. Then, a type checker checks that the annotated program does not have any inconsistencies, such as applying a function with the type `Int` → `Int` to a value of type `String`, and rejects any program it cannot prove to be consistent. Thus the annotations serve as a form of

documentation that is directly tied to the code, and the type checker provides a measure of early error detection before the program runs. Furthermore, IDEs and compilers can leverage the type information from annotations to provide improved tooling (like type-directed code completion) and optimizations (based on knowledge of types' memory representation [Chou et al. 2018; Dillig et al. 2011; Essertel et al. 2019; Lattner and Adve 2005; McMichen et al. 2024; Walker and Morrisett 2000; Wang et al. 2018]).

Despite the benefits of types, dynamically-typed languages remain extremely popular among programmers for a variety of reasons. The restrictiveness that makes static type checking useful also rules out many correct programs, simply because the type checker isn't sophisticated enough to prove their consistency. Dynamic languages allow programmers the freedom to write such correct programs without worrying about how smart the type checker is. Along the same lines, dynamic languages allow for faster prototyping because programmers need not satisfy the type checker at every step of the way. This flexibility can also make extending existing code easier for the same reason.

Gradual typing was born from a recognition that both static and dynamic typing offer different kinds of benefits, so it could be useful to have the best of both worlds [Flanagan 2006; Gray et al. 2005; Knowles and Flanagan 2010; Matthews and Findler 2007, 2009; J. G. Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006].

Gradual typing attempts to achieve this fusion by allowing programmers to partially annotate programs, mixing and matching statically and dynamically typed code as they please. One approach to make this a reality is to make the type checker more flexible. Specifically, gradual type checkers are augmented to allow untyped code, assigning it a special "dynamic" type, and to optimistically allow the interactions of statically typed code with dynamic code by making the dynamic type compatible with all other types. For instance, a gradual type

checker would allow a function of type `Int` $\rightarrow$ `Int` to be applied to a dynamically-typed value, because it's possible that the dynamically-typed value will be an integer. Thus the gradual type checker optimistically allows untyped code to mix with typed code. An alternative approach, that of Typed Racket [Tobin-Hochstadt and Felleisen 2008], does not support interactions via a dynamic type, but instead requires developers to choose between static and dynamic types at the granularity of whole modules; typed modules can import untyped code via explicit developer annotations describing its expected type, which the type checker trusts.

Type systems are meant to preclude certain classes of mistakes, however, such as applying a function to values of the wrong type. Indeed, many of the benefits of static typing arise specifically from this preclusion (e.g. early error detection, optimization), so the gradual-type-checker's optimistic judgments about untyped code may benefit from some kind of enforcement at runtime in order to maintain the benefits of static types. In other words, gradual programs could have checks to catch mismatches between the expectations of typed code and the actual values it receives at runtime. By catching the mismatches with targeted checks, the gradual type system can both protect the benefits of static types and provide information to help the programmer locate and fix the mistake causing the mismatch.

These type-value mismatch checks can be implemented in many ways, or not at all, even all within the same exact type system, giving rise to a multitude of different designs, called *semantics*, for gradually typed languages. The various choices available affect the guarantees available to programmers about types [Greenman and Felleisen 2018; Greenman, Felleisen, and Dimoulas 2019], the performance of gradually typed programs [Campora, Chen, and Walkingshaw 2018; Greenman and Felleisen 2018; Greenman and Migeed 2018; Greenman, Takikawa, et al. 2019; Takikawa, Feltey, et al. 2016; Vitousek, J. G. Siek, et al. 2019],

and the debugging information available when type-value mismatches occur [Greenman, Felleisen, and Dimoulas 2019; Vitousek, Swords, et al. 2017]. Furthermore, these various concerns are interconnected, making it so that all of the semantics select different trade-offs between them, and even making it potentially useful to combine semantics within a single language [Greenman 2020].

Faced with all of the semantics for gradually typed languages, how can language designers pick between them? Is any way of checking for type-value mismatches more useful than others for debugging such problems? While theoretical work clearly distinguishes some of the approaches as offering stronger guarantees and tailored information for debugging than others, performance analyses show that those benefits come at great cost [Greenman and Felleisen 2018; Greenman, Felleisen, and Dimoulas 2019; Takikawa, Feltey, et al. 2016]. Is the price worth paying for the debugging benefits?

The next two subsections illustrate the major differences between the semantics from the lens of this question, demonstrating how how each affects the error information they provide in the face of two quite different kinds of bugs.

5.1.1 Three Flavors of Gradual Typing

In this work, we focus on the three most prominent approaches to enforcing static types in gradual typing. These can be divided into two groups: academic and industrial. Academic implementations of gradual typing consider the meaning of type annotations important, so they insert dynamic checks enforcing the annotations at run time, and emphasize the safety properties that these semantics offer. The *Natural* semantics translates types into contracts protecting typed code in all interactions with untyped code (and not in the interactions of purely typed code, since those are already checked by the type checker) [Tobin-Hochstadt

and Felleisen 2006]. An alternative academic semantics called *Transient* enforces types by directly translating type annotations into checks in all typed code [Vitousek, Swords, et al. 2017]. Industrial implementations (e.g. Flow, Hack, or TypeScript[1]), on the other hand, almost universally forego enforcement of types. These languages use the *Erasure* semantics, in which the type annotations exist for best-effort type checking and IDE tooling, and the implementation's compiler erases the types after type checking as if they did not exist. In addition to the differences in checking, each of these semantics offers completely different error information to help programmers debug type-value mismatches.

The remainder of this section summarizes the differences between the Natural, Transient, and Erasure semantics with one illustrative example using (Typed) Racket syntax.

Consider the program sketch in figure 5.1. Each box represents a module: the top bar lists the name and whether it is using typed (blue) or untyped (red) syntax.

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

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

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

`crypto-pack-lib` (at the bottom left) also imports `pack` and ascribes it the same type as

[1] See `https://flow.org`, `https://hacklang.org`, and `https://www.typescriptlang.org`, respectively.

types : typed/racket

```
(provide Entry Entries)

(define-type Entry
  (Pairof Symbol String))

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

typed-pack-lib : typed/racket

```
(provide typed-pack)

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

(define typed-pack pack)
```

crypto-pack-lib : typed/racket

```
(provide crypto-pack)

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

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

pack-lib : racket

```
(provide pack _ _ _)
(require types)
_ _ _  dependencies     _ _ _
_ _ _  and definitions  _ _ _
(: pack (-> JSON Entries))
(define (pack d)
  ;; process JSON data and
  ;; package as a dictionary
  ;; (association list)
  (make-hash _ _ _)) ;; BUG!
```

client : racket

```
(require json)
(require typed-pack-lib)
(reqired crypto-pack-lib)
_ _ _ other dependencies _ _ _
_ _ _ and definitions    _ _ _
;; read data from files, pack
;; and share securely

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

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

 _ _ _ rest of client _ _ _
;; (length public-data)
;; (length secret-data)
```

Figure 5.1: One mixed-typed program, three interpretations.

typed-pack-lib. It applies the function in the definition of the exported crypto-pack function, which encrypts its input before passing it to pack.

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

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

As this analysis implies, if a return-type check fails, the problem is that the untyped module, here pack-lib, supplied a function that is not a match for the type ascribed by the typed module. Hence either the type at the boundary between the two modules is wrong or, if the programmer trusts the type, the untyped module is at fault. If the check of an argument's type fails, responsibility lies with client. After all, either the type it ascribes to the argument is wrong or the argument it produces clashes with the type. Due to proxies, *Natural* can easily track the boundary, type, and responsible parties that correspond to each check. Thus, in the example of figure 5.1, as pack returns, the return-type check fails and

Natural blames the boundary of `pack-lib` with `typed-pack-lib` and `crypto-pack-lib`, respectively, for the two `define`s in `client`.

Under *Transient*, typed code is compiled so that all entry points to functions check their arguments at run time and all function calls check their return values against the expected type [Vitousek, Swords, et al. 2017]. Furthermore, Transient uses *shallow* checks, meaning they inspect only the top constructor of a value [Greenman 2020]. Since retrieving a value from within a structure (or list, array, hash table etc.) is performed via a function call, the content of a complex value is checked on a piecemeal basis.

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

In order to locate the corresponding boundaries for failed checks, Transient maintains a map from values to the boundaries between typed and untyped modules that they cross, plus the corresponding types. In the example, the map records that `pack` crosses from `pack-lib` to `typed-pack-lib` and from `pack-lib` to `crypto-pack-lib` with the type that appears in the `required/typed` forms in the example. Since the failed check corresponds to the return type of `pack`, assuming that the type is correct, the responsible party is the source of the two boundary crossings: `pack-lib`. In general though, Transient blames more than one boundary. In fact, the theoretical work of Greenman, Dimoulas, et al. [2023] and Greenman, Felleisen, and Dimoulas [2019] shows that for some programs Transient constructs a blame sequence that excludes responsible parties and includes modules irrelevant to the failing check.

Under *Erasure*, the compiler checks the specified types and then discards them when

Table 5.1: Summary

	`public-data`	`secret-data`
Natural	error, blaming the boundary between `pack-lib` and `typed-pack-lib`	error, blaming the boundary between `pack-lib` and `crypto-pack-lib`
Transient	no error	error, blaming the boundaries between `pack-lib` and `typed-pack-lib`/ `pack-lib` and `crypto-pack-lib`
Erasure	no error*	no error*
	*but *Erasure* does signal an error on list access	

it generates code. The generated code includes whatever checks the underlying language uses for its run-time system. Hence, in the Racket code of the example, neither the call to `typed-pack` nor the call to `crypto-pack` signals an error due to the gradual type system. If at some later point `client` tries to inspect the elements of the lists that `typed-pack` and `crypto-pack` are supposed to produce (such as in the commented code at the end), Racket's safety checks signal a violation and point to some place in `client`. The information in this exception, plus its stack trace, may help the programmer find the source of the type-value mismatch between the specified types of `pack` in the two typed modules and its actual results.

Table 5.1 summarizes the illustration. Each cell describes the result of evaluating the column's definition (in `client`) under the row's semantics.

5.1.1.1 Debugging with Gradual Blame

How can a type-value mismatch error provided by Natural, Transient, or Erasure help a programmer locate the mistake? To simplify the question, let us first assume that all type annotations in a program are correct, so the mistake can only lie in the code of a component. In the next chapter, we consider the alternative where type annotations themselves contain mistakes.

In the context of gradual typing, a programmer has two pieces of information when a type-value mismatch signals exceptional behavior: the error message and the state of the program. Hence, a way for a programmer to make progress is to use the available information from the error to improve the program. Specifically, the programmer can translate the Wadler–Findler slogan into a debugging method, searching for the source of the type-value mismatch by adding type annotations to some of the untyped parts of the program identified in the error. If the type checker rejects an annotation derived from the context, the rational programmer has found the source of the problem. Otherwise, the programmer can conclude that the just-annotated parts are not the problem and re-runs the program—which must, by the slogan, blame a different location for the problem. At this point, the programmer can iterate the process.

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

Now consider the concrete (and simplistic) example of figure 5.2. Initially the code base consists of the two red modules on the left plus the blue module at the bottom; red indicates untyped, while blue means typed. When a typed module imports an untyped module, it must assign types to the imported identifiers for the type checker's sake. Here `main` specifies

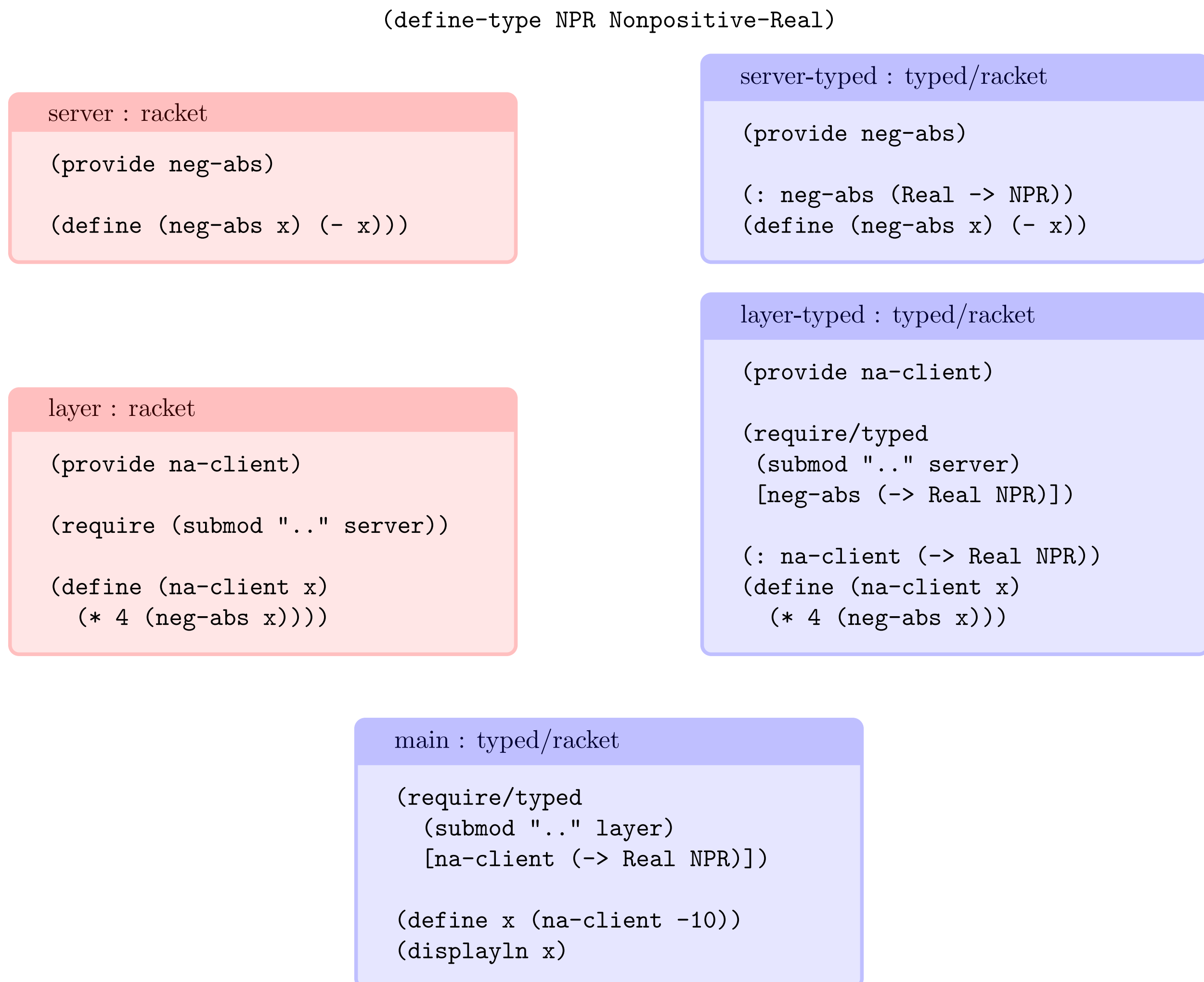


Figure 5.2: A simplistic debugging scenario.

that `na-client` consumes a `Real` number and produces a non-positive one.[2] A program execution ends in this error:

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

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

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

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

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

```
blaming: (interface for neg-abs)
 (assuming the contract is correct)
```

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

In sum, the design of blame-assignment mechanisms explicitly advertises the blame information as helpful for debugging type-value mismatches. The error messages of blame-assignment mechanisms include suspect locations at the boundary of typed and untyped code fragments. The Wadler–Findler slogan suggests that the source of the problem is concealed due to a lack of types, so adding types to the untyped fragment should lead to the source of the type-value mismatch.

5.2 Challenges

Instantiating the rational programmer method in this setting poses two new challenges. The first concerns the comparison of the effect of blame on the rational programmer across three different mechanisms; the second challenge is about finding a large number of representative debugging scenarios; and the third is the resulting huge space of possibilities. A coincidental challenge is the need for distinct and diverse implementations of gradually typed languages. We therefore use Racket, which is the only language in which all three major semantic variants are available in a robust and comparable manner [Greenman, Lazarek, et al. 2022]: Typed Racket implements Natural, Shallow Racket implements Transient, and plain Racket implements Erasure.

The *first challenge* stems from the differences between the blame assignment mechanisms

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

We reconcile these differences within the rational programmer framework using modes that represent the different types of information the rational programmer takes into account when debugging a scenario (chap. 3). Intuitively, different blame strategies correspond to different modes of operation. For instance, one Transient mode may assign types to the oldest element of a blame sequence because it corresponds to the earliest point in the execution that can discover a type-value mismatch. Another mode may opt to treat the sequence as a stack and add types to its newest element. If both modes are equally successful in locating a type-value mismatch, measuring the rational programmer's debugging effort with each mode may answer which is the most effective.

The *second challenge* is to find a representative, curated collection of programs with type-value mismatches. The type-value mismatches must represent mistakes that programmers accidentally create and that the run-time checks of academic systems catch. In other words, the experiment calls for a collection of mistakes in mixed-typed programs that is representative of those "in the wild." Unfortunately no such collection exists, and with good reason. The kind of mistakes needed are typically detected by unit or integration tests; even if it takes some time to find their sources, these mistakes do not make it into code repositories with appropriate commit messages.

Following the prior chapter, we use mutation analysis to generate a suitable corpus of programs, but conventional mutation analysis is useless. Mutation analysis traditionally aims

to inject bugs that challenge test suites, and it discards those that yield ill-typed mutants as *incompetent*. Indeed, mutation analysis frameworks are fine-tuned to avoid them, and yet, it is precisely those mutators that are needed for evaluating blame assignment strategies.

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

Hence, an experimental analysis of blame in this setting needs a mostly new set of mutators. Roughly speaking, the new mutators inject type errors into fully typed programs. Applying such a mutator to any typed component produces a mutated component. A debugging scenario results from removing the types from the mutated component. For the design of such mutators, the authors relied on their own extensive programming experience though not without discovering a major pitfall: some of their original mutators systematically produced programs that immediately revealed the source of the type-value mismatch. All of the remaining ones yield *interesting debugging scenarios* (see sec. 5.6.3).

The next three sections explain how to overcome these challenges within the framework of sections 2 and 3.

5.3 The Hypothesis for Gradual Types

Section 5.1.1.1 describes how, based on an intuitive understanding of blame, a programmer can translate errors from a gradual type system into the location of a bug by adding type annotations. This process consists of following error-provided hints through the program,

adding types to the components guided by the error information from the system. Eventually, the new annotations allow the type checker to discover the problem statically.

This chapter describes a rational programmer experiment that tests the hypothesis that this process is generally able to translate gradual typing error information into a static error. Specifically, the hypothesis is that

> for a program containing a type-level mistake in the code, adding type annotations guided by the error information available reliably leads to a static type-checker error.

To test this hypothesis, according to the outline of the method from chapter 2, we must lay out a procedure that precisely captures the debugging process. The next section distills the ideas of section 5.1 into an automated procedure.

5.4 The Procedure for Gradual Types

Section 5.1.1.1 explains how a migratory type setting helps with finding the source of a type-value mismatch. Roughly speaking, it encourages the rational programmer to equip a module with types if it is blamed in an error message.

In other words, and at a high level, the procedure is essentially the same as that of the prior chapter, but using types instead of contracts. It uses blame to identify a component, and adds a specification for that component in the form of type annotations. However, in this setting, these new type annotations may then allow the type checker to detect the problem statically, providing a new way for the procedure to terminate in success.

Putting these pieces together, the new procedure is:

1. run the program under one of the semantics to get an error identifying some component

A;

2. try to annotate A with types, turning it into a typed component if possible, otherwise A is already typed: terminate in failure;

3. type-check the resulting program—if it type checks, go back to step 1;

4. otherwise, we now have a static error: terminate in success.

Following the prior chapter, we define modes that precisely describe this procedure as tracing a path through a lattice of type migration.

5.4.1 The Type Migration Lattice

Like chapter 4, we follow Greenman [2023], Greenman, Takikawa, et al. [2019], and Takikawa, Feltey, et al. [2016] to describe the set of all possible type migrations with a lattice. The lattice describes the space in which the modes of the rational programmer search for bugs. Unlike for contracts, however, the lattices in this setting differ from those in the preceding chapter in two ways.

First, the "contract map" in this setting maps each component of a program to just two levels: untyped, or typed. In this simpler setting, we can equivalently describe configurations as the set of components that are typed; the bottom configuration, which before we described by a map from every component to the untyped level, is more succinctly described here as the empty set, and the top configuration as the set of all components in the program. For the remainder of this chapter we use this equivalent, simpler notation for configurations.

The second difference is that for all debugging scenarios in this setting, the mutated module is ill-typed. So at all configurations that type the mutated module, the type checker

immediately points out the type-level mistake in the mutated component. Those configurations are a-priori uninteresting as debugging scenarios, because there is no work to be done by our procedure, so we rule them out.

5.4.2 How to Make Comparable Rational Programmers

Each mode in this setting receives different kinds of information and thus may construct different paths in the lattice. As section 5.2 outlines, evaluating blame relies on comparing modes of the rational programmer within the same semantics and across different semantics. Hence the task at hand is to define modes that correspond to different semantics and process different kinds of information, but all operate within the common structure of blame trails.

5.4.2.1 The Natural Rational Programmer

The Natural semantics assigns blame to exactly one boundary. A blame assignment has the following specific meaning: the typed module may make incorrect type assumptions about the untyped module in its interface, or the correct interface exposes a bug in the untyped module (or its dependencies). Our setup rules out the first alternative (but see sec. 5.9), and therefore the rational programmer extends the trail to a scenario that swaps out the untyped module for its typed counterpart.

The definition of the mode that uses Natural's blame therefore closely mirrors that of section 4.4. The only difference is that in this setting, we parameterize the blame and

exception metafunctions with the semantics used by the mode.

Mode definition: Natural blame

A Natural blame trail *is a sequence of scenarios* $s_0, ...s_n$ *of a program* P *such that for all* $0 \leq i \leq n-1$, $s_i \subset s_{i+1}$ *and*

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

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.*

5.4.2.2 The Transient Rational Programmer

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

This ambiguity in Transient blame raises the question of how the rational programmer should react when the language produces a blame sequence. Our answer is that the rational programmer has at least two reasonable options. The first one is to select the untyped module that is added to the blame sequence first and assign types to only that one—after all, if fully checked, the types of this first module should be able to detect a type-value

mismatch earlier in the evaluation of a program than the later ones. The second option is to select the module that is added to the blame sequence last, effectively interpreting the blame sequence as a boundary-aware stack.

