

How to Evaluate Blame for Gradual Types

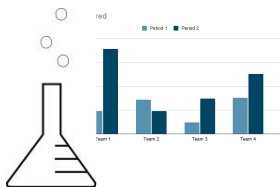
(Part 2)

Lukas Lazarek,
Ben Greenman, Matthias Felleisen,
Christos Dimoulas

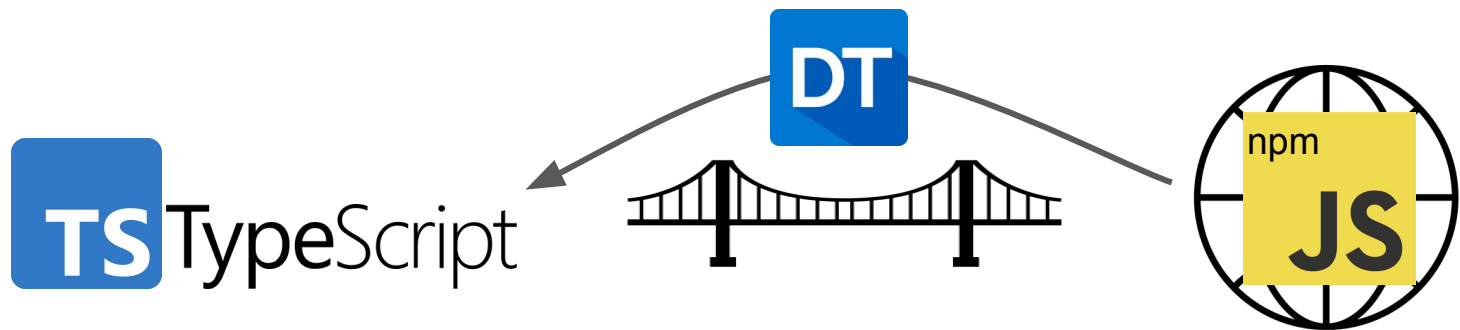


Research Question

How does Gradual Typing help with debugging
mistaken type annotations?



Mistaken Types: A Story of Gradual Typing in Practice



Gradual Typing in Practice

The screenshot shows the GitHub repository for DefinitelyTyped. The repository name is 'DefinitelyTyped / DefinitelyTyped'. It has 664 watchers, 29.4k forks, and 44.9k stars. The repository is public and has 15 branches and 1 tag. The commit history shows a recent merge PR #66046 by paulpestov. The file list includes .github, docs, scripts, types, .editorconfig, .eslintrc.cjs, .gitattributes, .gitignore, .npmrc, .prettierrignore, .prettierrc.json, .remarkrc.json, LICENSE, README.es.md, README.fr.md, README.it.md, and README.ja.md.



Type interfaces

```
sort:
  any[]
  (any any → boolean)
  → any[]
```



Type Interface Mistakes Happen Often

Mixed Messages: Measuring Conformance and Non-Interference in TypeScript¹

Jack Williams¹, Jakob Zalewski¹

- 1 University of Edinburgh, UK
- 2 University of Edinburgh, UK
- 3 University of Edinburgh, UK
- 4 University of Edinburgh, UK

Abstract

TypeScript participates in typing. The DefinitelyTyped repository hosts type declarations for thousands of JavaScript libraries. Given the lack of formal connection between the types and the corresponding code, a natural question is *are the types right?* An equally important question, as DefinitelyTyped and the libraries it supports change over time, is *how can we keep the types from becoming wrong?*

In this paper we offer Scotty, a tool that detects mismatches between the types and code in the DefinitelyTyped repository. More specifically, Scotty checks each package by converting its types into contracts and installing the contracts on the boundary between the library and its test suite. Running the test suite in this environment can reveal mismatches between the types and the JavaScript code. As automation and generality are both essential if such a tool is going to remain useful in the long term, we focus on techniques that sacrifice completeness, instead preferring to avoid false positives. Scotty currently handles about 20% of the 8000 packages on DefinitelyTyped (01% of the packages whose code is available and whose test suite passes).