These two modes of rationalizing give rise to two different notions of trail.[3]

Mode definition: Transient first blame

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

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

where

1. $\mathit{first}\,[\![\mathit{multiblame}\,[\![P, s]\!]]\!]$ *is the first untyped module that Transient adds to the blame sequence for* s *under the Transient semantics, and*
2. $\mathit{exception}_{\mathit{Transient}}\,[\![P, s]\!]$ *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 $\mathit{multiblame}\,[\![P, s_i]\!]$ that Transient adds to the blame sequence rather than the first.

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

5.4.2.3 The Erasure Rational Programmer

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

> **Mode definition**: Erasure
>
> *An* Erasure trail *is a sequence of scenarios* $s_0, ...s_n$ *of a program* P *such that for all* $0 \leq i \leq n-1$, $s_i \subset s_{i+1}$ *and* $s_{i+1} \setminus s_i = \{exception_{Erasure}\,[\![P, s_i]\!]\}$.

5.5 The Experiment in Precise Terms

5.5.1 Success, Failure, and Usefulness

As in the preceding chapter, the actions of the rational programmer create a blame trail in $\mathcal{L}[\![P]\!]$ starting from a debugging scenario. However, a trail in this setting ends successfully when it reaches a scenario that contains the mutated module, because the type checker rejects its typed version outright. At this point, the source of the type-value mismatch is identified. Hence, a trail that ends at an ill-typed scenario successfully pinpoints the location of the bug.

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

That is, failing Natural blame trails are those that end in a scenario that does not reveal the

bug statically, yet also does not blame an untyped module. Thus the rational programmer has no further hints on how to continue the search for the bug.

Following section 4.4, we also define a Natural exceptions mode to serve as a baseline against which to judge the value of Natural's blame. This mode uses the Natural semantics, so it has all the same checks as the Natural blame mode, but ignores blame information and only uses stacktraces.

Mode definition: Natural exceptions

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

The definition of success for a Natural exception trail follows that for a Natural blame trail. Together, the definitions for the two modes allow the comparison of the usefulness of blame with that of mere exceptions for debugging a scenario in the context of Natural semantics in exactly the same way as in section 4.4.

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

Similar to the Natural rational programmer, we define *Transient exception trails* to 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.*

The definition of success/failure and usefulness of the two interpretations of Transient blame are obvious adaptations of the definitions for Natural, so we omit them here.

5.5.2 Experimental Questions

In line with the discussion so far, the examination collects data to answer three initial questions for interesting debugging scenarios:

Q_1 Is blame useful in the context of Natural?

Q_2 Is first blame useful in the context of Transient?

Q_3 Is last blame useful in the context of Transient?

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

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

In terms of the space of experimental questions of table 3.1 (page 27), Q_1 through Q_3 capture the first column of questions, and Q_* the second column, with information from the third column (i.e. debugging effort) being a useful tie-breaker.

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

Figure 5.3: Experimental questions and their relevant modes.

Now that there are both multiple techniques and modes in play, it is a little trickier compared to chapter 4 to keep track of which are involved in answering which experimental questions. Table 5.3 summarizes how each question relates to different kinds of trails/modes of the rational programmer. For example, experimental question Q_1 asks whether blame is valuable for Natural and the experiment uses the Natural blame and exception trails to answer it, so Q_1 shows up in the cells for Natural blame and Natural exceptions.

Analogous to the experimental question of section 4.4, Q_1 and Q_3 map to the first column of table 3.1 (page 27). Answering them therefore demands analogous comparisons of the success of, for example, Natural blame and Natural exception trails for all debugging scenarios.

In detail, figure 5.4 summarizes this experimental process for one mode of the rational programmer and connects it with the mutations from section 5.6. The process is repeated for the same roots with the Natural-exceptions mode. After completing, the test bed reports the success/failure results of the trails to determine the proportion of scenarios where Natural-blame is more useful than Natural-exceptions. Question Q_1 has a positive answer if a root exists where the above is true because it is evidence that there is at least one interesting scenario that the rational programmer manages to debug because of blame information. The process is analogous for Q_2 and Q_3, using the respective modes.

For Q_*, the process is a bit more involved. Answering this question calls for a comparison of the percentage of scenarios where one mode is more useful than the other and the

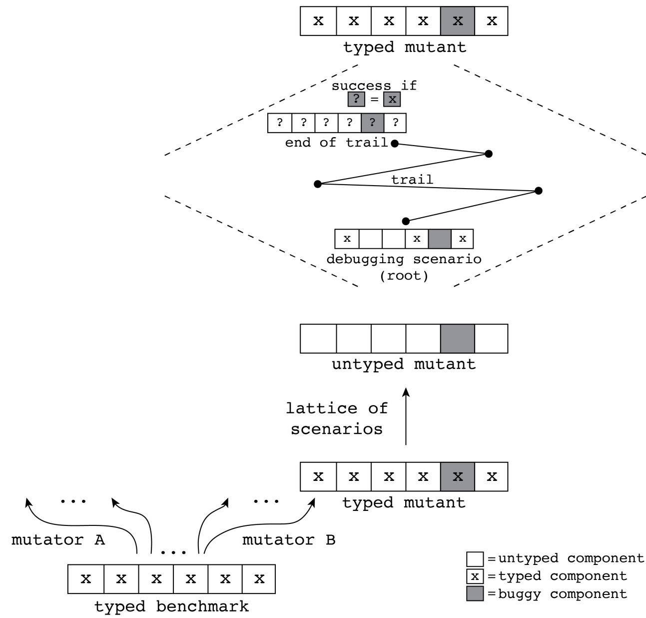


Figure 5.4: The experimental process for one mode of the rational programmer.

inverse. For instance, deciding whether blame for Natural is more useful than Transient-first requires comparing the percentage of scenarios where Natural-blame is more successful than Transient-first with the percentage where Transient-first is more successful than Natural-blame. Repeating the whole process for every pair of modes produces a complete picture of the comparative usefulness of blame.

5.6 Obtaining Debugging Scenarios for Gradual Types

Putting the rational programmer to work means generating many mutants and turning those into debugging scenarios. The process must start with a suitable collection of representative programs (sec. 5.6.1). Since existing mutators do not generate useful mutants, the next step is to develop new mutators (sec. 5.6.2) and to validate their suitability on the benchmarks (sec. 5.6.3).

5.6.1 The Experimental Benchmarks

As in the preceding chapter, the benchmark programs for this rational-programmer experiment must

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

In fact, the same suite of programs used in the prior chapter—Greenman, Takikawa, et al. [2019]'s collection of Typed Racket programs called the GTP benchmarks—forms a suitable basis that satisfies these criteria. Particularly relevant in this new setting, the benchmark suite consists of fully typed, correct programs, written by a number of different authors who had maintained and evolved these programs over time. The programs range widely in size, complexity, purpose, origin, and in programming style. They rely on many Typed Racket features: occurrence typing [Tobin-Hochstadt and Felleisen 2010], types for mutable and immutable data structures [Prashanth and Tobin-Hochstadt 2010], types for first-class classes and objects [Takikawa, Strickland, et al. 2012], and types for Racket's numeric tower [St-Amour, Tobin-Hochstadt, et al. 2012]. Finally, all of the programs are deterministic, so any changes in the programs' behavior between runs can be solely attributed to the actions of the rational programmer.

Table 5.2: Summary of benchmarks

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

Table 5.2 describes the ten benchmark programs that meet all the criteria, and furthermore come with the largest dependency graphs of the twenty programs in the suite. This additional filter reflects that Typed Racket requires that entire modules be either typed or untyped, and so finding errors in benchmarks with small dependency graphs would be trivial for the rational programmer.

5.6.2 How to Mutate Code for Type-level Mistakes

For the evaluation of a blame strategy, mutators must produce type-level mistakes that the run-time checks of gradual typing systems or the safety checks of the underlying language can detect. Once detected, the rational programmer should be able to locate the mistake by gradually adding types to blamed modules. In other words, the suitability of the mutators hinges on their ability to generate interesting debugging scenarios (see sec. 5.6.3).

Table 5.3 describes 16 mutators that satisfy these constraints. As the last column indicates, some specialize or generalize chapter 4's mutators, which in turn are borrowed from the vast literature on mutation testing [Y. Jia and Harman 2011]. Only two are directly

Table 5.3: Summary of mutators

name	description	example	origin
`constant`	swaps a constant with another of different type	5.6 → 5.6+0.0i	+
`deletion`	deletes the final expression from a sequence	`(begin x y z)` → `(begin x y)`	+
`position`	swaps two sub-expressions	`(f a 42 "b" 0)` → `(f a 42 0 "b")`	++
`list`	replaces `append` with `cons`	`append` → `cons`	new
`top-level-id`	swaps identifiers defined in the same module	`(f x 42)` → `(g x 42)`	new
`imported-id`	swaps identifiers imported from the same module	`(f x 42)` → `(g x 42)`	new
`method-id`	swaps two method identifiers	`(send o f x 42)` → `(send o g x 42)`	new
`field-id`	swaps two field identifiers	`(get-field o f)` → `(get-field o g)`	new
`class:init`	swaps values of class initializers	`(new c [a 5] [b "hello"])` → `(new c [a "hello"] [b 5])`	new

`class:parent`	replaces the parent of classes with `object%`	`(class a% (super-new))`
		→ `(class object%`
		`(super-new))`
			new

`class:public`	makes a public method private and vice versa	`(class object%`
		`(define/public (m x)`
		`x))`
		→ `(class object%`
		`(define/private (m x)`
		`x))`
			++

`class:super`	removes `super-new` calls	`(class a%`
		`(super-new))`
		→ `(class a% (void))`
			new

`arithmetic`	swaps arithmetic operators	`+` → `-`	++
`boolean`	swaps `and` and `or`	`and` → `or`	‡
`negate-cond`	negates conditional test expressions	`(if (= x 0) t e)` → `(if (not (= x 0)) t e)`	‡
`force-cond`	replaces conditional test expressions with `#t`	`(if (= x 0) t e)` → `(if #t t e)`	new

‡ inherited from, + specializes one of, ++ generalizes one of chapter 4's mutators

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

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

Figure 5.5: Example program using occurrence typing.

inherited; many mutators are brand new. For the latter, we relied on our decades-long experience of making type-level mistakes in Typed Racket, some of which take non-trivial effort to debug.

Most of the mutators are self-explanatory. The first six apply to all gradually typed languages; the next six to those that include classes and objects. The last four target distinguishing features of Typed Racket's type system, specifically its sophisticated type system. For example, one mutation produced by `arithmetic` replaces a `+` with a `-` in an attempt to change the type of the arithmetic expression; `+`'s result is a `Positive-Integer` when all arguments are positive integers, while `-` yields `Integer` [St-Amour, Tobin-Hochstadt, et al. 2012]. The other three also aim to confound the occurrence type system. Figure 5.5 illustrates how this confusion works. The function deals with an input that is either a `Real` or `#false`; the conditional deals with the first type in the `then` branch and the second type in the `else` branch. If a mutator wraps `(not ·)` around the test of the conditional, the resulting mutant is ill-typed and, when run, this function eventually causes a run-time type check to signal a type-value mismatch.

5.6.3 Are These Mutations Interesting?

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

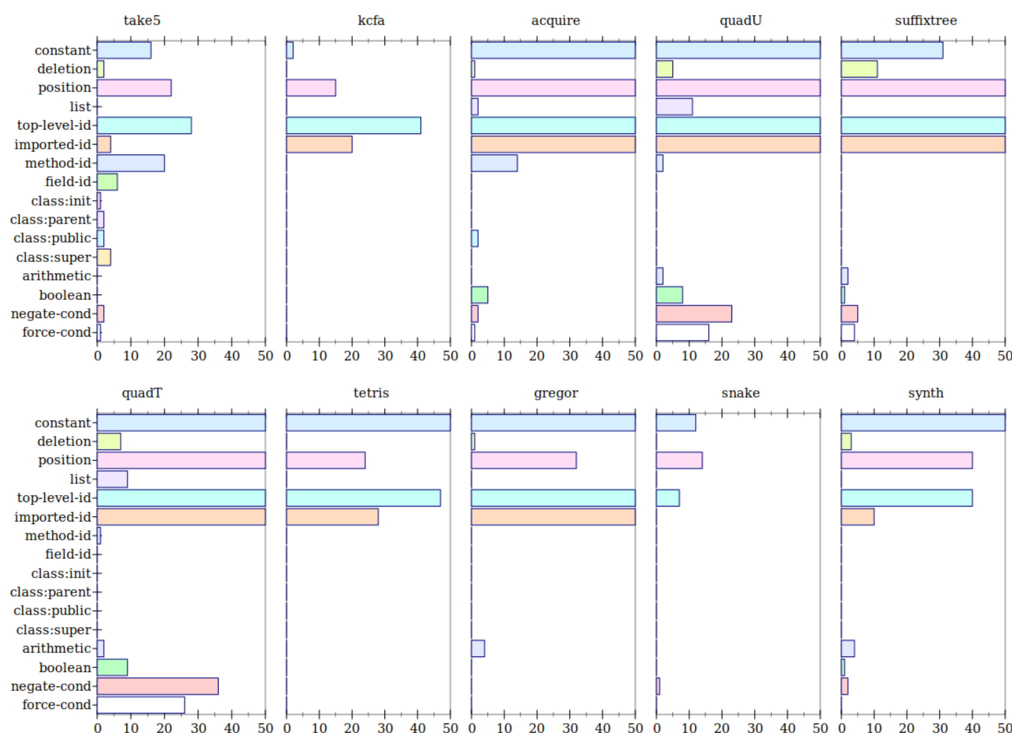
Here is the rationale for these three conditions:

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

2. The goal of a comparative evaluation is to give the rational programmer a chance to debug the same scenario using different pieces of information. In the case of gradual typing semantics, a meta-theorem due to Greenman and Felleisen [2018] says that if a program raises an exception under Erasure, it also errors under all other semantics. Hence, a comparison of blame information insists that an interesting mutant *raises a run-time exception under Erasure*.

   Note While this choice favors Erasure over Transient and Natural and, for the same reason, Transient over Natural, some form of bias towards one or the other semantics is unavoidable. As it turns out, the effect of this bias is small; section 5.8 discusses the bias in detail and quantifies it.

3. If the evaluation of a mutated module immediately raises an exception because of the changes, there is no work for the rational programmer. Indeed, if the stack trace contains source pointers to two modules, the scenario is still uninteresting. Every ordinary benchmark program comes with a `main` module that acts as a driver, whose



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

Figure 5.6: Breakdown of interesting mutants by mutator, per benchmark.

source is guaranteed to be included in the stack trace. Hence, the definition of interesting mutation insists on the presence of three different modules in the stack trace. This guarantees that the debugging scenario demands a sufficiently sophisticated effort, due to the interaction between the buggy module with its context. In these cases, the rational programmer must contend with at least two modules involved in a faulty interaction.

The definition of interesting mutants creates a powerful filter. All together, the listed mutators produce 16,800 interesting mutants across all benchmarks; see figure 5.6 for an

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

The goal of filtering for interesting mutants guided countless iterations of adding, removing, and refining mutators in table 5.3. For an illustrative example, consider a candidate mutator that casts the tests of conditionals to the `Any` type. Like the example explained at the end of the preceding subsection, this mutant would suppress occurrence typing. But, it would not be interesting because an execution would not raise a run-time error, for the static type of a conditional expression does not affect its runtime behavior. Hence this candidate mutator is not included in the final set.

5.6.4 Sampling the Space of Debugging Scenarios

As is, the chosen mutators generate approximately one million debugging scenarios for the chosen benchmarks. This number of scenarios is far too large to even identify the interesting ones among them. Hence, this experiment follows the same stratified random sampling strategy from the prior chapter to render the experiment computationally feasible. Specifically, the experiment samples 80 interesting mutants per benchmark, evenly-distributed across all of the mutators that contribute mutants for the benchmark. Some benchmarks have less than 80 mutants with interesting scenarios, in which case the only choice is to include them all. The result is a total of 752 interesting mutants across all benchmarks. Finally, the third

level of sampling randomly draws 96 debugging scenarios from each configuration lattice with replacement. The final sample thus consists of 72,192 interesting scenarios.

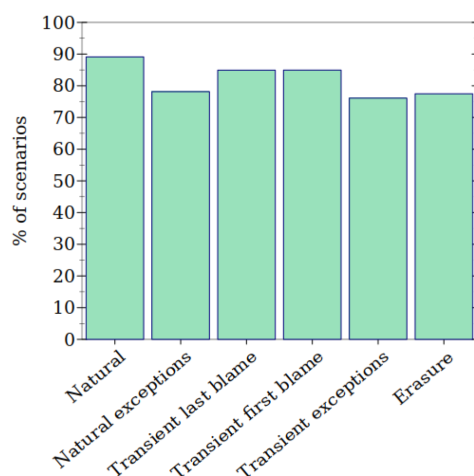
5.7 Results

We ran the experiment giving each debugging scenario a 4 minute timeout and a 6GB memory limit. Running the experiment on all debugging scenarios took over 30,000 compute hours or roughly three-and-a-half compute years.

Figure 5.7 summarizes the overall success rates of every mode. The success rates illustrate a few points that underlie the rest of the analysis. The first notable piece of information from this figure is that every mode has failed debugging scenarios, not just Erasure. This should not come as a surprise to the astute reader. Running a rational programmer mode on a scenario may result in an exception that carries no useful information about which module to equip with types next. For instance the stack trace of the exception may not contain frames from any untyped module of the program. This can happen at any point along a blame trail, causing it to fail.

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

In Transient, similar to Natural, most failures are due to unhelpful exception information (1851 for both Transient first and last blame). However, Transient also has a substantial number of failures because scenarios hit the time and/or memory limits of the experiment (˜770 scenarios). Additionally, there are nearly 1,000 cases where Transient reports an



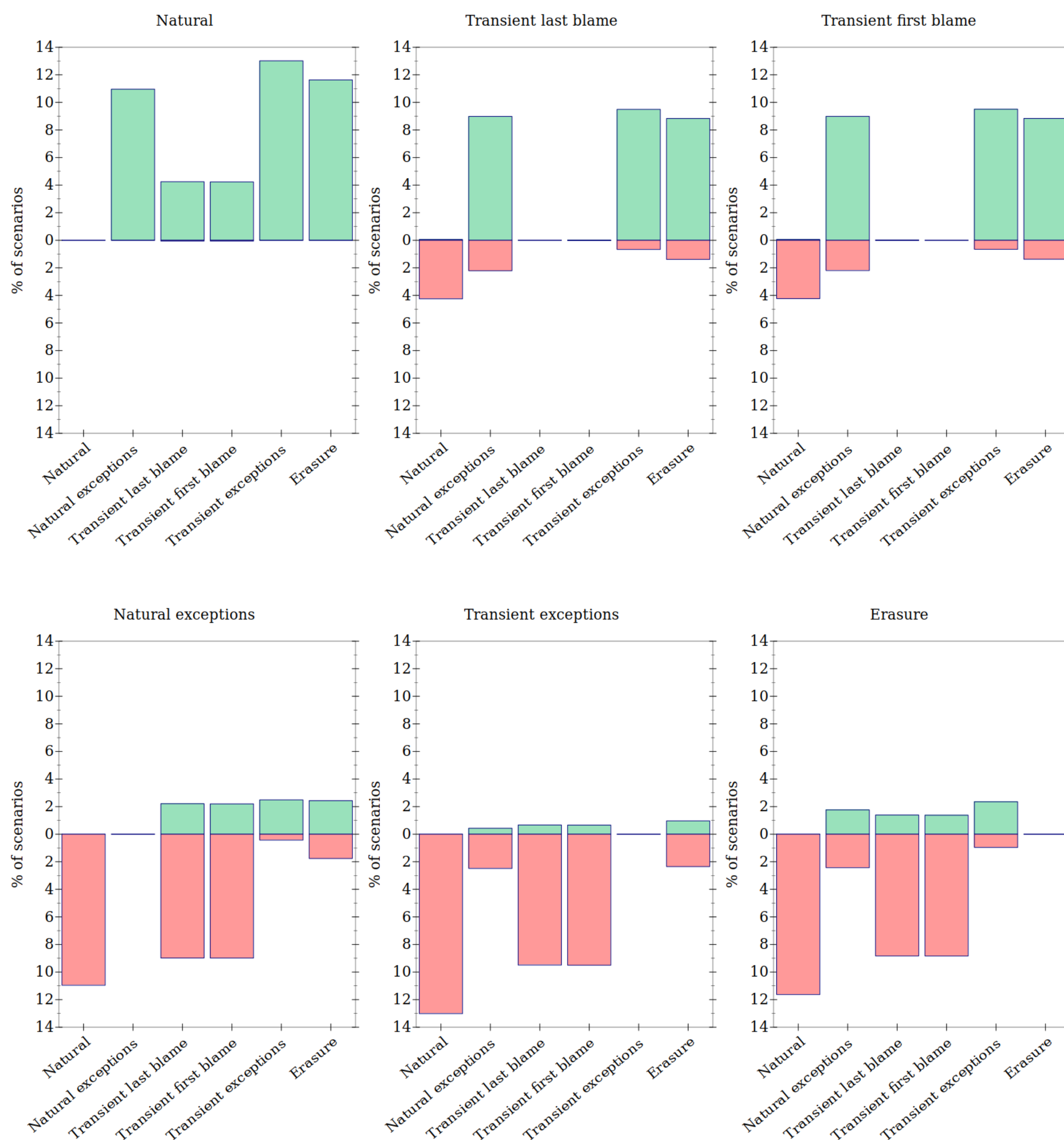
The upper bound margin of error is 0.02%.

Figure 5.7: Percentage rates of success.

empty blame set, leaving the rational programmer without hints about how to proceed. Sections 5.8.4 and 5.8.5 address these causes of failure for Transient and how they affect the experiment.

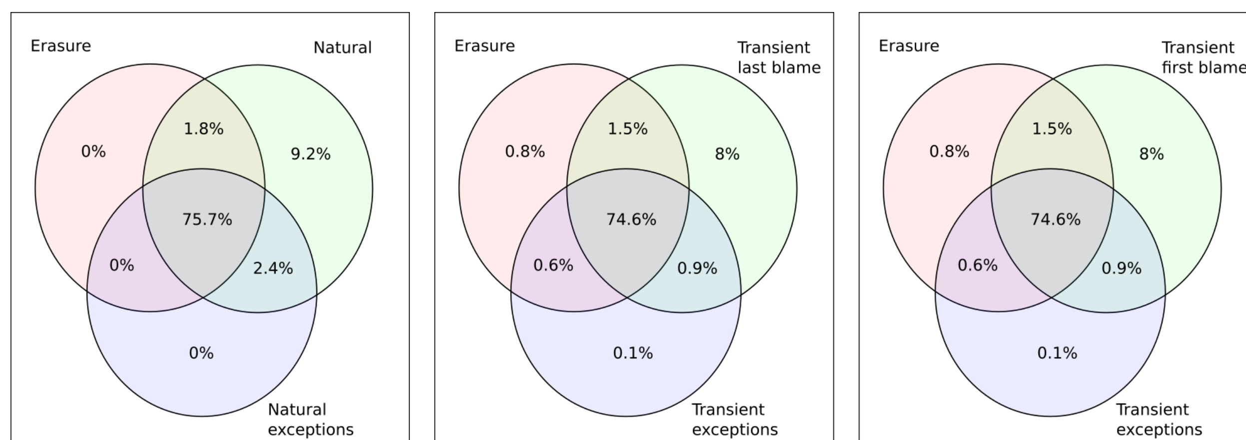
The second key observation from figure 5.7 is that the modes that use blame all outperform those that do not. In particular, Natural and both of Transient's blame modes succeed in 85 - 90% of the scenarios, while their corresponding exception modes succeed in less than 80% of them, and so too for Erasure. The only exception is that the random programmer always succeeds; the figure omits this mode because it just reflects the fact that every scenario has finitely many modules, so the random programmer eventually types the buggy module.

Figure 5.8 depicts a head-to-head comparison of every mode's performance against every other mode (except Random). The comparison answers the four questions from section 5.5.2. Each plot shows the proportion of scenarios where one mode performs better or worse than each other mode. In particular, each bar above zero represents the proportion where the plot's named mode succeeds and the mode on the x-axis fails; the corresponding bar below



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

Figure 5.8: Head to head usefulness comparisons.



Each diagram shows the overlap of the successful scenarios for three modes. For example, in the leftmost diagram, all three modes succeed on the same scenario 75.7% of the time, only Natural and Natural exceptions succeed on 11.6% of the scenarios, only Natural and Erasure succeed on 1.8%, and Natural alone succeeds on 9.2%. The upper bound margin of error is 0.02%.

Figure 5.9: Blame usefulness analysis

zero represents the proportion of the inverse case. For example, the plot titled "Natural" shows that Natural outperforms Natural exceptions in about 11% of the scenarios, and the inverse (Natural performs worse than Natural exceptions) never happens. Similarly, the plot titled "Transient last blame" shows that Transient last blame outperforms Natural exceptions in about 9% of the scenarios, but conversely it performs worse than Natural exceptions in about 2% of the scenarios.

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

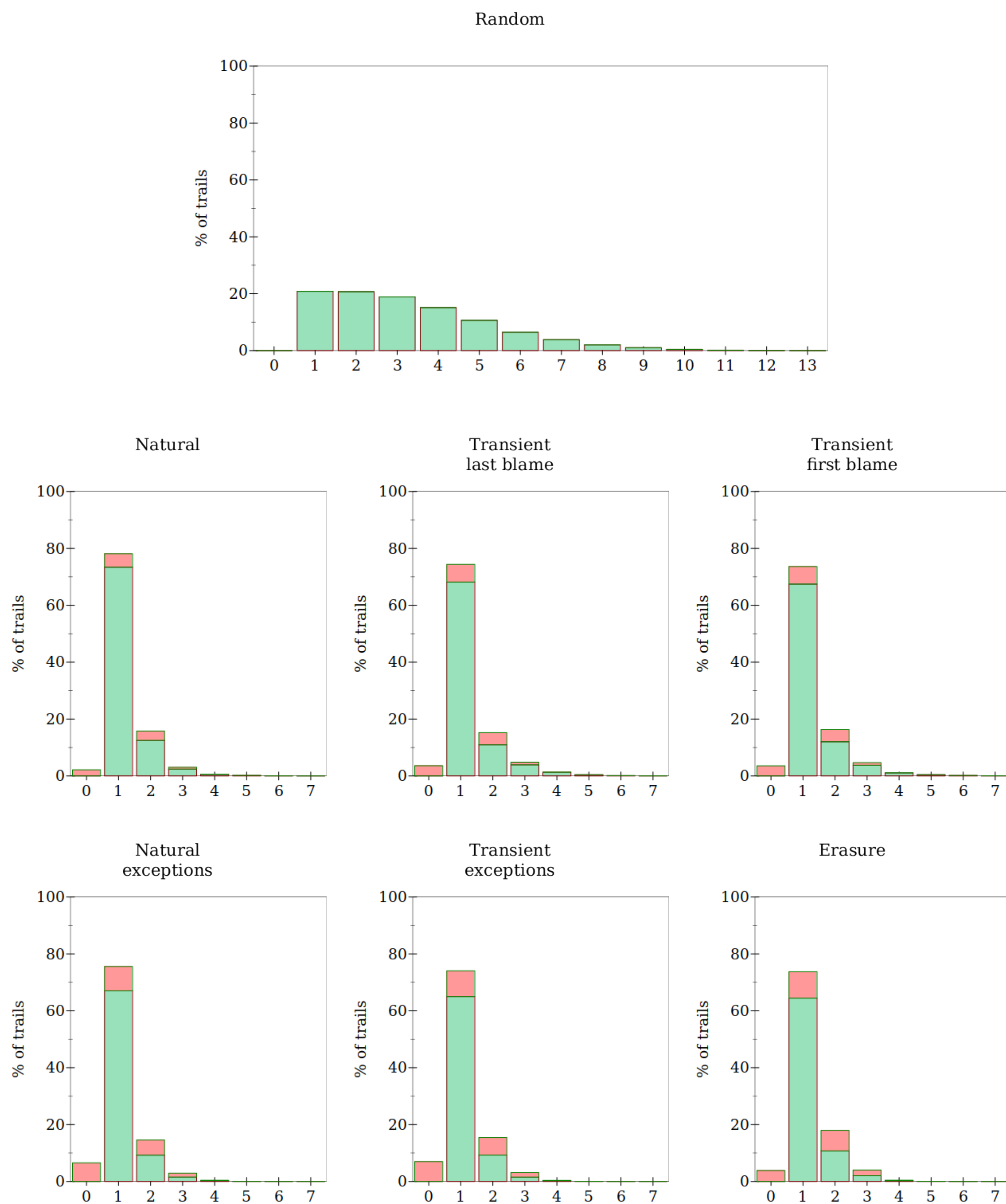
Figure 5.8 also provides answers for Q_*. Blame for all three semantics is significantly more useful than Erasure exceptions—by almost 12% for Natural and almost 9% for Transient. Natural blame is more useful than both versions of Transient blame by a small percentage

(about 4%). The Transient first and Transient last blame are practically indistinguishable. Finally, Natural exceptions are more useful than Transient exceptions, although only in a small percentage of scenarios (about 2.5%). A rare few scenarios (about 0.5%) show the opposite, despite the theoretically advantageous additional checks of Natural.

An alternative way to understand the answers for questions Q_1 to Q_3 is to analyze the success of each semantics in comparison to Erasure. Figure 5.9 depicts the results of this analysis. Specifically, the figure shows one Venn diagram per mode of the rational programmer that uses blame. Each diagram shows the overlap of successful scenarios for the blame mode, its corresponding exception mode, and Erasure. For example, the leftmost diagram (Natural) shows that all three modes succeed on 75.7% of the scenarios, only Natural and Natural exceptions succeed on 11.6% of the scenarios, only Natural and Erasure succeed on 1.8%, and Natural alone succeeds on 9.2%. This analysis highlights the success trade-offs each semantics offers against Erasure, with and without blame. For instance, the analysis for Natural clearly illustrates that, when choosing between Natural blame, Natural exceptions, and Erasure, Natural blame is the absolutely most successful: all of the successes of the other two modes are subsets of Natural's successful scenarios. On the other hand, Transient's blame modes fare similarly but the choice is not so clear-cut.

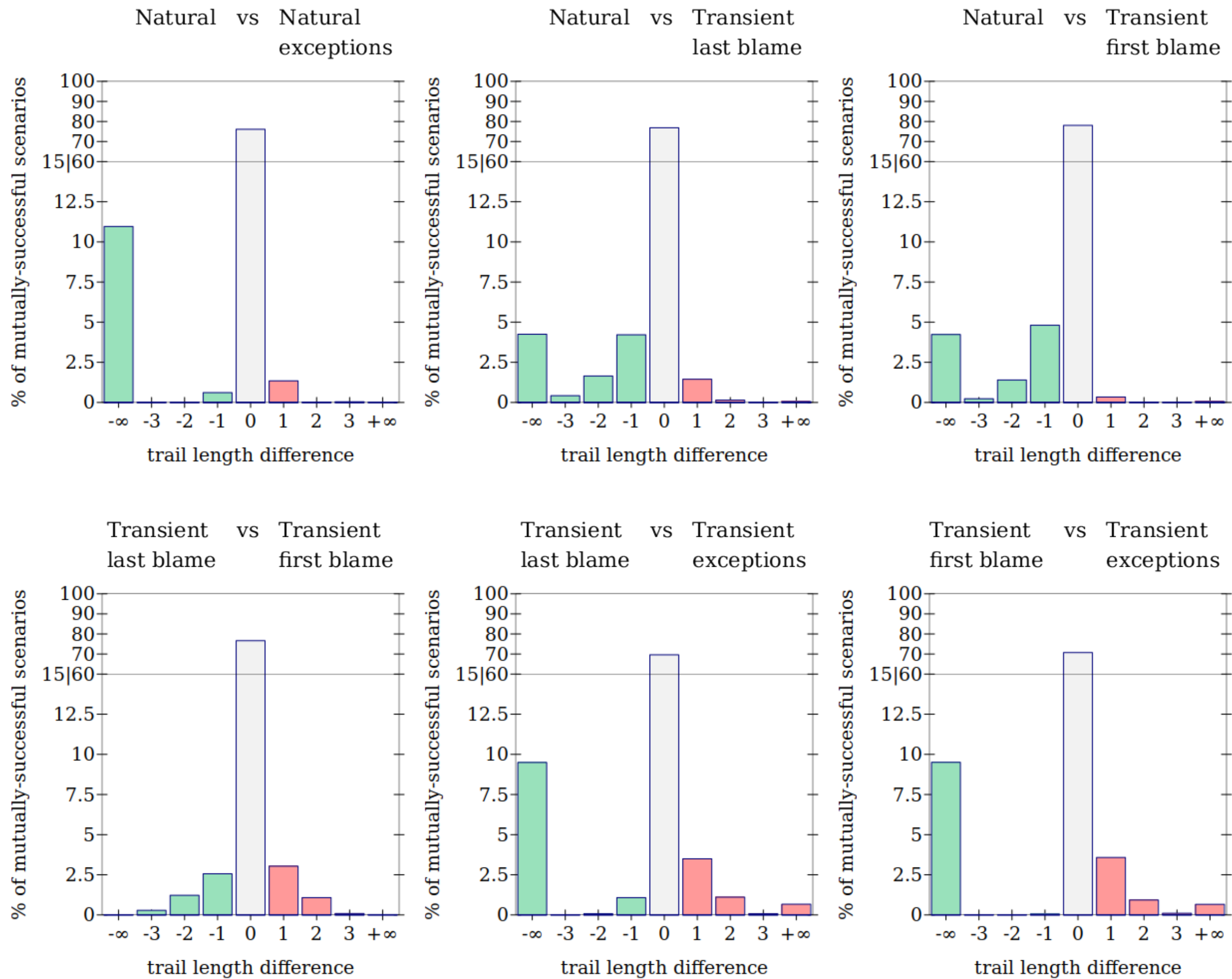
Turning to programmer effort, figure 5.10 shows the estimated distribution of blame trail lengths for the interesting debugging scenarios. There are two immediate takeaways from the figure. First, the effort for successfully debugging interesting scenarios (in green) for the random mode of the rational programmer is highly spread out, as expected. In contrast, in the other modes, successful effort coalesces at the left side of the plot, meaning that in most cases the programmer needs to type a single module to debug a scenario.

Figure 5.11 provides head-to-head comparisons of effort. The comparison between two



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

Figure 5.10: Programmer effort



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

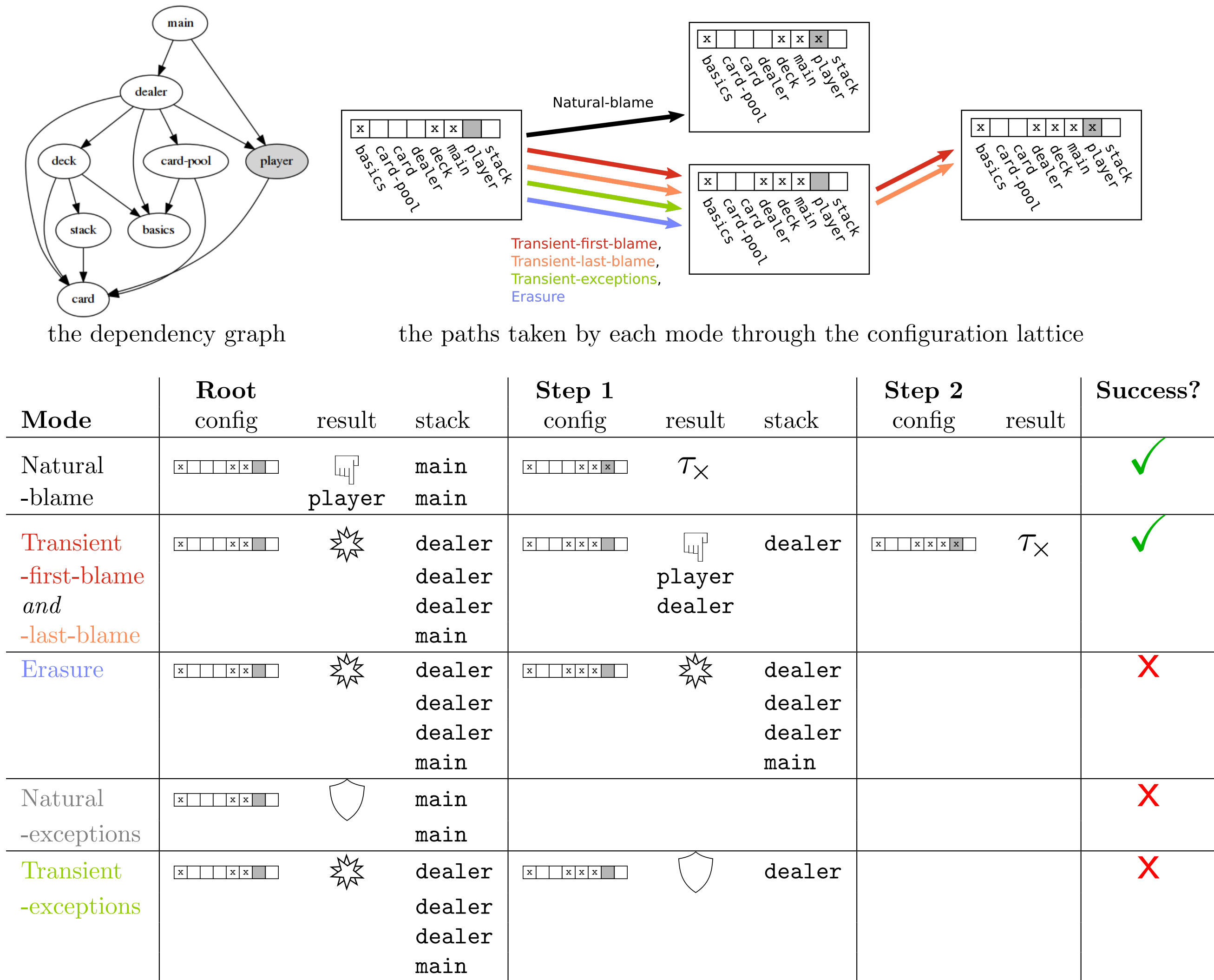
Figure 5.11: Effort comparisons

modes boils down to the difference in length between their trails for all scenarios where they both succeed. Hence, each plot in the figure shows the distribution of scenarios with length differences ranging from -3 (the first mode's trail is 3 steps shorter than the second's) to 3 (the first mode's trail is 3 steps longer than the second's). The figure offers several insights about how modes compare in terms of effort that complement the insights about how they compare in terms of success rates from figure 5.8. First, Natural blame rarely produces shorter trails than Natural exceptions, and occasionally produces slightly longer ones. Hence, the experiment provides evidence that blame helps the rational programmer debug more scenarios but does not shorten the debugging process compared to exceptions. Second, Natural relatively often (close to 8% of the scenarios) produces shorter trails than both Transient blame modes, and sometimes the trails are significantly shorter. Finally, Transient's blame modes share the characteristic with Natural that blame sometimes produces longer trails than their corresponding exception modes.

5.8 Lessons Learned

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

Figure 5.12 summarizes one particularly interesting debugging scenario from the `take5` benchmark. The module dependency graph of this benchmark is shown in the top left of the figure. Its mutated `player` module provides a method under a different name than the client module, `dealer`, expects. In Typed Racket's gradual type system, this mistake corresponds to a type-value mismatch—and all rational-programmer modes come to different conclusions (besides the two Transient blame modes).



Mode	Root config	Root result	Root stack	Step 1 config	Step 1 result	Step 1 stack	Step 2 config	Step 2 result	Success?
Natural -blame	☒☐☐☐☒☒■☐	blame: player	main main	☒☐☐☐☒☒☒■☐	does not type check				✓
Transient -first-blame *and* -last-blame	☒☐☐☐☒☒■☐	runtime failure	dealer dealer dealer main	☒☐☐☒☒☒■☐	blame: player dealer	dealer	☒☐☐☒☒☒☒■☐	does not type check	✓
Erasure	☒☐☐☐☒☒■☐	runtime failure	dealer dealer dealer main	☒☐☐☒☒☒■☐	runtime failure	dealer dealer dealer main			X
Natural -exceptions	☒☐☐☐☒☒■☐	blame ignored	main main						X
Transient -exceptions	☒☐☐☐☒☒■☐	runtime failure	dealer dealer dealer main	☒☐☐☒☒☒■☐	blame ignored	dealer			X

Legend

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

result

symbol	denotation
(pointing hand)	the configuration signals a dynamic type check failure, blaming the module(s) below
$\mathcal{T}_{\times}$	the configuration does not type check
(starburst)	the configuration fails a check by the runtime system
(shield)	the configuration signals a dynamic type check failure for which blame is ignored

Figure 5.12: An example scenario from take5, with every mode's resulting trail.

The rest of figure 5.12 illustrates the blame trails for every mode of the rational programmer (except Random) for the debugging scenario in two different ways:

- The top right shows the blame trails produced by every mode of the rational programmer as paths through the configuration lattice starting at the root (leftmost) configuration. Each configuration is represented by a sequence of boxes corresponding to modules in the program, with an x indicating that the module is typed. The mutated module has a gray box.

- The table in the middle expands the information in the diagram with the details of every step in each trail. Every row of the table represents the trail of one mode. The middle-three columns depict the steps of a blame trail:

  Root describes the result of running the root configuration in this row's mode.

  Step 1 displays the result of the rational programmer's reaction to the outcome of running the root.

  Step 2 shows the result of reacting to the outcome of running the step-1 configuration, if any.

  Finally, the **Success?** column summarizes whether exploring the trail succeeds.

To make this table concrete, compare rows 1 and 4. The first one shows that running the root configuration under the Natural-blame mode fails due a dynamic type check and blames the `player` module; typing that module and running again then results in a type error, and hence the trail is successful. By contrast, the Natural-exceptions mode (row 4) yields stack information for the root configuration that is unhelpful; it identifies only `main`, which is already typed. Hence, this trail immediately gets stuck.

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

5.8.1 Interpreting the Results

The results of the experiment suggests a number of high-level conclusions about blame strategies in the gradual typed world, at least for mistakes occuring in code rather than type annotations. First, all modes with run-time type checks have a fairly high success rate, regardless of whether these checks assign blame or throw plain exceptions. That said, the success rates of the modes without blame are on par with that of the Erasure mode. Second, error messages with blame assignments are more helpful than those without. The results also indicate, though, that blame is not critical in a majority of cases, and these two points together suggest investigating whether run-time type checking and blame tracking are worth the performance cost. Third, the Natural approach fares better than the Transient approach, but only by a small margin. Since Natural offers complete and sound path-based blame while Transient offers incomplete but sound heap-based blame [Greenman, Felleisen, and Dimoulas 2019], the results call for a study concerning the relative strengths of the two models of blame. Fourth, given that Transient's sound but shallow run-time type checks do not seem to hamper debugging, a language that supports *both* Natural and Transient might help reduce the number of wrappers and thus address the well-known performance issues of sound gradual typing [Greenman 2020, chapter 6]. Fifth, the fact that both modes of the Transient rational programmer are equally successful suggests that returning the whole blame sequence may not be beneficial. If so, Reticulated Python could limit the size of blame

sequences to attempt to mitigate its serious performance problems (see below).

5.8.2 Threats to Validity

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

The second set of threats questions four rather different aspects. To start with, just like the experiment of chapter 4, the rational programmer is an abstract model subject to questions about how it corresponds to the real world (sec. 4.7.2), and similar questions apply to the definition of "interesting scenarios" (sec. 5.8.3). Then, the generalization of these results to languages other than Typed Racket may not be direct, due to differences in design and runtime implementation that affect error messages (sec. 5.8.6). The remaining threats are about the accuracy and cost of Transient blame, respectively (secs. 5.8.4 and 5.8.5).

5.8.3 Threat: Is the Definition of Interesting Scenarios Reasonable?

Section 5.6.3 defines criteria for interesting mutations, one of which limits the scenarios under consideration to those with mistakes that raise a run-time error under Erasure. In other words, the experiment is Erasure-biased: it only considers the usefulness of blame when the safety checks of the underlying language alone are sufficient to detect the mistake. In reality, some mistakes require run-time type checks to be detected [Greenman, Felleisen, and Dimoulas 2019], and it is possible that blame has more to offer for these kinds of mistakes. If that is the case, then the results of the experiment on a population of scenarios including

such mistakes should be different.

In fact, we have reproduced the experiment with the opposite bias and the results demonstrate that the choice of bias does not affect the conclusions in a significant manner. Specifically, we filter to select only scenarios that raise a run-time error under Natural. Corresponding versions of the result figures are in appendix B, but at a high level the takeaway is that the Natural-blame mode improves over all other modes in slightly more scenarios (on the order of a few percent). Somewhat more interestingly, the exception modes, including Erasure, improve over both Transient blame modes in about 5% more scenarios in this variation. Thus Transient's blame appears even less useful in comparison with simple stacktraces, but only by a small measure.