Perhaps unsurprisingly, running the tests with these contracts in place revealed many errors in DefinitelyTyped. More surprisingly, despite the inherent limitations of the techniques we use, this exercise led to one hundred accepted pull requests that fix errors in DefinitelyTyped, demonstrating the value of this approach for the long-term maintenance of DefinitelyTyped. It also revealed a number of lessons about working in the JavaScript ecosystem and how details beyond the semantics of the language can be surprisingly important. Best of all, it also revealed a few places where programmers preferred incorrect types, suggesting some avenues of research to improve TypeScript.

CCS Concepts: Software and its engineering → Software verification and validation.

Additional Key Words and Phrases: Contracts, Gradual Typing, TypeScript, Definitely Typed, Buggy Types

ACM Reference Format: 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. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 142 (October 2022), 26 pages. <https://doi.org/10.1145/3563305>

1 Introduction

We have good news and enforce conformance be

1 Introduction

We have good news and enforce conformance be

This work was supported by EPSRC grants EP/R014434/1 and EP/R014434/2.

© Jack Williams, 2022. Licensed under CC BY 4.0 International.

Permission to make digital or physical copies of this work is granted by copyright for non-commercial use, provided that the copyright notice and this permission notice are included in any copy of the work.

For more information, contact the author(s).

1145/3563305/1

1145/3563305/1

1145/3563305/1

1145/3563305/1

Highly Illogical, Kirk: Spotting Type Mismatches in the Large Despite Broken Contracts, Unsound Types, and Too Many Linters

JOSHUA HOEFLICH, Northwestern University, USA
ROBERT BRUCE FINDLER, Northwestern University, USA
MANUEL SERRANO, Inria/UCa, France

The DefinitelyTyped repository hosts type declarations for thousands of JavaScript libraries. Given the lack of formal connection between the types and the corresponding code, a natural question is *are the types right?* An equally important question, as DefinitelyTyped and the libraries it supports change over time, is *how can we keep the types from becoming wrong?*

In this paper we offer Scotty, a tool that detects mismatches between the types and code in the DefinitelyTyped repository. More specifically, Scotty checks each package by converting its types into contracts and installing the contracts on the boundary between the library and its test suite. Running the test suite in this environment can reveal mismatches between the types and the JavaScript code. As automation and generality are both essential if such a tool is going to remain useful in the long term, we focus on techniques that sacrifice completeness, instead preferring to avoid false positives. Scotty currently handles about 20% of the 8000 packages on DefinitelyTyped (01% of the packages whose code is available and whose test suite passes).

Perhaps unsurprisingly, running the tests with these contracts in place revealed many errors in DefinitelyTyped. More surprisingly, despite the inherent limitations of the techniques we use, this exercise led to one hundred accepted pull requests that fix errors in DefinitelyTyped, demonstrating the value of this approach for the long-term maintenance of DefinitelyTyped. It also revealed a number of lessons about working in the JavaScript ecosystem and how details beyond the semantics of the language can be surprisingly important. Best of all, it also revealed a few places where programmers preferred incorrect types, suggesting some avenues of research to improve TypeScript.

CCS Concepts: Software and its engineering → Software verification and validation.

Additional Key Words and Phrases: Contracts, Gradual Typing, TypeScript, Definitely Typed, Buggy Types

ACM Reference Format: 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. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 142 (October 2022), 26 pages. <https://doi.org/10.1145/3563305>

1 INTRODUCTION

We built Scotty, a prototype infrastructure that scans every package in DefinitelyTyped, a large, Microsoft-maintained repository of TypeScript type declarations, in order to find inconsistencies between JavaScript implementations and their corresponding type declarations. Scotty is a mostly automatic system. It triggers errors when a mismatch is observed, but an absence of errors does not guarantee the correctness of a type declaration and, conversely, an error reported by Scotty is Authors' addresses: Joshua Hoeflich, Northwestern University, USA, Robert Bruce Findler, Northwestern University, USA, Manuel Serrano, Inria/UCa, France.

This work was supported by EPSRC grants EP/R014434/1 and EP/R014434/2.

© Jack Williams, 2022. Licensed under CC BY 4.0 International.

Permission to make digital or physical copies of this work is granted by copyright for non-commercial use, provided that the copyright notice and this permission notice are included in any copy of the work.