5.8.4 Threat: Why Does Transient Lose Blame?

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

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

the construction of the blame map, instead of the identifier-based matching in Reticulated and Shallow Racket.

5.8.5 Threat: Is the Transient Blame Assignment Mechanism Realistic?

The results in section 5.7 also show that the cost of Transient blame is quite high. Under the Transient semantics, some of the debugging scenarios exceed the 4-minute timeout or the 6GB-memory limit. To put those limits into context, the fully typed and fully untyped benchmarks all normally complete in a few seconds with minimal memory usage. Furthermore, none of the mixed Natural configurations hit these limits, and with the blame map turned off, the Transient semantics also runs these programs in a short amount of time and well within the memory limit. In short, even though the Transient rational programmer appears to do well in the experiment, the implementation of the Transient blame strategy might be unrealistic.

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

1. The 2017 implementation of Reticulated fails to insert certain soundness checks[4] and blame-map updates[5] from the paper.

2. While Reticulated attempts to infer types for local variables, the impoverished nature

[4] Missing check (accessed April 2024): `https://github.com/mvitousek/reticulated/issues/36`
[5] Missing cast (accessed April 2024): `https://github.com/mvitousek/reticulated/issues/43`

of its type system does not allow the ascription of precise types and often resorts to type `Dynamic` [Greenman 2020, section 5.4.4]. Code with type `Dynamic` has fewer constraints to check at run-time—and much less information to track in the blame map.

3. Vitousek, Swords, et al. [2017] use small benchmarks. Four have since been retired from the official Python benchmark suite because they are too small, unrealistic, and unstable.[6] On the flip side, all the benchmarks in the GTP suite are larger than the official Python benchmarks. Reticulated Python runs the translation of the smallest GTP benchmark in approximately 40 seconds without blame but times out after 10 minutes with blame.

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

5.8.6 Threat: Different Languages, Different Types, Different Checks

The results described in this chapter depend upon the language at hand; other languages may have different type systems and different runtimes, each of which affect the errors and accompanying messages that type-value mismatches produce. For instance, consider the differences between Typed Racket (which we use in this chapter) and TypeScript (which is the most popular gradually typed language available today). 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 produces different stack traces than Racket.

[6] Release notes (accessed April 2024): `https://pyperformance.readthedocs.io/changelog.html`

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 (or any other language) is needed to clarify whether the conclusions of this work transfer from one linguistic setting to another. This dissertation 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.

5.9 Summary

The interviews of Tunnell Wilson et al. [2018] suggest that programmers prefer the run-time checking of Natural over other soundness methods. But, the opinion of a random set of programmers does not mean that blame assignment adds value. Similarly, researchers and language designers have implicitly answered this question one way or another without evidence for the blame-strategy dimension. The experiment presented in this chapter provides some justification for the programmers' leanings and helps language creators revisit their decisions. Of course, the design choice remains a trade-off along several dimensions, and the presented experiment sheds light on only one of them.

This chapter does *not* address a problem in the gradually typed world that was pointed out early on by practical researchers [St-Amour and Toronto 2013; Feldthaus and Møller 2014; Williams, Morris, Wadler, and Zalewski 2017] and that has recently received theoretical attention [Campora and Chen 2020; Greenman, Felleisen, and Dimoulas 2019]: mistakes in type annotations themselves. Developers use gradual typing to move an untyped code base into the typed realm, and to this end, they need typed APIs for the vast repositories

of already-existing libraries. Instead of converting the libraries themselves, language implementors merely create facade modules that import untyped functions and export them with type annotations, like `typed-pack-lib` in figure 5.1. With those facades, the compiler can type-check typed modules, but these retroactive additions of types to a library may result from a misunderstanding of the code. *In short, any retroactively ascribed type may thus be a mistake itself.*

The cited evidence suggests that this scenario is quite common and largely unaddressed. Hence, the next chapter applies the rational programmer framework to evaluate the pragmatics of error information when mistakes occur in type annotations.

CHAPTER 6
EXPERIMENT 3: GRADUAL TYPES AND BUGS IN TYPE ANNOTATIONS

This chapter follows up on the prior chapter's investigation to demonstrate how to close the open thread of what the pragmatics of gradual types are in the context of debugging bugs in type annotations rather than code. It is an adaptation of Lazarek, Greenman, et al. [2023] and joint work with Ben Greenman, Matthias Felleisen, and Christos Dimoulas.

The chapter begins with the essential background on the problem of buggy gradual types (sec. 6.1), instantiating the pieces of the framework (secs. 6.2-6.5), describing the results of the experiment (sec. 6.6), and discussing them (secs. 6.7-6.8).

6.1 Background: Gradual Types Can Be and Often Are Wrong

The preceding chapter describes and compares the process of debugging with error information provided by the three semantics for gradual typing in the face of type-level mistakes in one's code. However, the notion of migratory typing [Tobin-Hochstadt, Felleisen, et al. 2017]—which describes a process in which developers retroactively ascribe types to existing codebases in the process of maintenance, extension, or reuse—inherently allows a quite different type of problem to arise, where type annotations themselves may contain mistakes. In that setting, it is not necessarily clear that the gradual typed semantics' error information would be as useful in the original setting. The following subsections describe this different setting and offer an intuition for debugging there.

6.1.0.1 Wrong Gradual Types

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 client 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; Feldthaus and Møller 2014; Hoeflich et al. 2022; Kristensen and Møller 2017b; Williams, Morris, Wadler, and Zalewski 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

[1] `https://github.com/microsoft/TypeScript/network/dependents?dependent_type=PACKAGE` (accessed July 2023)

[2] `https://github.com/DefinitelyTyped/DefinitelyTyped`

the program runs, the second has investigated various approaches to run-time checking the boundary between typed and untyped pieces of code.

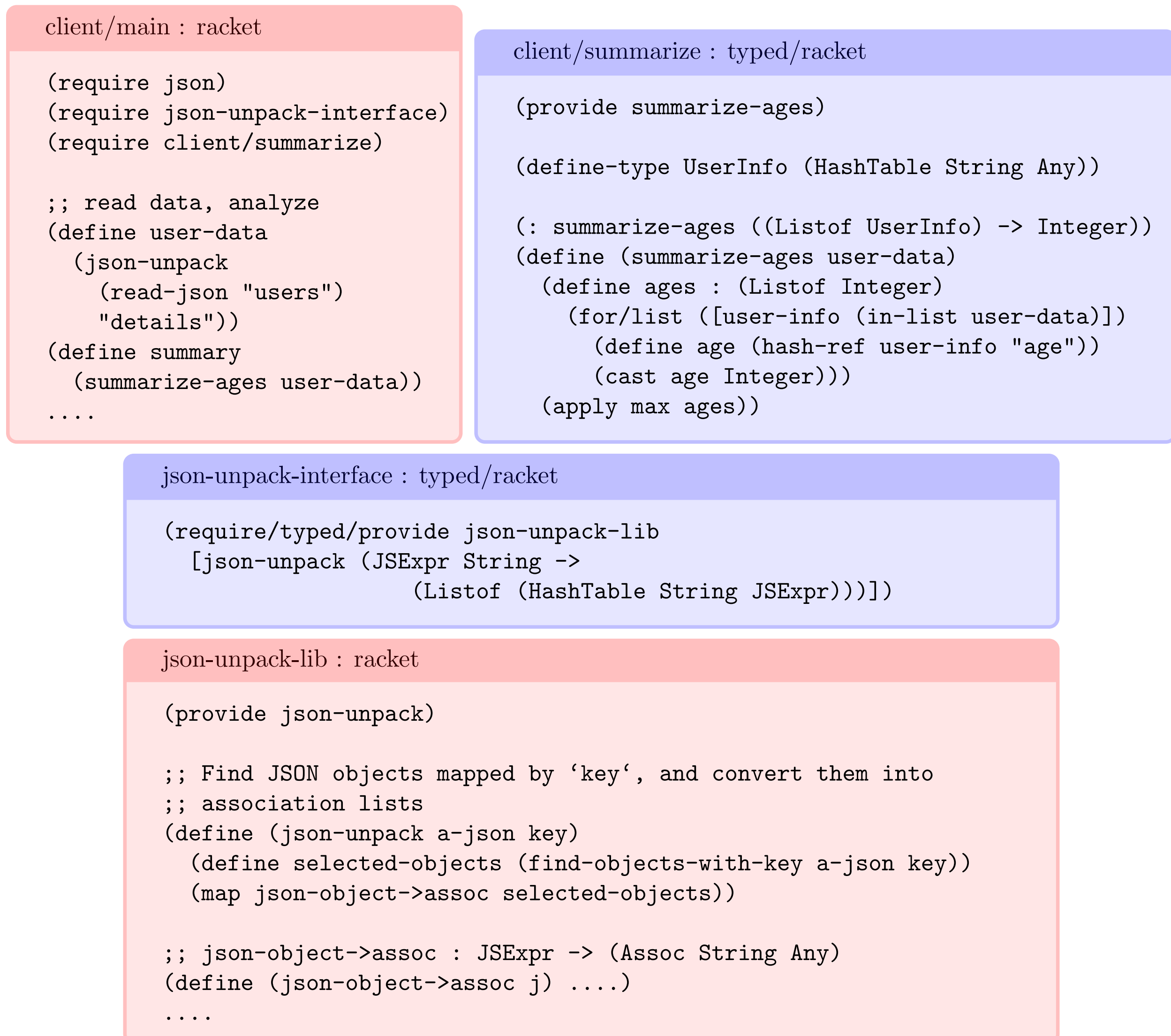
As described earlier, TypeScript is an industrial product that erases types and thus does not offer any special support to help programmers in the face of wrong type interfaces. The point is to keep type annotations from interfering with performance, or as the TypeScript website advertises, “TypeScript 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, 2010, 2008; Tobin-Hochstadt, Felleisen, et al. 2017] and Reticulated Python [Vitousek, Kent, et al. 2014; Vitousek, J. G. Siek, et al. 2019; Vitousek, Swords, et al. 2017]) compile types to run-time checks that aim to discover *type-value mismatches* between types imposed on untyped code and the latter's actual behavior. Moreover, when such run-time checking systems catch a type-value mismatch, they *blame* the boundary where a type interface and an untyped value (closure, object, class) are out of sync. Like in the previous setting, the question is whether the blame information from the Natural and Transient semantics offers useful hints, that is, hints that describe the cause of the mismatch and thus assist the developer with the debugging task.

Figure 6.1 sketches a program that illustrates the differences among the three semantics in the context of type interface mistakes concretely. 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.



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

Figure 6.1: One program with an incorrect type interface, three interpretations.

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 an association list. 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 type-value 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.

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, Felleisen, 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, Swords, 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 6.1, the chosen semantics matters, because three different semantics deliver three different hints. How can these hints be used to debug the program?

6.1.0.2 Debugging Type Interface Mistakes

The objective is to track down an incorrect type interface. To this end, a programmer can exploit feedback from the gradual type system and the error messages it produces to modify the program and obtain more information. The modifications aim to directly identify the incorrect type interface or to make a change that provides new information.

The modification strategy is like the one presented in section 5.1.1.1, 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 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 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 6.1. 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 a type-value 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 programmer may deduce that the type in `json-unpack-interface` must be wrong, for instance based upon the warning "`assuming the contract is correct`", or 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.

A simpler process can indirectly simulate this outcome as well. Specifically, the programmer 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 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 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 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 programmer gives the interface the benefit of the doubt, annotates `json-unpack-lib`, and reaches a type error.

A 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 programmer must therefore interpret the trace as blame assignment; a straightforward interpretation is to select 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 programmer using Erasure is stuck at this point.

In sum, essentially the same debugging process from section 5.1.1.1 appears to be applicable in the face of type interface mistakes as well.

6.2 The Hypothesis for Type Interface Mistakes

Section 6.1.0.2 describes how, based on an intuitive understanding of blame, to translate errors from a gradual type system into the location of type interface mistakes by adding type annotations. This process consists of following error-provided hints through the program, adding types to the components guided by the error information from the system. Eventually, the new annotations allow the type checker to discover the problem statically.

This chapter describes a rational programmer experiment that tests the hypothesis that this process is generally able to translate gradual typing error information into a static error identifying type interface mistakes. Specifically, the hypothesis is that

> for a program containing a type-interface mistake, adding type annotations guided by the error information available reliably leads to a static type-checker error.

To test this hypothesis, according to the outline of the method from chapter 2, we must

lay out a procedure that precisely captures the debugging process. The next section describes it in detail.

6.3 The Procedure for Type Interface Mistakes

The procedure in this setting is exactly the same as the preceding chapter. Furthermore, this chapter's experiment requires all the same modes as the preceding chapter. That said, there are subtle differences in the definition of the modes, so their definitions are provided in full.

6.3.1 The Type Migration Lattice

Like chapter 5, we follow Greenman [2023], Greenman, Takikawa, et al. [2019], and Takikawa, Feltey, et al. [2016] to describe the set of all possible type migrations with a lattice. The lattice describes the space in which the modes of the rational programmer search for bugs. Once again, however, there is a minor difference between the lattices traversed by the rational programmer in this experiment compared to those of the preceding chapter.

Just like in the preceding chapter, we define configurations of program P as a set of those components in P that are typed. However, in this setting, one of the typed components, $\mathcal{I}$, plays the role of the (wrong) type interface between the library and its clients as described in section 6.1.0.1. Since this component is always typed, we exclude it from the set of components that describe a configuration. Hence, 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.

6.3.2 How to Make Comparable Rational Programmers

Just like in the preceding chapter, a blame trail is simply an ascending chain of configurations of P starting at a debugging scenario. Unlike the preceding chapter, however, any configuration in this setting s_0 of $\mathcal{L}[\![P]\!]$ can be a debugging scenario.

While extending the trail, the rational programmer eventually encounters a scenario s_n that is the end of the trail. In this setting, 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 type-value mismatch. The trail ends in success. See section 6.1.0.2 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 6.5.3 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 or 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 type-value mismatch.

Besides this difference in the termination of blame trails, the modes of the rational programmer are defined in essentially the same way as in the preceding chapter.

6.3.2.1 The Natural Rational Programmer

This experiment uses the same definition of Natural blame trails as the preceding chapter.

Mode definition: Natural blame

A Natural blame trail *is a sequence of scenarios* $s_0, ...s_n$ *of a program* P *such that for all* $0 \leq i \leq n-1$, $s_i \subset s_{i+1}$ *and*

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

where

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

6.3.2.2 The Transient Rational Programmer

Just like the preceding chapter, we define two interpretations of Transient's multiple blame: one that selects the first untyped component in the blame sequence, and another that selects

the last. The definitions are the same as before, reproduced here for completeness.

Mode definition: Transient first blame

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

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

where

1. $\mathit{first}\,[\![\mathit{multiblame}\,[\![P, s]\!]]\!]$ *is the first untyped module that Transient adds to the blame sequence for s under the Transient semantics, and*
2. $\mathit{exception}_{\mathit{Transient}}\,[\![P, s]\!]$ *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 $\mathit{multiblame}\,[\![P, s_i]\!]$ that Transient adds to the blame sequence rather than the first.

6.3.2.3 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 a sequence of scenarios* $s_0, ...s_n$ *of a program* P *such that for all* $0 \leq i \leq n-1$, $s_i \subset s_{i+1}$ *and* $s_{i+1} \setminus s_i = \{exception_{Erasure}\,[\![P, s_i]\!]\}$.

6.4 The Experiment in Precise Terms

6.4.1 Success, Failure, and Usefulness

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.

Definition: *A Natural blame trail* $s_0, ...s_n$ *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.*

Just like the preceding chapter, we define a Natural exceptions mode to serve as a baseline

against which to judge the usefulness of blame.

Mode definition: Natural exceptions

A Natural exception trail *is a sequence of scenarios* $s_0, ...s_n$ *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, just like the preceding chapter.

Definition: *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.*

Likewise, we define a baseline mode for Transient blame trails and use it to define the corresponding usefulness of Transient blame.

Mode definition: Transient exceptions

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

The definition of success/failure and usefulness of the two interpretations of Transient blame are obvious adaptations of the definitions for Natural, so we omit them here.

6.4.2 Experimental Questions

We can now state the experimental questions 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])?

In terms of the space of experimental questions of table 3.1 (page 27), Q_1 through Q_3 capture the first column of questions, and Q_* the second column, with information from the third column (i.e. debugging effort) being a useful tie-breaker—all the same as the preceding chapter.

Since the questions and experimental setup at a high level are the same as the preceding chapter, the procedure for answering them is also the same. Rather than repeating it all here, it suffices to recall that the process to answer these 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.

6.5 Obtaining Debugging Scenarios with Type Interface Mistakes

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 type-value 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 6.5.1). 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 6.5.2). 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 6.5.3). Just like the preceding chapter, all these adapted mutants have the same migration lattices because they all share the same type-able components, and 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 in much the same way as the preceding chapter to obtain a sufficiently large and diverse but computationally feasible corpus for the rational programmer experiment (section 6.5.4).

6.5.1 The Seed of the Scenario Corpus

Our starting point is the same set of programs as the last chapter: Greenman, Takikawa, et al. [2019]'s GTP collection of Typed Racket programs.

In their original state, however, 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 the key feature of interesting type declarations. In particular, they cannot include data structure definitions, i.e., the type for Racket's `struct`s. The reason is that `struct`s in Racket are by default generative, and the type for a given `struct` is generated by its definition. 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 re-export `struct`s from the library side because ascribing those types would duplicate the data type definitions, creating new and incompatible types. In sum, as Greenman, Takikawa, et al. [2019] 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.[3] The key is to change all `struct`s to so called pre-fabricated `struct`s. These are

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

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 may also contain definitions for the data types, which opens up their mutation for the creation of incorrect views for client components.

6.5.2 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 6.1, 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 6.1 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

Table 6.1: 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]))`

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.[4]

- The last two swap fields in a structure type or class type definition.

6.5.2.1 Are These Mutators Interesting?

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 programs whose type interfaces ascribe the wrong type to values such that

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

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, Morris, Wadler, and Zalewski [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 type-value 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 a type-value 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. A type-value 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, Felleisen, and Dimoulas 2019], it guarantees to signal an error if the type-value 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 6.6 and 6.7.1 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 6.2 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.

6.5.3 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

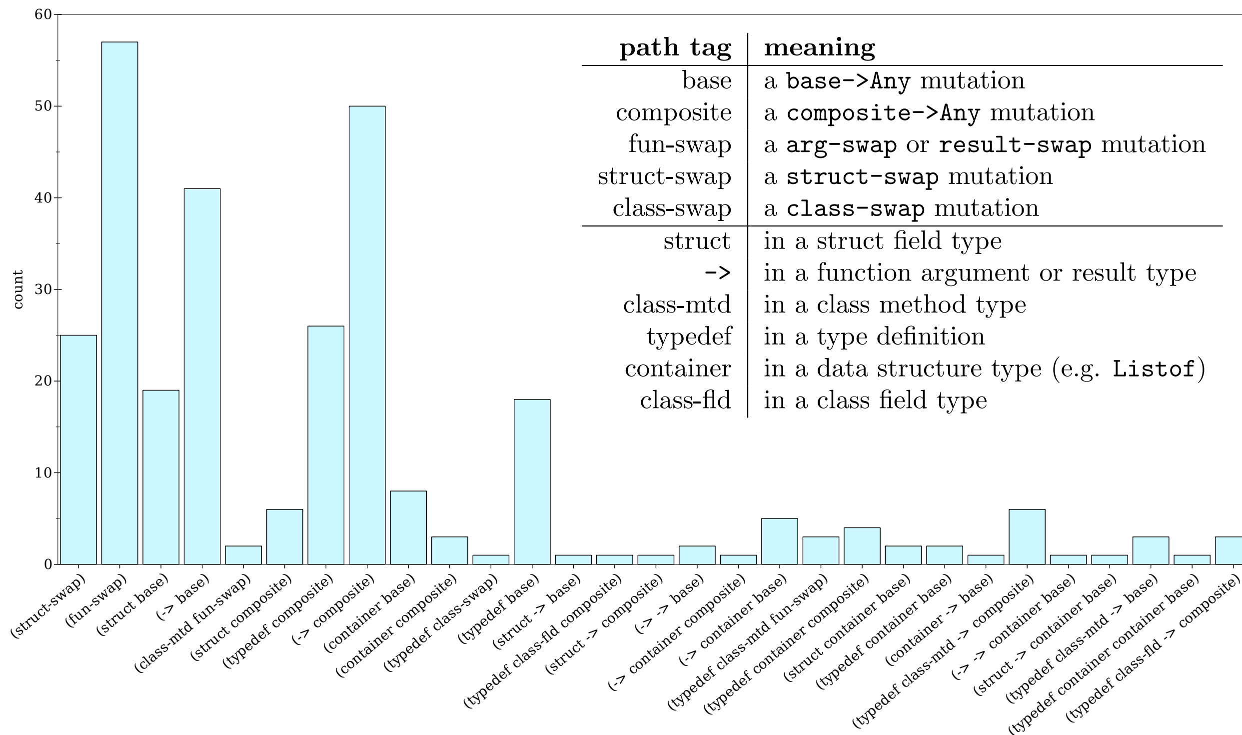


Figure 6.2: Type mistakes captured by final mutant population.

of the original interface, mutating the interface creates a disconnect between the client components and the types described by the mutated interface. Figure 6.3 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 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.

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

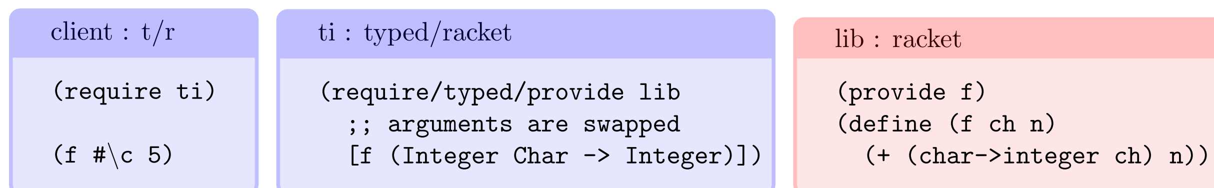


Figure 6.3: A simple program illustrating the need for adaptors.

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 untyped flow-adaptor module between the original interface and the mutated one.

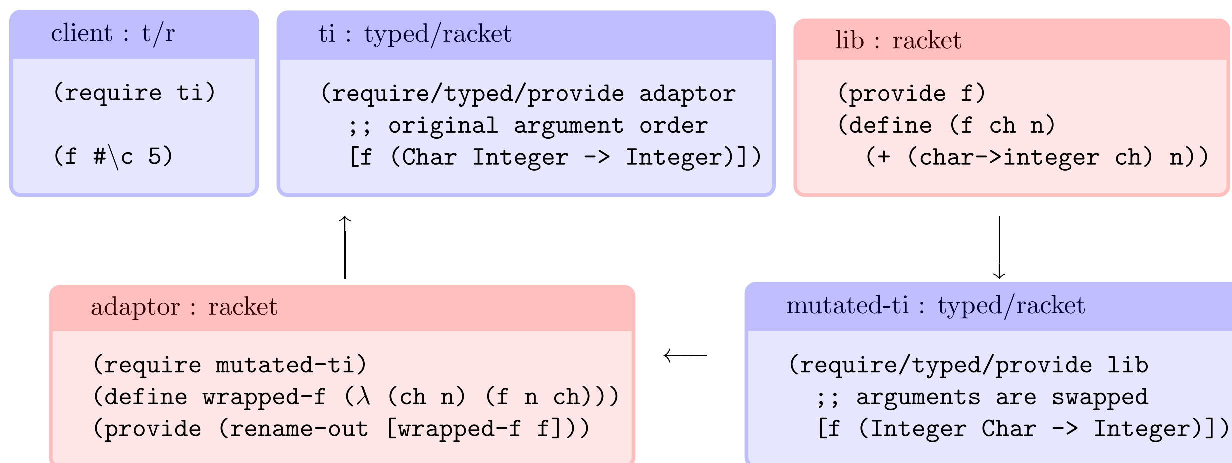
   The flow adaptor adjusts the flow of values at run time from the original type signatures to the mutated ones, respectively. It simulates a client that uses an exported value according to the mutated type.

   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.

In short, the faulty type interface in a debugging scenario 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 6.3. 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 6.4 represents the result of these 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.

Figure 6.4: Adapting the program of figure 6.3.

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, Tobin-Hochstadt, 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, we can return to the remark (2) in section 6.3.1 (page 130). 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 6.3.1, 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.

6.5.4 Sampling Debugging Scenarios

The 294 usable mutants across the selected GTP programs yield over two million different mutant × configuration pairs, which are the debugging scenarios that the rational programmer can explore. Hence, to perform the experiment within a feasible time-frame, we must once again sample this scenario space to obtain a reduced yet representative population.

We follow the same stratified random sampling approach as in chapter 5, using the same stratification criteria, with only minor differences to account for the difference in setting. We again filter out trivial scenarios, where the type checker directly identifies the type-value 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. As before, filtering out the trivial scenarios results in a computable sub-lattice of the migration lattice, in which no configuration contains a typed variant of the library component with faulty interface types. Thus we sample 100 interesting scenarios from that sub-lattice per mutant, just like the preceding chapter, this time arriving at a final population of 29,400 debugging scenarios for the rational programmer experiment.

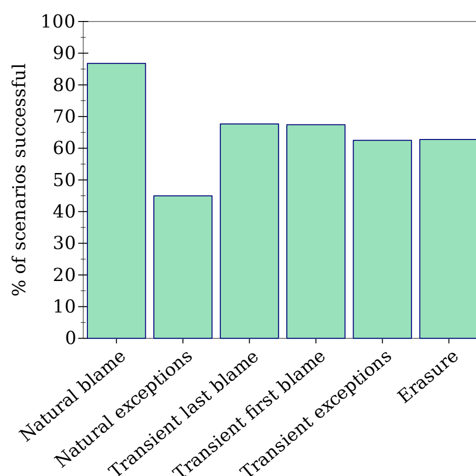
6.6 Results

We ran 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 6.5 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 successful for only 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



The upper bound margin of error is 0.08%.

Figure 6.5: Percentage rates of success.

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.

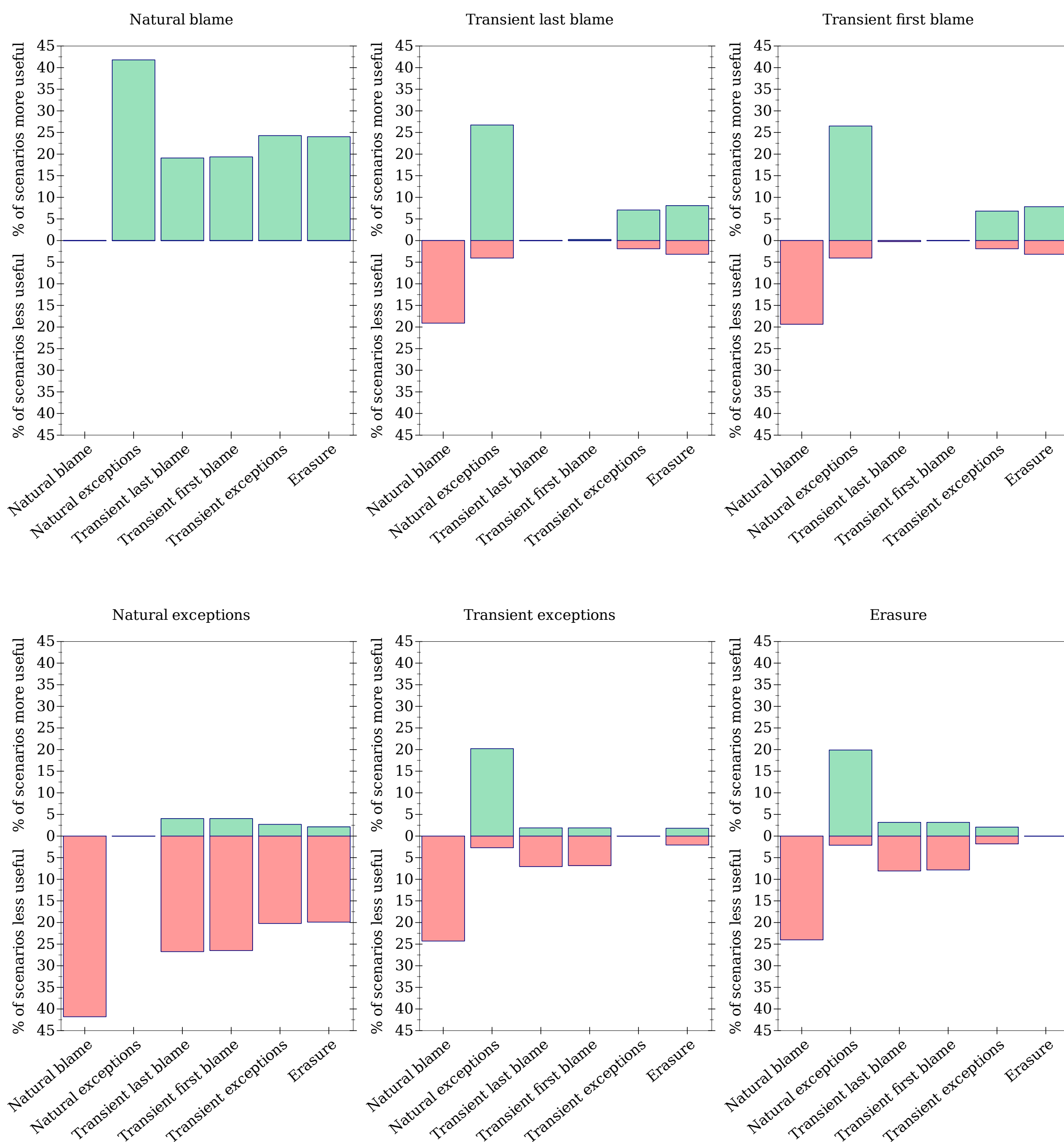
Similarly the stacktrace information does not help the Natural exceptions mode in about 15,000 scenarios. Of those, 3,400 scenarios are the same as those that stymie the Natural blame mode. In around 11,500 additional scenarios the Natural run-time type checks do signal a type-value mismatch, but the 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. Chapter 5 describes the exact same problem in greater detail.

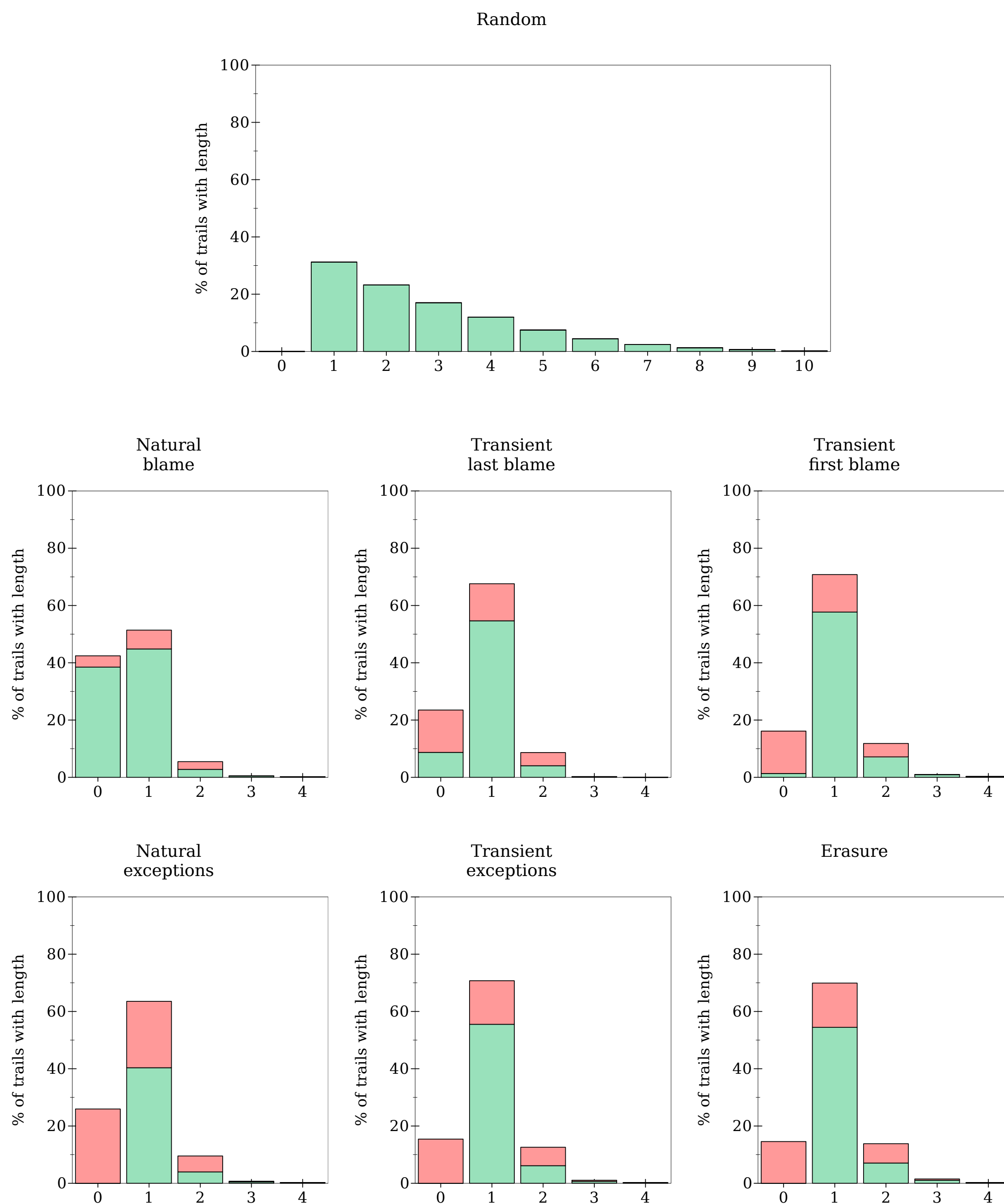
Finally, the Erasure mode fails in two ways: either the stacktrace information available from the exceptions of the underlying language are unhelpful, or the program terminates without any error. Unhelpful stacktrace information account for 8,700 of the Erasure mode's failures, and the program terminates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

Figure 6.6 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



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%.

Figure 6.6: Head to head usefulness comparisons.



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%.

Figure 6.7: Trail length distributions per mode.

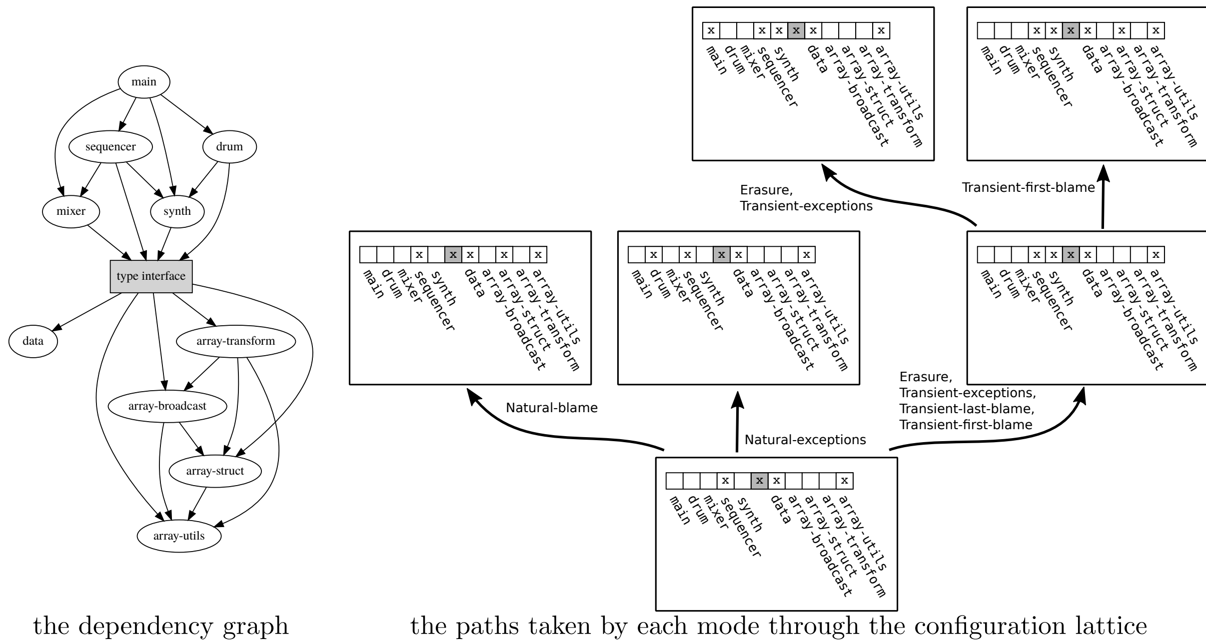
where the Natural exceptions mode is more successful than each of the other modes except for the Natural blame mode.

These results offer answers to the experimental question from section 6.4.2. 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, 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 6.7 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.

6.7 Lessons Learned

An intuitive understanding of the rational programmer's workings is instrumental to interpreting the aggregate results of the previous section. Figure 6.8 provides a detailed account of one scenario from the GTP program `synth`, which offers a useful illustration of how each



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	[ ][ ][ ][x][ ][x][x][ ][ ][ ][x]	☟ array-struct	drum	[ ][ ][ ][x][ ][x][x][ ][x][ ][x]	$\mathcal{T}_\times$				✓
Transient -first-blame	[ ][ ][ ][x][ ][x][x][ ][ ][ ][x]	✵	synth main	[ ][ ][ ][x][x][x][x][ ][ ][ ][x]	☟ array-struct	synth main	[ ][ ][ ][x][x][x][x][ ][x][ ][x]	$\mathcal{T}_\times$	✓
Transient -last-blame	[ ][ ][ ][x][ ][x][x][ ][ ][ ][x]	✵	synth main	[ ][ ][ ][x][x][x][x][ ][ ][ ][x]	☟ type-interface	synth main			✓
Erasure	[ ][ ][ ][x][ ][x][x][ ][ ][ ][x]	✵	synth main	[ ][ ][ ][x][x][x][x][ ][ ][ ][x]	✵	synth main	[x][ ][ ][x][x][x][x][ ][ ][ ][x]	✵	✗
Natural -exceptions	[ ][ ][ ][x][ ][x][x][ ][ ][ ][x]	⛉	drum	[ ][x][ ][x][ ][x][x][ ][ ][ ][x]	⛉	drum			✗
Transient -exceptions	[ ][ ][ ][x][ ][x][x][ ][ ][ ][x]	✵	synth main	[ ][ ][ ][x][x][x][x][ ][ ][ ][x]	✵	synth main	[x][ ][ ][x][x][x][x][ ][ ][ ][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
$\mathcal{T}_\times$	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

Figure 6.8: An example scenario from synth, with the trails that each mode explores.

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 6.5.3, 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,

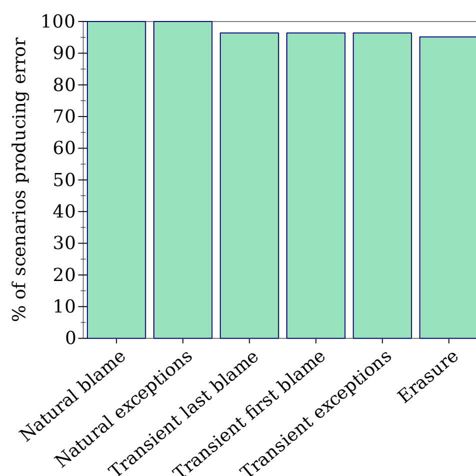
this situation corresponds to one where there is an untyped library module in between the buggy type interface and the typed library module that detects the mismatch, which would be blamed, and which, once annotated, would make the mismatch apparent to the type checker. Thus the two modes take different paths to success in this scenario.

6.7.1 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 chapter 5'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 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 6.6 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 upper bound margin of error is 0.01%.

Figure 6.9: Estimated percentage rates of bug detection (i.e. halting with an error).

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 type-value mismatches go entirely unnoticed when using Erasure, and the data in figure 6.9 suggests that is probably the case.

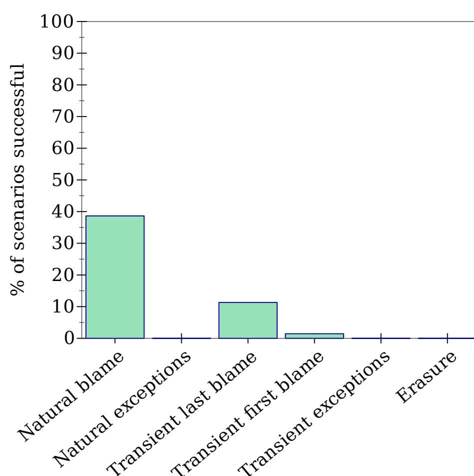
Figure 6.9 furthermore directly quantifies the effect of our choice to filter mutants using the Natural semantics, illustrating that the resulting bias affects at most four percent of the scenarios we test. In detail, the main way that this choice of filtering (sec. 6.5.2) biases in favor of Natural is that with the Natural semantics we may include scenarios that Natural is able to detect, but the other semantics cannot. If there were many such scenarios in our results, it would raise the question of whether the experiment is even an apples-to-apples comparison of error message quality—as opposed to bug-detecting ability. However, the

proportion of these scenarios is significantly smaller than the size of the effects informing our conclusions. To further underscore the insignificance of this bias, appendix C presents the results of a reproduction of the experiment with a bias in favor of Erasure instead of Natural; those results are at a high level identical to the preceding section, differing only by small percentages.

6.7.2 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. 6.5), 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, and just as in the other two experiments, the rational programmer itself does not necessarily reflect the way real programmers use gradual types or debug mistakes in type interfaces (sec. 4.7.2). Also like the prior experiment, the results of this experiment do not necessarily translate directly to other gradually typed languages, but the method of evaluation applies to any such other contexts (sec. 5.8.6). At a more technical level, the experiment design assumes that the rational programmer can



The upper bound margin of error is 0.04%.

Figure 6.10: Estimated percentages of trails that succeed without typing library modules.

inspect and annotate library components, which real programmers may not be able to do (sec. 6.7.3).

6.7.3 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 a type-value 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 6.10 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.

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 6.6, even if stack-trace information points to client components, adding types to client components can never turn the type-value 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.

6.8 Summary

When it comes to *detecting* type interface mistakes, all semantics are essentially equally good, at least for the programs under consideration here. 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 the preceding chapter 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.

CHAPTER 7
RELATED WORK

This chapter assesses related work along three different dimensions. First is the object-level, encompassing related work on the objects of study in this dissertation, namely contracts, gradual typing, and their error information. Second is the meta-level, referring to work related to the methods of this dissertation; that is, prior work on evaluating error/debugging information as well as debugging strategies. Finally, the scenario construction for the experiments of prior chapters employs or draws inspiration from a handful of Software Engineering techniques that have storied research literatures. The rest of the chapter is accordingly divided into three sections concerning each of these dimensions.

7.1 Contracts and Gradual Typing

7.1.1 Contracts

Eiffel is the first programming language to popularize the idea and practice of contracts with the introduction of the "Design by Contract" methodology [Meyer 1991, 1992, 1988], which systematizes ideas about using assertions to check function inputs and outputs (see Parnas [1972]'s work). Findler and Felleisen [2002] use delayed checks to lift contracts to the world of higher order functions, and introduce blame to pinpoint the component at fault when a contract fails. This work has since led to a significant body of research on the design of higher order contract systems; including support for various host language features [Degen et al. 2012; Greenberg et al. 2012; Hinze et al. 2006; L. Jia et al. 2016; Strickland, Dimoulas, et al.

2013; Strickland and Felleisen 2009a], extensions of the contract system's expressiveness and its efficient implementation [Disney et al. 2011; Feltey et al. 2018; Findler, Guo, et al. 2007; Greenberg 2015; Keil and Theimann 2015; Moy, Dimoulas, et al. 2024; Moy and Felleisen 2023; Scholliers et al. 2015; Strickland and Felleisen 2009b; Strickland, Tobin-Hochstadt, et al. 2012; Swords 2019; Swords et al. 2018; Williams, Morris, and Wadler 2018], and applications and extensions of contracts for checking properties beyond partial functional correctness or in different ways [Dimoulas, Moore, et al. 2014; Heidegger et al. 2012; Moore, Dimoulas, Findler, et al. 2016; Moore, Dimoulas, D. King, et al. 2014; Waye et al. 2017; Xu et al. 2009; C. Zhang et al. 2022]. Alongside and intermixed therein is work on various aspects of contract system semantics [Blume and McAllester 2006; Degen et al. 2008, 2010, 2009; Dimoulas and Felleisen 2011; Dimoulas, Findler, Flanagan, et al. 2011; Dimoulas, Tobin-Hochstadt, et al. 2012; Findler and Blume 2006; Findler, Felleisen, and Blume 2004].