For more information, contact the author(s).

1145/3563305/1

1145/3563305/1

1145/3563305/1

1145/3563305/1

Type Test Scripts for TypeScript Testing

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

FRANK KROBIL, Universität Wien, Austria

Checking Correctness of TypeScript Interfaces for JavaScript Libraries

Asger Feldthaus
Aarhus University
asf@cs.au.dk

Anders Møller
Aarhus University
amoller@cs.au.dk

JavaScript programming language adds optional types, with support for interaction with existing libraries via interface declarations. Such declarations are written for hundreds of libraries, but they do not always match the code they are intended to describe. In this paper, we present a pragmatic approach to check correctness of interface declarations for JavaScript libraries. Our tool, called Scotty, checks each package by converting its types into contracts and installing the contracts on the boundary between the library and its test suite. Running the test suite in this environment can reveal mismatches between the types and the JavaScript code. As automation and generality are both essential if such a tool is going to remain useful in the long term, we focus on techniques that sacrifice completeness, instead preferring to avoid false positives. Scotty currently handles about 20% of the 8000 packages on DefinitelyTyped (01% of the packages whose code is available and whose test suite passes).

Perhaps unsurprisingly, running the tests with these contracts in place revealed many errors in DefinitelyTyped. More surprisingly, despite the inherent limitations of the techniques we use, this exercise led to one hundred accepted pull requests that fix errors in DefinitelyTyped, demonstrating the value of this approach for the long-term maintenance of DefinitelyTyped. It also revealed a number of lessons about working in the JavaScript ecosystem and how details beyond the semantics of the language can be surprisingly important. Best of all, it also revealed a few places where programmers preferred incorrect types, suggesting some avenues of research to improve TypeScript.

CCS Concepts: Software and its engineering → Software verification and validation.

Additional Key Words and Phrases: Contracts, Gradual Typing, TypeScript, Definitely Typed, Buggy Types

ACM Reference Format: Asger Feldthaus, Anders Møller. 2022. Checking Correctness of TypeScript Interfaces for JavaScript Libraries. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 142 (October 2022), 26 pages. <https://doi.org/10.1145/3563305>

This work was supported by EPSRC grants EP/R014434/1 and EP/R014434/2.

© Jack Williams, 2022. Licensed under CC BY 4.0 International.

Permission to make digital or physical copies of this work is granted by copyright for non-commercial use, provided that the copyright notice and this permission notice are included in any copy of the work.

For more information, contact the author(s).

1145/3563305/1

1145/3563305/1

1145/3563305/1

1145/3563305/1

1145/3563305/1

1145/3563305/1

Generation of TypeScript Declaration Files from JavaScript Code

Fernando Cristiano
Hochschule Karlsruhe
Karlsruhe, Germany
fernando.cristiano@hs-karlsruhe.de

Peter Thiemann
Albert-Ludwigs-Universität Freiburg
Freiburg, Germany
thiemann@informatik.uni-freiburg.de

Abstract
Developers are starting to write large and complex applica-

1 Introduction
JavaScript is the most popular language for writing web applications [7]. It is also increasingly used for back-end applications running in Node.js, a JavaScript-based server-side platform. JavaScript is appealing to developers because its forgiving dynamic typing enables them to create simple pieces of code very quickly and proceed on a trial-and-error basis.

JavaScript was never intended to be more than a scripting language and, thus, lacks features for maintaining and evolving large codebases. However, nowadays developers create large and complex applications in JavaScript. Mistakes such as mistyped property names and misunderstood or unexpected type coercions cause developers to spend a significant amount of time in debugging. There is ample evidence for such mishaps. For example, a JavaScript code blog¹ collects experiences from developers facing unexpected situations while programming in JavaScript. Listing 1 exposes some of these uninitiated JavaScript behaviors.

The cognitive load produced by these behaviors is mitigated in languages that use built-in tools based on type information. This insight motivated the creation of TypeScript, a superset of JavaScript with expressive type annotations [13]. It has become a widely used alternative among JavaScript developers, because it incorporates features that are helpful for developing and maintaining large applications [1]. TypeScript enables the early detection of several kinds of run-time errors and the integration of code intelligence tools like auto completion in an IDE.