Within the space of work on the semantics of contracts, Dimoulas, Findler, Flanagan, et al. [2011] define formal correctness criteria for blame called Complete Monitoring; this constitutes an effort to evaluate the specialized debugging information of higher-order contracts at a theoretical level. In particular, Complete Monitoring gives meaning to blame as a view of the flow of the witness of a contract violation. However, this semantic definition does not shed light on the practical relationship between blame and bugs in programs, which is the aim of this dissertation. That said, this work has been a primary source of inspiration for the hypotheses that we test.

7.1.2 Gradual Typing

Gradual typing has been a topic of research interest for nearly two decades, beginning with several related seminal ideas: combining the benefits of static and dynamic typing in a sin-

gle language [J. G. Siek and Taha 2006]; enabling a smooth, incremental transition from a prototyping-friendly dynamic programming style to a maintenance-friendly type discipline [Tobin-Hochstadt and Felleisen 2006]; safely interoperating between (different) dynamic and statically typed languages [Gray et al. 2005; Matthews and Findler 2007, 2009]; and extending the flexibility of static specification analysis with the ability to offload checks to runtime contracts [Knowles and Flanagan 2010]. These ideas have served as the foundation for a significant body of research on and around gradual typing, including extending the core idea to handle various language and type system features [Broman and J. G. Siek 2017; Garcia and Cimini 2015; Igarashi, Thiemann, et al. 2019; Malewski et al. 2021; Miyazaki et al. 2019; New et al. 2023; Rastogi, Chaudhuri, et al. 2012; J. G. Siek and Vachharajani 2008; Takikawa, Strickland, et al. 2012; Tobin-Hochstadt and Felleisen 2008; Toro and Tanter 2017; Turcotte et al. 2020; Vitousek, Kent, et al. 2014; Wolff et al. 2011; Wrigstad et al. 2010; Ye and Oliveira 2023], and analyzing the performance of or efficiently implementing runtime systems for gradual typing [Bauman, Bolz-Tereick, et al. 2017; Bauman, Bolz, et al. 2015; Campora, Chen, and Walkingshaw 2018; Campora, Khan, et al. 2024; Castagna et al. 2019; Chevalier-Boisvert et al. 2021; Greenman 2022, 2023; Greenman and Felleisen 2018; Greenman and Migeed 2018; Greenman, Takikawa, et al. 2019; Greenwood-Thessman et al. 2021; Kuhlenschmidt et al. 2019; Moy, Nguyen, et al. 2021; Muehlboeck and Tate 2021; Richards, Arteca, et al. 2017; J. Siek, Thiemann, et al. 2015; Takikawa, Feltey, et al. 2016; Vitousek, J. G. Siek, et al. 2019] — including notably Feltey et al. [2018], who introduce and evaluate a significant optimization that is actively used today in Typed Racket.

Another direction for understanding and improving the performance of gradual typing explores different semantics for typed-untyped interaction. Chapter 5.1 introduces the Natural, Transient, and Erasure semantics in-depth, but the literature describes a wide array of

other semantics as well. Pyret [Developers 2018] assigns fixed-size data types the Natural semantics and functions a Transient semantics. The Amnesic [Greenman, Felleisen, and Dimoulas 2019] semantics is similar to Transient but uses wrappers instead of in-lined checks, and Greenberg [2015]'s Forgetful and associated semantics perform several variations of this strategy. Nom [Muehlboeck and Tate 2017] and other *concrete* semantics [Rastogi, Swamy, et al. 2015; Richards, Arteca, et al. 2017; Richards, Nardelli, et al. 2015; Wrigstad et al. 2010] assume that every value comes with a type tag and use tag checks to supervise the interactions between typed and untyped code. The semantics derived with the Abstracting Gradual Typing technique [Garcia, Clark, et al. 2016] are variants of Natural. The Monotonic semantics [Kuhlenschmidt et al. 2019; Rastogi, Swamy, et al. 2015; J. Siek, Vitousek, et al. 2015; Swamy et al. 2014] differs from Natural in the treatment of mutable data; it associates every heap location with a type and rejects updates that lower the precision of types. Threesomes and coercion/cast semantics [Almahallawi 2020; Herman et al. 2010; J. G. Siek and Wadler 2010] describe versions of Natural's wrapping semantics that collapse multiple wrappers to achieve space efficiency. In response to the large variety in these semantics and their different trade-offs [Gierczak et al. 2024] along many dimensions, Greenman [2020, 2022] explores the properties and benefits of combining semantics that do *deep* checking (like Natural), *shallow* checking (like Transient), and none at all (like Erasure) in a single language, controllable by the programmer.

The various choices made by these different semantics affect the debugging information they produce. Prior work studies (some of) those differences only from a theoretical perspective [Greenman, Felleisen, and Dimoulas 2019]. Aside from Natural and Transient (sec. 5.1), only the Amnesic, Nom, Monotonic, and Threesomes semantics present interesting blame strategies. Amnesic offers a kind of precise blame in the style of Natural, achieved through

limited wrappers, but on the basis of Transient-style shallow tag checks instead of deep ones, and also has tunable knobs (e.g. how much history to track) that can affect the quality of blame. Nom has a nominal type system for an object-oriented language that offers a kind of blame that identifies one (failing) cast from type dynamic to some incompatible type — where compatibility means membership in the expected-type's inheritance hierarchy. Monotonic similarly offers a kind of blame that identifies two casts of a mutable reference to incompatible types. Also similarly, Threesomes identifies a (failing) cast in the source program between two incompatible types — where compatibility means having a well-defined join point in a precision lattice of types. The experiments in this dissertation exclude the first and last of these (Amnesic and Threesomes) because they are purely theoretical constructions, the second (Nom) because it imposes severe restrictions on programmers, and the third (Monotonic) because it would require an impractical re-engineering of the Racket runtime.

7.1.3 Type Mistakes in Gradual Typing

There are significant bodies of adjacent literature related to mistakes in type annotations. In particular, a number of papers investigate the prevalence of mistakes in gradual types, their theoretical underpinnings, and proposing approaches to detect and repair them [St-Amour and Toronto 2013; Campora and Chen 2020; Cristiani and Thiemann 2021; Feldthaus and Møller 2014; Greenman, Dimoulas, et al. 2023; Greenman, Felleisen, and Dimoulas 2019; Hoeflich et al. 2022; Kristensen and Møller 2017b; Williams, Morris, Wadler, and Zalewski 2017]. While none of this work offers a systematically curated collection of type mistakes and their fixes suitable for experiments like those of this dissertation, their observations have been a primary source of inspiration for the mutation operators we define in chapter 6.

7.2 Evaluations of Debugging Information and Strategies

7.2.1 Fault Localization

The well-established area of fault localization is related to this dissertation at a methodological level insofar as work in this space evaluates debugging information or (semi-)automated procedures for locating bugs. The origins of fault localization (FL) go back to the interactive debugging approach of Shapiro [1983], and modern automatic FL research build on the work of Agrawal [Agrawal 1991; Agrawal et al. 1995] and Jones et al. [2002], who use comparisons of successful and failing executions of a program to deduce a set of likely faulty program statements. There is also a connection from these ideas to types and type checkers in an extensive body of work on the accuracy of type checker error messages, the foundations of which are summarized by Heeren [2005], and a significant group of work about helping programmers debug type errors in *statically typed* settings [Becker et al. 2016; Chen and Erwig 2014; Pavlinovic et al. 2014; Seidel, Jhala, et al. 2016, 2018; D. Zhang and Myers 2014], including notably Wu and Chen [2017]'s finding that a significant portion of such errors arise from incorrect type annotations.

While this dissertation does present strategies for locating faults, it expressly does not aim to propose a technique for FL. The goal is rather to analyze blame from a pragmatics perspective that connects semantics with real buggy programs, and use this analysis to evaluate the design of contract systems and semantics for gradual typing with respect to the debugging information they offer. Whether the blame shifting procedures presented in this dissertation could inform practical debugging strategies for developers to follow while debugging, or be implemented in tools supporting debugging, are questions demanding further research.

Besides the procedures themselves, the main methodological connection with the aforementioned work might be through the methods by which they evaluate their proposed FL techniques. In this respect too, however, this dissertation does not propose an evaluation method for FL. Intent aside, there are also differences in approach between this work and standard FL evaluation methods. The standards set by the prior-mentioned seminal works consist primarily of metrics or scores describing how frequently the tools report the true bug location in various suites of student assignment solutions. Popular metrics in the literature include: notions of accuracy [Pavlinovic et al. 2014] (how syntactically close the suggested location is to the true bug's location); precision and recall [Loncaric et al. 2016; Seidel, Sibghat, et al. 2017; D. Zhang and Myers 2014; D. Zhang, Myers, et al. 2015] (what proportion of the suggested locations are bugs / what proportion of bugs the tool locates); proportion of bugs located in the top N suggestions [Chen and Erwig 2014] (for tools that suggest multiple, usually ranked, possible locations); and yet others [Wu, Campora III, et al. 2017]. The other main evaluation method involves testing tools with real developers in user studies, supporting a richer scope where information may not be directly accurate but nonetheless useful for some cognitive processes [Lerner et al. 2007; Seidel, Jhala, et al. 2016, 2018; Seidel, Sibghat, et al. 2017]. These evaluations therefore differ substantively in approach as well as goal (analyzing a FL tool vs connections between language design and error messages) as compared to this dissertation.

7.2.2 User Studies Investigating Pragmatics and Debugging

More broadly, the literature includes a wealth of user studies aiming to understand developer efficiency, the usability of tools, and various diverse aspects of developer process while programming; of those, user studies around debugging appear to be the most relevant to

this dissertation. Specifically, there are user studies investigating developer use of error messages and debugging tools or the debugging process broadly (e.g. Barik et al. [2018], Kume et al. [2016], Marceau et al. [2011b], Reichl et al. [2023], Silva et al. [2018], and Soremekun et al. [2023]). Notably, Marceau et al. [2011a] proposes a general rubric for evaluating the effectiveness of error messages for novice programmers based on their behavior in response to a message. Also notable, Schwerter [2023] proposes a slicing-based debugging aid for gradually typed programs and evaluates it with a user study. These user studies contribute an important lens of understanding the practical realities of software engineering and debugging, however it is of an entirely different (and complementary) nature to the focus of this dissertation. In particular, those studies examine the behavior of *human programmers*, with the goal of better understanding how they think and interact with debugging tools or information. In contrast, this dissertation studies how *language features and designs* affect the quality of debugging information produced by *programs*—quality as judged by the idealized blame-shifting procedure—with the goal of understanding the effects of linguistic design choices within this scope of debugging information. The connection from that goal to programmers is therefore tangential; the results of this dissertation's evaluations show that procedure to be reliably successful using blame in our test programs, suggesting that the procedure itself may be a useful tool for explaining (teaching) the meaning of blame in terms of concrete examples. Ultimately however, this dissertation does not draw any connections from its results to developer behavior.

7.3 Methodological Inspirations for Scenario Generation

The scenario corpus generation processes described in this dissertation draw upon two well-studied techniques from the Software Engineering literature: mutation and software compo-

nent adaptation.

All of the experiments of chapters 4-6 obtain faulty programs using mutation. Research on mutation testing began with the work of DeMillo, Guindi, et al. [1988] and DeMillo, Richard J. Lipton, et al. [1978] and Richard J Lipton [1971], and has since seen significant interest in the field of software engineering. Y. Jia and Harman [2011] provide a cogent overview of the history of mutation testing, its prevalent techniques, and its limitations. While mutation testing was first developed in the context of imperative programming languages, Le et al. [2014] describe the application of mutation testing techniques to higher order functional programs and demonstrate its effectiveness. However, the applicability of mutation testing techniques to generate faults in place of real faults in research is not immediately clear, and the kinds of faults generated by mutation are often quite distinct from those in real programs, as described by Gopinath, Jensen, et al. [2014]. On the other hand, Just et al. [2014] describe the traditional use of mutation testing for fault injection, and provide empirical evidence that such faults effectively simulate real faults in the context of test suite evaluation.

Finally, chapter 6's debugging scenario construction incorporates techniques from software component adaptation [Keller and Hölzle 1998; Mätzel and Schnorf 1997]. The idea of component adaptation is to reuse existing components in a software system for new purposes, adapting the component to the new interface or to provide new functionality by creating an *adapter layer* implementing the required changes. The motivation is that developers can thus reuse software components (reducing duplicated work and maintenance burden) while still maintaining separation of concerns and without needing to modify the original component (and possibly introduce bugs in along the way). The adapter layers of chapter 6 are essentially an application of this idea, but instead of adding functionality, our adapters change the original component's behavior in small ways to match the type mutations we

introduce in the component's type interface.

CHAPTER 8
CONCLUSION

In summary, this dissertation provides arguments in support of the thesis, restated from chapter 1:

> Evaluating the pragmatics of debugging with contracts and gradual types, with the rational programmer, provides evidence that contract-based semantics and blame are useful for debugging.

As primary evidence in support of the thesis, the prior chapters describe the design, implementation, and results of three experiments evaluating the pragmatics of debugging with contracts and gradual typing. In the case of contracts (chapter 4), the rational programmer helps reveal that while carefully-designed blame information does live up to its hypothesized debugging benefits, primitive stacktrace information appears to do so nearly as well. In the case of gradual typing, the evaluation results suggest that when mistakes occur in code (chapter 5), the specialized error information of academic approaches provides marginally better debugging help. In the case of gradual typing when mistakes occur in type annotations (chapter 6), however, the results suggest that the special error information does offer valuable debugging help.

At the meta level, these results demonstrate the value of the rational programmer method and point to several directions for future investigations. The following subsections consider a few such directions.

8.0.1 Future Work

8.0.1.1 Do the Results of These Experiments Apply to TypeScript?

Despite the strong similarities between the type systems of Typed Racket and TypeScript, it remains open whether the insights concerning the former—from chapters 5 and 6—apply to the latter, too. Confirming them in that setting presents an obvious line of future work starting from the results in this dissertation; in particular, this kind of evaluation is necessary to fully understand how the Natural and Erasure semantics can work together in practical implementations of gradual typing. Such an investigation requires a new backend for TypeScript and another rational programmer experiment.

8.0.1.2 More Realistic Notions of Cost

Future work should refine the cost aspect of the rational-programmer investigation, specifically cost as in developer time. The rational programmer, as instantiated in this dissertation, 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.

8.0.1.3 How Does Blame Fare with Modal Checking?

A classic developer request in the face of the performance overhead of contracts is a way to disable contracts in production. Dimoulas, Findler, and Felleisen [2013] suggests a more principled approach than a manual switch called option contracts, which support programmatically disabling and enabling contract checks. An extension of this idea is to define *modalities*, which are declarative policies for checking contracts alongside the contracts themselves. An open question, however, is how such policies for occasionally skipping contract checks affect the detection of bugs and the value of blame and stacktrace information, as well as the performance impact of contract checking. An adaptation of the evaluation of chapter 4 could incorporate performance measurement as well (using the framework of Takikawa, Feltey, et al. [2016]) to investigate this type of question.

8.0.1.4 How Does Blame Fare with Buggy Contracts?

Along the lines of the pragmatics question that chapter 6 targets, an open question following the results of chapter 4 is whether and how blame from contract systems may offer useful information in programs with buggy contracts. This is an interesting and challenging direction for future work, because unlike in the setting of gradual types, there is no type checker to call out mismatches between correct and incorrect contracts. Consider for example a program with a buggy function argument contract; even at the top of the configuration lattice for the program, blame from that contract never points to the module providing the function with that contract. Of course, the designers of contract systems are aware of this possibility, and so all blame messages in Racket bear the warning “assuming the contract is correct.” A debugging procedure that accounts for possibly-buggy contracts is therefore not a straightforward adaptation of those presented in this dissertation. At the very least, we

conjecture that it would need to heed both blame and this warning in some kind of balance, perhaps incorporating some common-sense heuristics to translate or reinterpret blame based on extra information in the error and higher-level knowledge of the program structure.

8.0.1.5 Formalizing the Blame Shifting Process in Terms of Decision Theory

This dissertation uses several variations on the relatively simple, deterministic blame shifting algorithm for locating bugs using various possibly-unreliable sources of information, the essence of which is inspired by the theory of blame. An interesting direction for future work is to formulate the task instead on the basis of decision theory, framing the rational programmer as a rational actor in the decision-theoretic sense. This alternative framing would allow, among other things, calculating theoretical upper and lower bounds on the successfulness of different debugging strategies, which could serve as useful context for interpreting the results of experiments like those in chapters 4-6. This is an attractive direction entailing many challenges, not the least of which is understanding what aspects of the debugging process and information can usefully be modeled as a decision problem.

REFERENCES

Hiralal Agrawal. 1991. “Towards Automatic Debugging of Computer Programs.” Ph.D. Dissertation. Purdue University.

Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. 1995. “Fault Localization Using Execution Slices and Dataflow Tests.” In: *Proceedings of Sixth International Symposium on Software Reliability Engineering*, 143–151. https://doi.org/10.1109/ISSRE.1995.497652.

Deyaaeldeen Almahallawi. 2020. “Towards Efficient Gradual Typing via Monotonic References and Coercions.” Ph.D. Dissertation. Indiana University.

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

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

J.H. Andrews, L.C. Briand, and Y. Labiche. 2005. “Is Mutation an Appropriate Tool for Testing Experiments?” In: *ICSE*, 402–411. https://doi.org/10.1109/ICSE.2005.1553583.

Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. 2018. “How Should Compilers Explain Problems to Developers?” In: *FSE*, 633–643. https://doi.org/10.1145/3236024.3236040.

Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. 2017. “Sound Gradual Typing: only Mostly Dead.” *PACMPL*, 1, OOPSLA, 54:1–54:24. https://dl.acm.org/doi/10.1145/3133878.

Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. 2015. “Pycket: a Tracing JIT for a Functional Language.” In: *IFL*, 22–34. https://dl.acm.org/doi/10.1145/2784731.2784740.

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, 148–175. https://doi.org/10.1080/08993408.2016.1225464.

Matthias Blume and David A. McAllester. 2006. "Sound and Complete Models of Contracts." *JFP*, 16, 4-5, 375–414. https://doi.org/10.1017/S0956796806005971.

David Broman and Jeremy G. Siek. 2017. "Gradually Typed Symbolic Expressions." In: *PEPM*, 15–29. https://dl.acm.org/doi/10.1145/3162068.

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, 15:1–15:29. https://doi.org/10.1145/3158103.

John Peter Campora, Sheng Chen, and Eric Walkingshaw. 2018. "Casts and Costs: Harmonizing Safety and Performance in Gradual Typing." *PACMPL*, 2, ICFP, 98:1–98:30. https://dl.acm.org/doi/10.1145/3236793.

John Peter Campora, Mohammad Wahiduzzaman Khan, and Sheng Chen. 2024. "Type-Based Gradual Typing Performance Optimization." *PACMPL*, 8, POPL, 89:1–89:33. https://dl.acm.org/doi/10.1145/3632931.

Giuseppe Castagna, Guillaume Duboc, Victor Lanvin, and Jeremy G. Siek. 2019. "A Space-efficient Call-by-value Virtual Machine for Gradual Set-Theoretic Types." In: *IFL*, 1–12. https://dl.acm.org/doi/10.1145/3412932.3412940.

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

Maxime Chevalier-Boisvert, Noah Gibbs, Jean Boussier, Si Xing (Alan) Wu, Aaron Patterson, Kevin Newton, and John Hawthorn. 2021. "YJIT: a Basic Block Versioning JIT Compiler for CRuby." In: *VMIL*, 25–32. https://dl.acm.org/doi/10.1145/3486606.3486781.

Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. "Format Abstraction for Sparse Tensor Algebra Compilers." *PACMPL*, 2, OOPSLA, 123:1–123:30 pages. https://doi.org/10.1145/3276493.

Fernando Cristiani and Peter Thiemann. 2021. "Generation of TypeScript Declaration Files from JavaScript Code." In: *MPLR*, 97–112. https://doi.org/10.1145/3475738.3480941.

Markus Degen, Peter Thiemann, and Stefan Wehr. 2008. "Contract Monitoring and Call-by-Name Evaluation." In: *Nordic Workshop on Programming Theory*.

Markus Degen, Peter Thiemann, and Stefan Wehr. 2010. "Eager and Delayed Contract Monitoring for Call-by-value and Call-by-Name Evaluation." *Logic and Algebraic Programming*, 79, 7, 515–549. https://doi.org/10.1016/j.jlap.2010.07.006.

Markus Degen, Peter Thiemann, and Stefan Wehr. 2012. "The Interaction of Contracts and Laziness." In: *PEPM*, 97–106. https://doi.org/10.1145/2103746.2103766.

Markus Degen, Peter Thiemann, and Stefan Wehr. 2009. "True lies: Lazy Contracts for Lazy Languages (Faithfulness is Better Than Laziness)." In: *4. Arbeitstagung Programmiersprachen*.

Richard A. DeMillo, Dana S. Guindi, Kim King, Mike M. McCracken, and Jefferson A. Offutt. 1988. "An Extended Overview of the Mothra Software Testing Environment." In: *Software Testing, Verification, and Analysis*, 142–151. https://doi.org/10.1109/WST.1988.5369.

Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. "Hints on Test Data Selection: Help for the Practicing Programmer." *Computer*, 11, 4, 34–41. https://doi.org/10.1109/C-M.1978.218136.

Pyret Developers. 2018. *Pyret Programming Language*. http://www.pyret.org/. (2018).

Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. "Precise Reasoning for Programs Using Containers." In: *POPL*, 187–200. https://doi.org/10.1145/1926385.1926407.

Christos Dimoulas and Matthias Felleisen. 2011. "On Contract Satisfaction in a Higher-Order World." *TOPLAS*, 33, 5, 16:1–16:29. https://doi.org/10.1145/2039346.2039348.

Christos Dimoulas, Robert Bruce Findler, and Matthias Felleisen. 2013. "Option Contracts." In: *OOPSLA*, 475–494. https://doi.org/10.1145/2509136.2509548.

Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. 2011. "Correct Blame for Contracts: no More Scapegoating." In: *POPL*, 215–226. `https://doi.org/10.1145/1926385.1926410`.

Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. 2014. "Declarative Policies for Capability Control." In: *Computer Security Foundations Symposium (CSF)*, 3–17. `https://doi.org/10.1109/CSF.2014.9`.

Christos Dimoulas, Max S. New, Robert Bruce Findler, and Matthias Felleisen. 2016. "Oh Lord, Please Don't Let Contracts be Misunderstood (Functional Pearl)." In: *ICFP*, 117–131. `https://doi.org/10.1145/2951913.2951930`.

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. "Complete Monitors for Behavioral Contracts." In: *ESOP*, 214–233. `https://doi.org/10.1007/978-3-642-28869-2_11`.

Tim Disney, Cormac Flanagan, and Jay McCarthy. 2011. "Temporal Higher-Order Contracts." In: *ICFP*, 176–188. `https://doi.org/10.1145/2034773.2034800`.

Grégory M. Essertel, Guannan Wei, and Tiark Rompf. 2019. "Precise Reasoning with Structured Time, Structured Heaps, and Collective Operations." *PACMPL*, 3, OOPSLA, 157:1–157:30 pages. `https://doi.org/10.1145/3360583`.

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`.

Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. "Collapsible Contracts: Fixing a Pathology of Gradual Typing." *PACMPL*, 2, OOPSLA, 133:1–133:27. `https://doi.org/10.1145/3276503`.

Robert Bruce Findler and Matthias Blume. 2006. "Contracts as Pairs of Projections." In: *FLP*, 226–241. `https://doi.org/10.1007/11737414_16`.

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

Robert Bruce Findler, Matthias Felleisen, and Matthias Blume. 2004. *An Investigation of Contracts as Projections*. Tech. rep. TR-2004-02. University of Chicago, Computer Science Department.

Robert Bruce Findler, Shu-yu Guo, and Anne Rogers. 2007. "Lazy Contract Checking for Immutable Data Structures." In: *IFL*, 111–128. `https://doi.org/10.1007/978-3-540-85373-2_7`.

Cormac Flanagan. 2006. "Hybrid Type Checking." In: *POPL*, 245–256. `https://doi.org/10.1145/1111037.1111059`.

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`.

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

Olek Gierczak, Lucy Menon, Christos Dimoulas, and Amal Ahmed. 2024. "Gradually Typed Languages Should Be Vigilant!" In: *OOPSLA (to appear)*.

Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. "Mutations: How Close are they to Real Faults?" In: *ISSRE*, 189–200. `https://doi.org/10.1109/ISSRE.2014.40`.

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

Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. 2005. "Fine-Grained Interoperability Through Mirrors and Contracts." In: *OOPSLA*, 231–245. `https://doi.org/10.1145/1094811.1094830`.

Michael Greenberg. 2015. "Space-Efficient Manifest Contracts." In: *POPL*, 181–194. `https://doi.org/10.1145/2676726.2676967`.

Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. 2012. "Contracts Made Manifest." *JFP*, 22, 3, 225–274. `https://doi.org/10.1017/S0956796812000135`.

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

Ben Greenman. 2022. "Deep and Shallow Types for Gradual Languages." In: *PLDI*, 580–593. https://doi.org/10.1145/3519939.3523430.

Ben Greenman. 2023. "GTP Benchmarks for Gradual Typing Performance." In: *REP*, 102–114. https://doi.org/10.1145/3589806.3600034.

Ben Greenman, Christos Dimoulas, and Matthias Felleisen. 2023. "Typed–Untyped Interactions: A Comparative Analysis." *TOPLAS*, 45, 4, 1–54, 1. https://doi.org/10.1145/3579833.

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

Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2019. "Complete Monitors for Gradual Types." *PACMPL*, 3, OOPSLA, 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 and Zeina Migeed. 2018. "On the Cost of Type-Tag Soundness." In: *PEPM*, 30–39. https://doi.org/10.1145/3162066.

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

Erin Greenwood-Thessman, Isaac Oscar Gariano, Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2021. "Naïve Transient Cast Insertion isn't (that) Bad." In: *Implementation, Compilation, Optimization of OO Languages, Programs and Systems*, 1–9. https://dl.acm.org/doi/10.1145/3464972.3472395.

Bastiaan J Heeren. 2005. "Top Quality Type Error Messages." Ph.D. Dissertation. Utrecht University.

Phillip Heidegger, Annette Bieniusa, and Peter Thiemann. 2012. "Access Permission Contracts for Scripting Languages." In: *POPL*, 111–122. https://doi.org/10.1145/2103656.2103671.

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. “Space-Efficient Gradual Typing.” *HOSC*, 23, 2, 167–189. https://doi.org/10.1007/s10990-011-9066-z.

Ralf Hinze, Johan Jeuring, and Andres Löh. 2006. “Typed Contracts for Functional Programming.” In: *FLOPS*, 208–225. https://doi.org/10.1007/11737414_15.

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, 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, 396–450. https://doi.org/10.1145/503502.503505.

Atsushi Igarashi, Peter Thiemann, Yuya Tsuda, Vasco T. Vasconcelos, and Philip Wadler. 2019. “Gradual Session Types.” *JFP*, 29, e17. https://www.cambridge.org/core/journals/journal-of-functional-programming/article/gradual-session-types/C6A5BBBD228A2C1B3E4A652E57B7CF89.

Limin Jia, Hannah Gommerstadt, and Frank Pfenning. 2016. “Monitors and Blame Assignment for Higher-Order Session Types.” In: *POPL*, 582–594. https://doi.org/10.1145/2837614.2837662.

Yue Jia and Mark Harman. 2011. “An Analysis and Survey of the Development of Mutation Testing.” *IEEE Transactions on Software Engineering*, 37, 5, 649–678. https://doi.org/10.1109/TSE.2010.62.

James A Jones, Mary Jean Harrold, and John Stasko. 2002. “Visualization of Test Information to Assist Fault Localization.” In: *ICSE*, 467–477. https://doi.org/10.1145/581339.581397.

René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. “Are Mutants a Valid Substitute for Real Faults in Software Testing?” In: *FSE*, 654–665. https://doi.org/10.1145/2635868.2635929.

Matthias Keil and Peter Theimann. 2015. “Blame Assignment for Higher-Order Contracts with Intersection and Union.” In: *ICFP*, 375–386. https://doi.org/10.1145/2784731.2784737.

Ralph Keller and Urs Hölzle. 1998. "Binary Component Adaptation." In: *ECOOP*, 307–329. https://doi.org/10.1007/BFb0054097.

Kenneth Knowles and Cormac Flanagan. 2010. "Hybrid Type Checking." *TOPLAS*, 32, 6, 1–34. https://doi.org/10.1145/1667048.1667051.

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, 90:1–90:25. https://doi.org/10.1145/3133914.

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

Izuru Kume, Masahide Nakamura, Yasuyuki Tanaka, and Etsuya Shibayama. 2016. "Evaluation of Diagnosis Support Methods in Program Debugging by Trace Analysis: An Exploratory Study." In: *ICIS*, 1–6. https://doi.org/10.1109/ICIS.2016.7550875.

Chris Lattner and Vikram Adve. 2005. "Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap." In: *PLDI*, 129–142. https://doi.org/10.1145/1065010.1065027.

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

Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2023. "How to Evaluate Blame for Gradual Types, Part 2." *PACMPL*, 7, ICFP, 194:1–194:28. https://doi.org/10.1145/3607836.

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

Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. 2014. "Mucheck: An Extensible Tool for Mutation Testing of Haskell Programs." In: *Software Testing and Analysis*, 429–432. https://doi.org/10.1145/2610384.2628052.

Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. "Searching for Type-Error Messages." In: *PLDI*, 425–434. https://doi.org/10.1145/1250734.1250783.

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

Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. 2016. "A Practical Framework for Type Inference Error Explanation." In: *OOPSLA*, 781–799. https://doi.org/10.1145/2983990.2983994.

Stefan Malewski, Michael Greenberg, and Éric Tanter. 2021. "Gradually Structured Data." *PACMPL*, 5, OOPSLA, 1–29. https://dl.acm.org/doi/10.1145/3485503.

Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011a. "Measuring the Effectiveness of Error Messages Designed for Novice Programmers." In: *Special Interest Group on Computer Science Education*, 499–504. https://doi.org/10.1145/1953163.1953308.

Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011b. "Mind Your Language: On Novices' Interactions with Error Messages." In: *Onward!* https://doi.org/10.1145/2048237.2048241.

Jacob Matthews and Robert Bruce Findler. 2007. "Operational Semantics for Multi-Language Programs." In: *POPL*, 3–10. https://doi.org/10.1145/1190216.1190220.

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

Kai-Uwe Mätzel and Peter Schnorf. 1997. *Dynamic Component Adaptation*. Tech. rep. Ubilab Technical Report 97.6.

Tommy McMichen, Nathan Greiner, Peter Zhong, Federico Sossai, Atmn Patel, and Simone Campanoni. 2024. "Representing Data Collections in an SSA Form." In: *CGO*, 308–321.

Bertrand Meyer. 1991. "Design by Contract." In: *Advances in Object-Oriented Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 1–50.

Bertrand Meyer. 1992. *Eiffel: The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN: 0-13-247925-7.

Bertrand Meyer. 1988. *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, USA.

Microsoft. N.d. *TypeScript*. Accessed February 23, 2023. `https://www.typescriptlang.org`.

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

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, 18:1–18:29 pages. `https://doi.org/10.1145/3290331`.

Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. 2016. "Extensible Access Control with Authorization Contracts." In: *OOPSLA*, 214–233. `https://doi.org/10.1145/2983990.2984021`.

Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. 2014. "SHILL: A Secure Shell Scripting Language." In: *OSDI*. Broomfield, CO, 183–199. ISBN: 978-1-931971-16-4. `https://doi.org/10.5555/2685048.2685063`.

Cameron Moy, Christos Dimoulas, and Matthias Felleisen. 2024. "Effectful Software Contracts." *PACMPL*, 8, POPL, 88:1–88:28. `https://doi.org/10.1145/3632930`.

Cameron Moy and Matthias Felleisen. 2023. "Trace Contracts." *JFP*, 33, e14. `https://doi.org/10.1017/S0956796823000096`.

Cameron Moy, Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2021. "Corpse Reviver: Sound and Efficient Gradual Typing via Contract Verification." *PACMPL*, 5, POPL, 53:1–53:28. `https://dl.acm.org/doi/10.1145/3434334`.

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

Fabian Muehlboeck and Ross Tate. 2021. "Transitioning from Structural to Nominal Code with Efficient Gradual Typing." *PACMPL*, 5, OOPSLA, 127:1–127:29. https://dl.acm.org/doi/10.1145/3485504.

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. "Producing Wrong Data without Doing Anything Obviously Wrong!" In: *ASPLOS*, 265–276. ISBN: 9781605584065. https://doi.org/10.1145/1508244.1508275.

Max S. New, Eric Giovannini, and Daniel R. Licata. 2023. "Gradual Typing for Effect Handlers." *PACMPL*, 7, OOPSLA, 1758–1786. https://dl.acm.org/doi/10.1145/3622860.

Mike Papadakis and Yves Le Traon. 2015. "Metallaxis-FL: Mutation-Based Fault Localization." *Software Testing, Verification and Reliability*, 25, 5-7, 605–628. https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1509.

D. L. Parnas. 1972. "A Technique for Software Module Specification with Examples." *Communications of the ACM*, 15, 5:330–5:336. https://doi.org/10.1145/355602.361309.

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, 111:1–111:27 pages. https://doi.org/10.1145/3485488.

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

Aseem Rastogi, 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.

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

Jan Reichl, Stefan Hanenberg, and Volker Gruhn. 2023. "Does the Stream API Benefit from Special Debugging Facilities? A Controlled Experiment on Loops and Streams with

Specific Debuggers." In: *ICSE*, 576–588. https://doi.org/10.1109/ICSE48619.2023.00058.

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

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

Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. 2015. "Computational Contracts." *Science of Computer Programming*, 98, P3:360–P3:375. https://doi.org/10.1016/j.scico.2013.09.005.

Felipe Andres Bañados Schwerter. 2023. "A Formal Framework for Understanding Run-Time Checking Errors in Gradually Typed Languages." Ph.D. Dissertation. University of British Columbia.

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)." *JFP*, 28, e13. https://doi.org/10.1017/S0956796818000126.

Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. 2017. "Learning to Blame: Localizing Novice Type Errors with Data-Driven Diagnosis." *PACMPL*, 1, OOPSLA, 60:1–60:27. https://doi.org/10.1145/3138818.

Ehud Y. Shapiro. 1983. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA. ISBN: 0262192187.

Jeremy G. Siek and Walid Taha. 2006. "Gradual Typing for Functional Languages." In: *Scheme and Functional Programming. University of Chicago, TR-2006-06*.

Jeremy G. Siek and Manish Vachharajani. 2008. "Gradual Typing with Unification-Based Inference." In: *DLS*, 1–12. https://dl.acm.org/doi/10.1145/1408681.1408688.

Jeremy G. Siek and Philip Wadler. 2010. "Threesomes, with and Without Blame." In: *POPL*, 365–376. https://doi.org/10.1145/1706299.1706342.

Jeremy Siek, Peter Thiemann, and Philip Wadler. 2015. "Blame and Coercion: Together Again for the First Time." In: *PLDI*, 425–435. https://doi.org/10.1145/2737924.2737968.

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

Fabio Pereira da Silva, Higor Amario de Souza, and Marcos Lordello Chaim. 2018. "An Empirical Assessment of Visual Debugging Tools Effectiveness and Efficiency." In: *SCCC*, 1–8. https://doi.org/10.1109/SCCC.2018.8705160.

Ezekiel Soremekun, Lukas Kirschner, Marcel Böhme, and Mike Papadakis. 2023. "Evaluating the Impact of Experimental Assumptions in Automated Fault Localization." In: *ICSE*, 159–171. https://doi.org/10.1109/ICSE48619.2023.00025.

Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van straaten, Robby Findler, and Jacob Matthews. 2009. "Revised6 Report on the Algorithmic Language Scheme." *JFP*, 19, S1, 1–301. https://doi.org/10.1017/S0956796809990074.

T. Stephen Strickland, Christos Dimoulas, Asumu Takikawa, and Matthias Felleisen. 2013. "Contracts for First-Class Classes." *TOPLAS*, 35, 3, 11:1–11:58. https://doi.org/10.1145/2518189.

T. Stephen Strickland and Matthias Felleisen. 2009a. "Contracts for First-Class Modules." In: *DLS*, 27–38. https://doi.org/10.1145/1640134.1640140.

T. Stephen Strickland and Matthias Felleisen. 2009b. "Nested and Dynamic Contract Boundaries." In: *IFL*, 141–158. https://doi.org/10.1007/978-3-642-16478-1_9.

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.

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

Cameron Swords. 2019. "A Unified Characterization of Runtime Verification Systems as Patterns of Communication." Ph.D. Dissertation. Indiana University.

Cameron Swords, Amr Sabry, and Sam Tobin-Hochstadt. 2018. "An Extended Account of Contract Monitoring Strategies as Patterns of Communication." *JFP*, 28, e4, 1–47. https://doi.org/10.1017/S0956796818000047.

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

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

Ferdian Thung, David Lo, Lingxiao Jiang, et al.. 2012. "Are Faults Localizable?" In: *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, 74–77. https://doi.org/10.5555/2664446.2664457.

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. 2010. "Logical Types for Untyped Languages." In: *ICFP*, 117–128. https://doi.org/10.1145/1863543.1863561.

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, 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.

Matías Toro and Éric Tanter. 2017. "A Gradual Interpretation of Union Types." In: *Proceedings of the 24th Static Analysis Symposium*, 382–404. https://doi.org/10.1007/978-3-319-66706-5_19.

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.

Alexi Turcotte, Aviral Goel, Filip Křikava, and Jan Vitek. 2020. "Designing Types for R, Empirically." *PACMPL*, 4, OOPSLA, 1–25. https://dl.acm.org/doi/10.1145/3428249.

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.

David Walker and J. Gregory Morrisett. 2000. "Alias Types for Recursive Data Structures." In: *TIC*, 177–206. https://dl.acm.org/doi/10.5555/647229.719257.

Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018. "Relational Program Synthesis." *PACMPL*, 2, OOPSLA, 155:1–155:27 pages. https://doi.org/10.1145/3276525.

Lucas Waye, Stephen Chong, and Christos Dimoulas. 2017. "Whip: Higher-Order Contracts for Modern Services." *PACMPL*, 1, ICFP, 36:1–36:28. https://doi.org/10.1145/3110280.

Jack Williams, J. Garrett Morris, and Philip Wadler. 2018. "The Root Cause of Blame: Contracts for Intersection and Union Types." *PACMPL*, 2, OOPSLA, 134:1–134:29. https://doi.org/10.1145/3276504.

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

Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. 2011. "Gradual Typestate." In: *ECOOP*, 459–483. https://doi.org/10.1007/978-3-642-22655-7_22.

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

Baijun Wu, John Peter Campora III, and Sheng Chen. 2017. "Learning User Friendly Type-Error Messages." *PACMPL*, 1, OOPSLA, 106:1–106:29. https://doi.org/10.1145/3133930.

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

Dana N. Xu, Simon Peyton Jones, and Koen Claessen. 2009. "Static Contract Checking for Haskell." In: *POPL*, 41–52. https://doi.org/10.1145/1480881.1480889.

Wenjia Ye and Bruno C. D. S. Oliveira. 2023. "Pragmatic Gradual Polymorphism with References." In: *Programming Languages and Systems*. Vol. 13990. Ed. by Thomas Wies. Springer Nature Switzerland, Cham, 140–167. https://doi.org/10.1007/978-3-031-30044-8_6.

Chenhao Zhang, Jason D. Hartline, and Christos Dimoulas. 2022. "Karp: a Language for NP Reductions." In: *ICSE*, 762–776. https://doi.org/10.1145/3519939.3523732.

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

Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2015. "Diagnosing Type Errors with Class." In: *PLDI*, 12–21. https://doi.org/10.1145/2737924.2738009.

APPENDIX A
STRATIFIED PROPORTION ESTIMATION

The goal of chapters 4-6's experiments are to estimate the proportion of scenarios for which each mode succeeds, or the proportion or scenarios for which a given mode improves over another. In statistical terms, this is the proportion estimation task, and since the experiments sample scenarios using stratified random sampling, there is a particular procedure for calculating the proportion estimate and margin of error that takes advantage of stratification. In detail, the algorithm for calculating an estimate of the proportion of a population satisfying some property, denoted p, based on a stratified set of samples is as follows.

1. Calculate the sample proportion within each group or stratum h, denoted p_h:

   $p_h = \frac{y_h}{n_h}$

   where y_h is the number of samples satisfying the property in stratum h and n_h is the total number of samples in stratum h.

2. Calculate the overall sample proportion by weighting each stratum's proportion according to the size of its sample:

   $p = \sum_h \frac{n_h}{n} p_h$

   where n is the total number of samples: $n = \sum_h n_h$.

3. Calculate the variance of each stratum's estimate, denoted s_h^2:

   $s_h^2 = \frac{n_h}{n_h - 1} p_h (1 - p_h)$

4. Calculate the variance of the overall proportion estimate, denoted s^2:

   $s^2 = \frac{1}{N^2} \left(\sum_h N_h^2 (1 - \frac{n_h}{N_h}) \frac{s_h^2}{n_h} \right)$

where N is the size of the total population, across all strata, and N_h is the size of each stratum's population.

5. Calculate the margin of error using the standard error, with a confidence level of 95%, denoted ME:
   $ME = z \times \sqrt{s^2}$
   where z is the z-score for a 95% confidence level: 1.96

To generalize this calculation to multi-tiered proportion estimation, we calculate proportion estimates from the leaves of the stratification tree (see figure 4.8) upward. That is, we treat each level of grouping as a single stratification, starting from mutants, and use the overall sample proportion p as p_h for the next level up. In detail, we perform the following sequence of calculations.

1. Calculate the proportion estimate and variance across the mutant strata according to the above procedure; call them p_m and s_m^2.

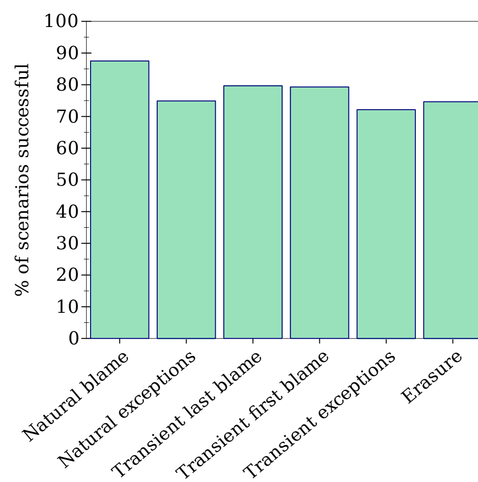
2. Calculate the proportion estimate and variance across the mutator strata according to the procedure, using each p_m as p_h and s_m^2 as s_h^2 above; call them p_M and s_M^2.

3. Calculate the proportion estimate and variance across the source program strata analogously; call them p_P and s_P^2.

4. Calculate the final, overall proportion estimate p and variance and s^2 analogously, and convert the variance to a final margin of error.

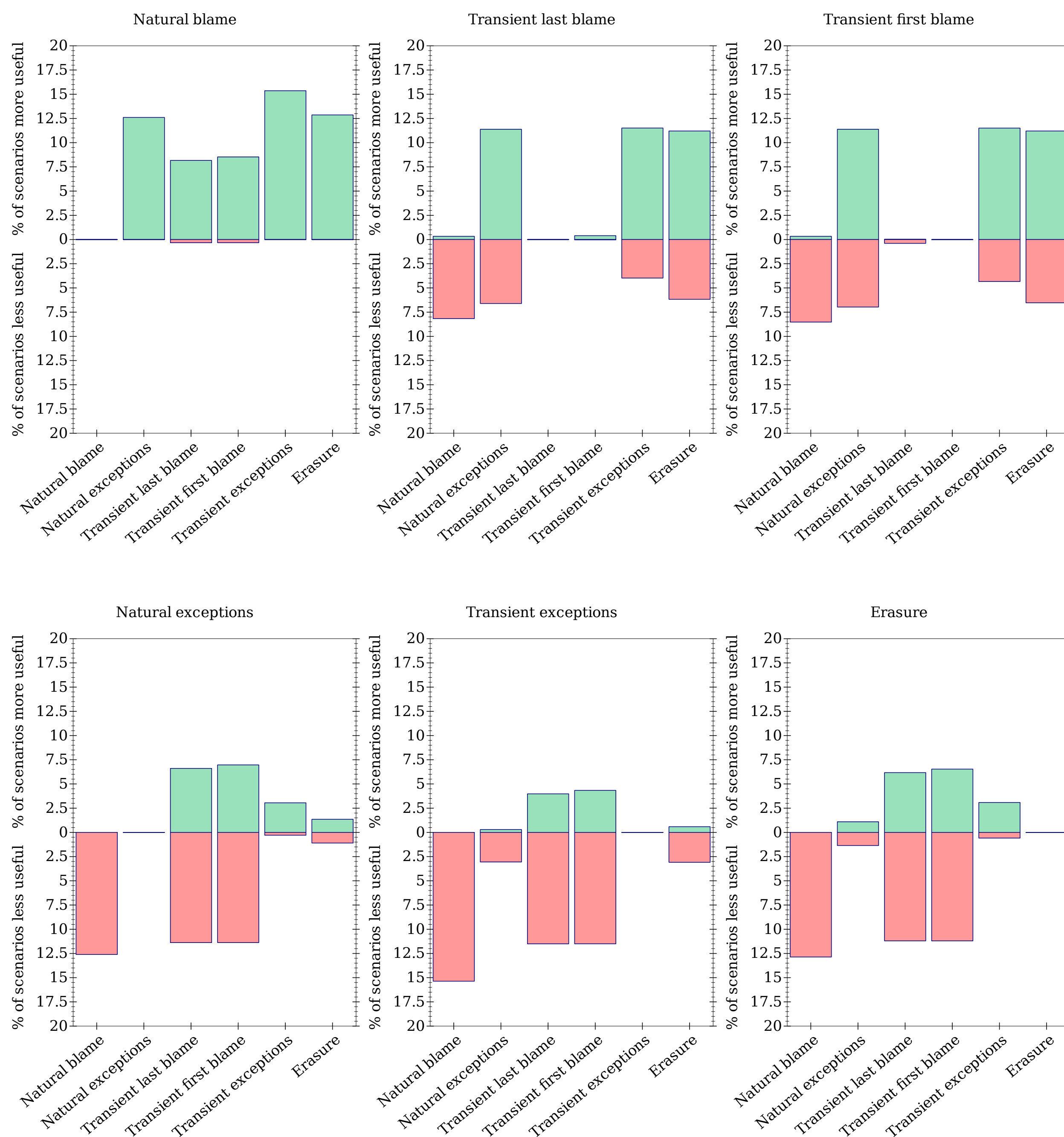
APPENDIX B
REVISITING EXPERIMENT 2 WITH THE NATURAL BIAS

A reproduction of the experiment of section 5 with the alternative bias demonstrates that the choice of bias does not significantly affect the section's conclusions. Specifically, we filter to select only scenarios that raise a run-time error under Natural. This appendix lists corresponding versions of the result figures of section 5. At a high level, the first takeaway is that the Natural-blame mode improves over all other modes in slightly more scenarios (on the order of a few percent). Somewhat more interestingly, the exception modes, including Erasure, improve over both Transient blame modes in about 5% more scenarios in this variation. Thus Transient's blame appears even less useful in comparison with simple stacktraces, but only by a small measure. Otherwise, the overall comparative success and usefulness comparisons follow the same basic patterns that inform the section's discussion and conclusions.



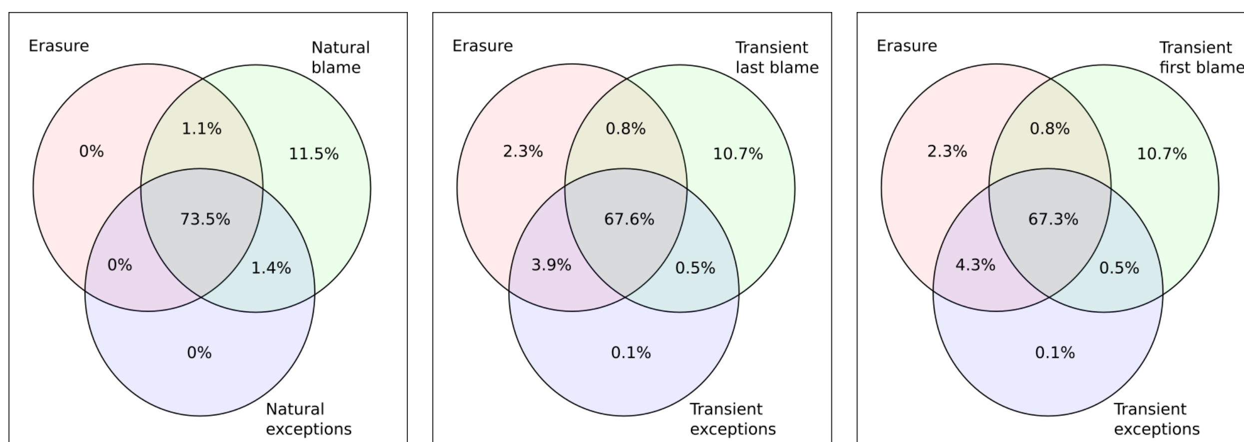
The upper bound margin of error is 0.02%.

Figure B.1: Percentage rates of success.



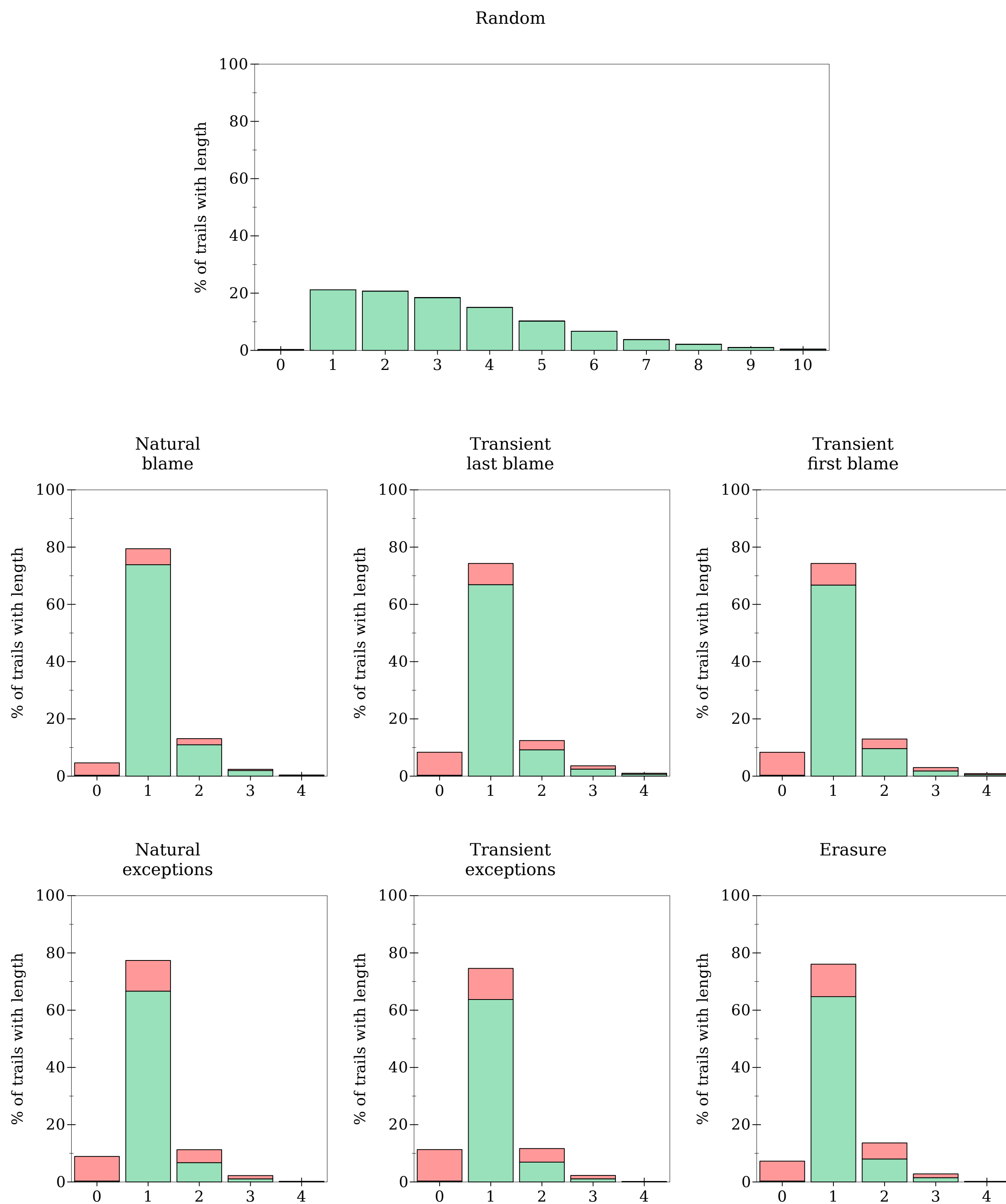
Each plot depicts a head-to-head comparison of the mode named above the plot vs. every other mode. The (green) portion above 0 is the estimated percentage of scenarios where the named mode is more useful than the other. The (red) portion below 0 is the estimated percentage of scenarios where the named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

Figure B.2: Head to head usefulness comparisons.



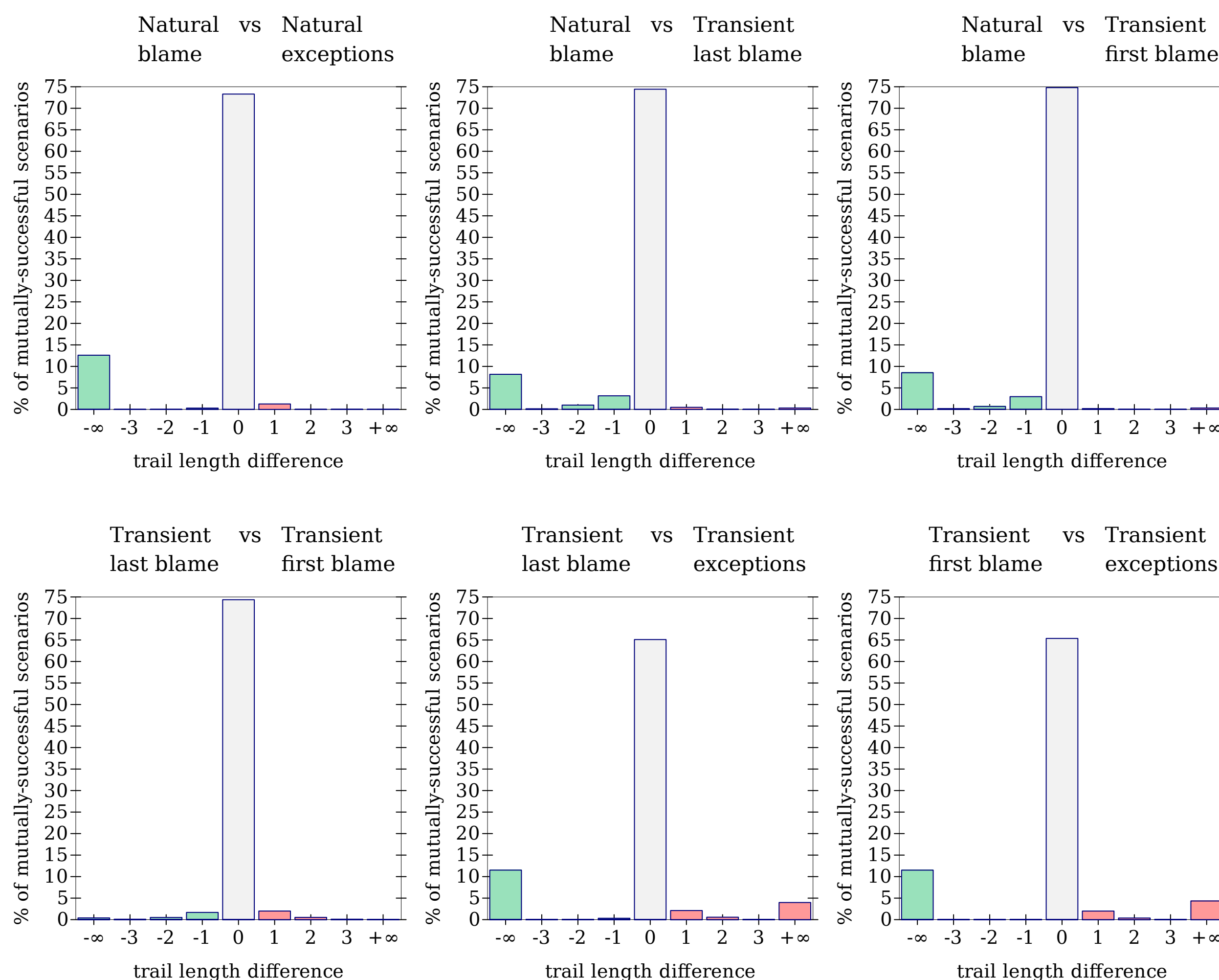
Each diagram shows the overlap of the successful scenarios for three modes. For example, in the leftmost diagram, all three modes succeed on the same scenario 75.7% of the time, only Natural and Natural exceptions succeed on 11.6% of the scenarios, only Natural and Erasure succeed on 1.8%, and Natural alone succeeds on 9.2%. The upper bound margin of error is 0.02%.

Figure B.3: Blame usefulness analysis



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

Figure B.4: Programmer effort

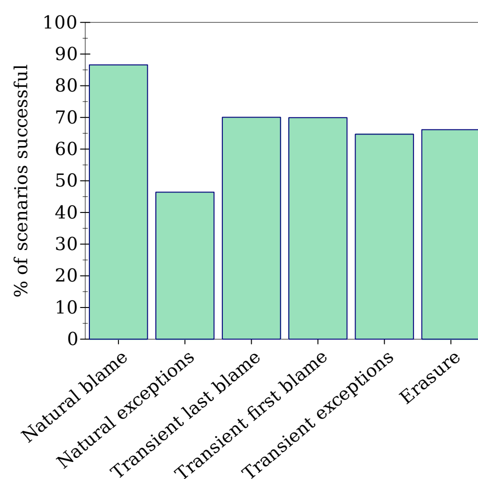


Each plot depicts the distribution of scenarios with trail length differences ranging from -3 to 3. A $-x$ difference denotes that the first mode's trail is x steps *shorter* than the second mode's trail for the same scenario; a positive difference denotes the inverse. A difference of ∞ indicates one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

Figure B.5: Effort comparisons

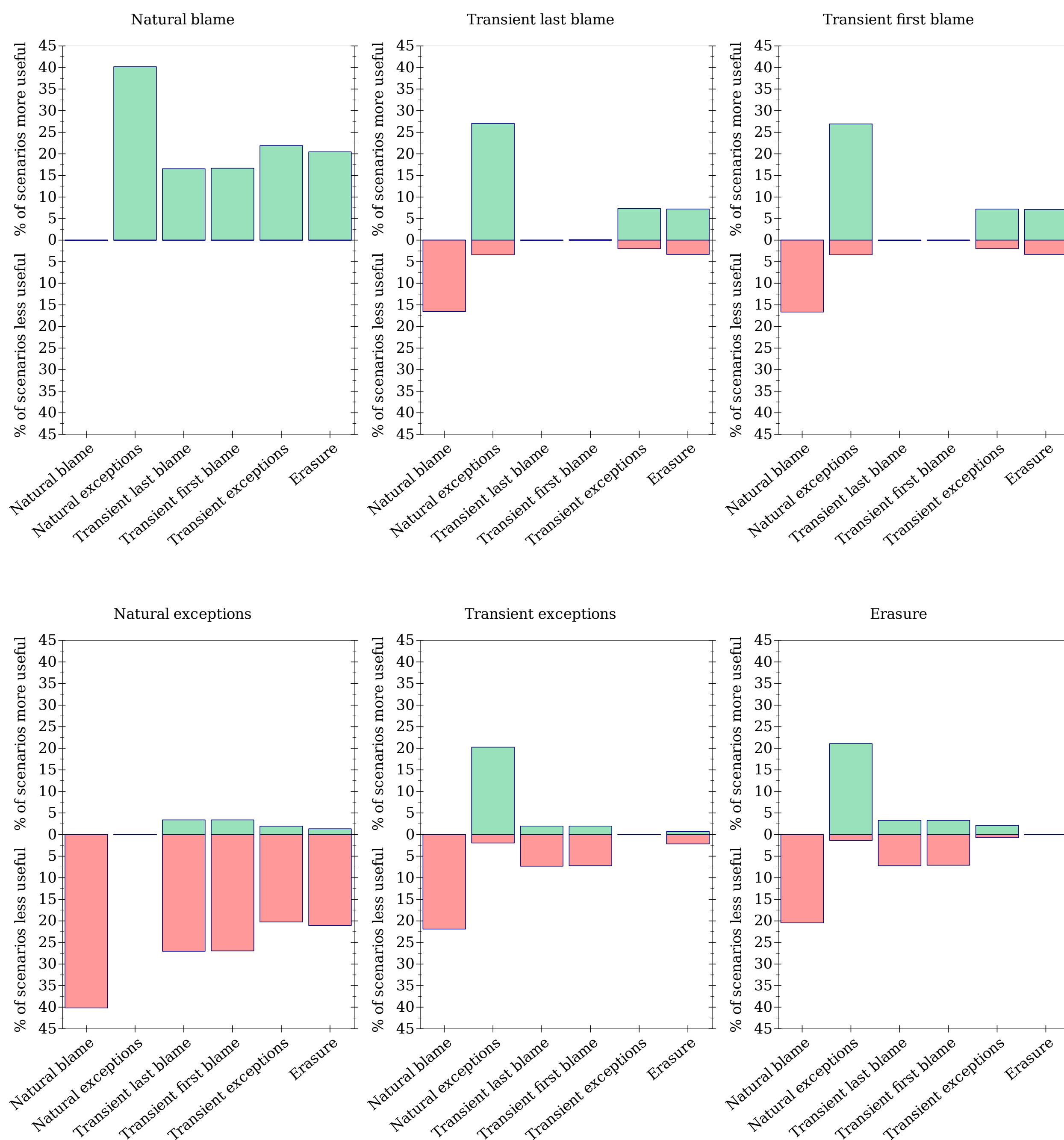
APPENDIX C
REVISITING EXPERIMENT 3 WITH THE ERASURE BIAS

A reproduction of the experiment of section 6 with the alternative bias demonstrates that the choice of bias does not affect the section's conclusions. Specifically, we filter to select only scenarios that raise a run-time error under Erasure. This appendix lists corresponding versions of the result figures of section 6. The overall takeaway is that there are only minor percentage-point variations between the results; the high level differences between modes entirely mirror those of section 6.6.



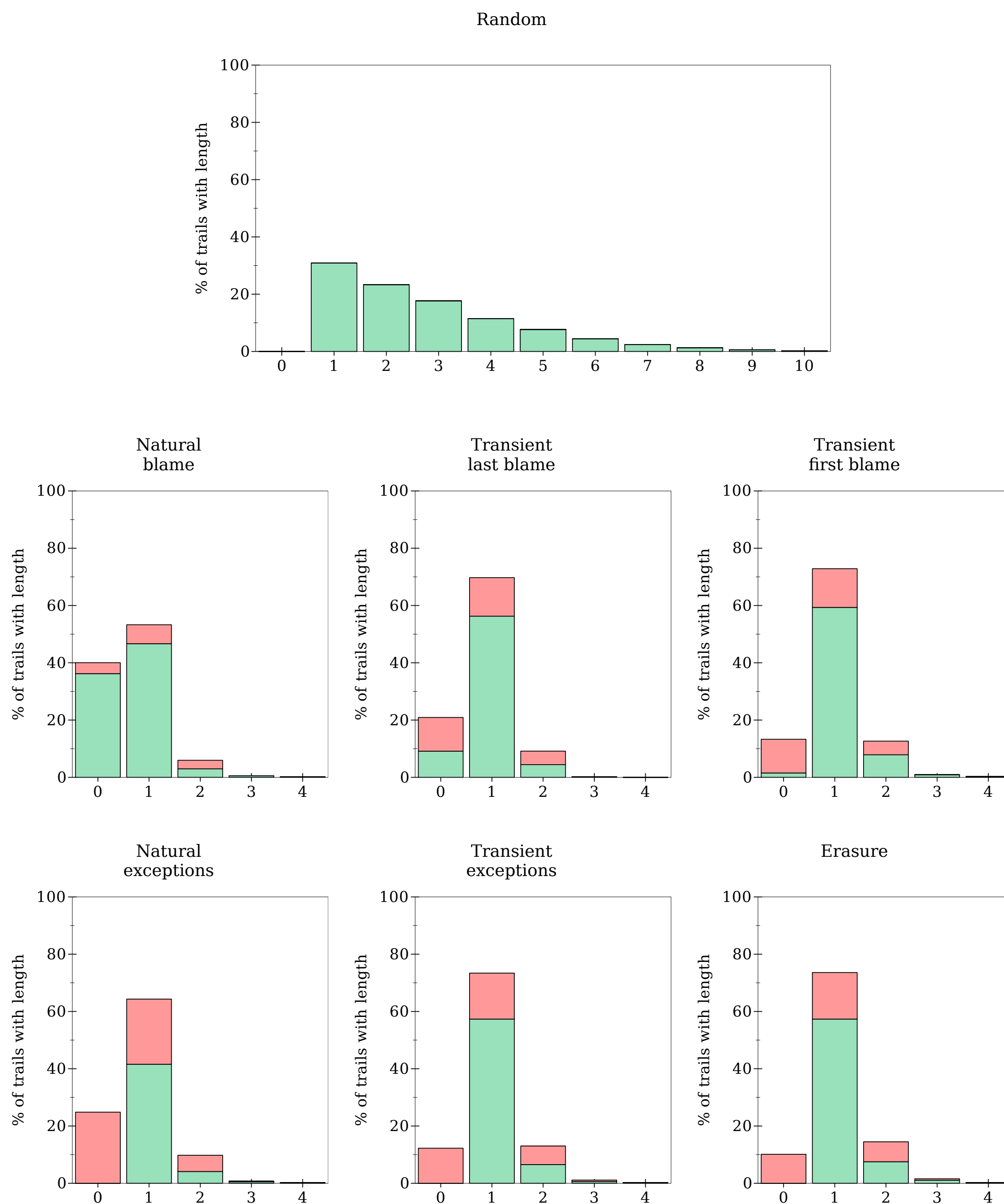
The upper bound margin of error is 0.08%.

Figure C.1: Percentage rates of success.



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%.

Figure C.2: Head to head usefulness comparisons.



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%.

Figure C.3: Trail length distributions per mode.