Despite the advantages, it is unrealistic to expect the world to switch to TypeScript in a day. Therefore, existing JavaScript libraries can be used in a TypeScript project by adding a declaration file that describes the library's API in terms of types. The Definitely Typed repository [5] has been created as a community effort to collect declaration files for popular JavaScript libraries. At the time of writing it contains declaration files for more than 6000 libraries.

However, in order to provide useful type checking, it is beneficial for the compiler to know what types to expect from the JavaScript code. For this purpose, a subset of TypeScript is used for describing the types of functions and objects provided by libraries written in JavaScript. Such a description has been written for the TypeScript library, the library can be used by TypeScript programmers as though it were written in TypeScript. This language subset is used extensively and is an essential part of the TypeScript ecosystem. As an example, the Definitely Typed repository, which is a community effort to provide high quality TypeScript declaration files, at the time of writing contains declarations for over 200 libraries, comprising a total of over 100,000 effective lines of code in declaration files.

TypeScript declaration files are written by hand, and often by others than the authors of the JavaScript libraries. If the programmer makes a mistake in a declaration file, tools that depend on it will misbehave: the TypeScript type checker may report perfectly correct code, and code completion in IDEs may provide misleading suggestions. The TypeScript compiler makes an attempt to check the correctness of the declaration file with respect to the library implementation, and types are not checked at runtime, so any bugs in it

languages; for instance, the compiler can quickly catch obvious coding blunders, and editors can provide code completion. Types also carry documentation values, giving programmers a streamlined language with which to document APIs. TypeScript does not ensure type safety, in other words, the type system is unenforced by design. Even for programs that pass static type checking, it is possible that a variable at runtime has a value that does not match the type annotation. This is seen as a trade-off necessary to keep the type system unrestrictive.

TypeScript is designed to compile to JavaScript using a shallow translation process. Each TypeScript expression translates into one JavaScript expression, and a value in TypeScript is the same as a value in JavaScript. This makes it possible to mix TypeScript code with JavaScript code, in particular existing JavaScript libraries, without the use of a foreign function interface as known from other programming languages. However, in order to provide useful type checking, it is beneficial for the compiler to know what types to expect from the JavaScript code. For this purpose, a subset of TypeScript is used for describing the types of functions and objects provided by libraries written in JavaScript.

Such a description has been written for the TypeScript library, the library can be used by TypeScript programmers as though it were written in TypeScript. This language subset is used extensively and is an essential part of the TypeScript ecosystem. As an example, the Definitely Typed repository, which is a community effort to provide high quality TypeScript declaration files, at the time of writing contains declarations for over 200 libraries, comprising a total of over 100,000 effective lines of code in declaration files.

TypeScript declaration files are written by hand, and often by others than the authors of the JavaScript libraries. If the programmer makes a mistake in a declaration file, tools that depend on it will misbehave: the TypeScript type checker may report perfectly correct code, and code completion in IDEs may provide misleading suggestions. The TypeScript compiler makes an attempt to check the correctness of the declaration file with respect to the library implementation, and types are not checked at runtime, so any bugs in it

<https://github.com/borisyankov/DefinitelyTyped>

1145/3563305/1

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2022 Copyright held by the owner/authors.

2475-1421/2022/10-ART142

<https://doi.org/10.1145/3563305>

the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtime (MPLR '22), September 29–30, 2022, Montréal, Québec, Canada. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3473738.3480941>

1 Introduction

JavaScript is the most popular language for writing web applications [7]. It is also increasingly used for back-end applications running in Node.js, a JavaScript-based server-side platform. JavaScript is appealing to developers because its forgiving dynamic typing enables them to create simple pieces of code very quickly and proceed on a trial-and-error basis.

JavaScript was never intended to be more than a scripting language and, thus, lacks features for maintaining and evolving large codebases. However, nowadays developers create large and complex applications in JavaScript. Mistakes such as mistyped property names and misunderstood or unexpected type coercions cause developers to spend a significant amount of time in debugging. There is ample evidence for such mishaps. For example, a JavaScript code blog¹ collects experiences from developers facing unexpected situations while programming in JavaScript. Listing 1 exposes some of these uninitiated JavaScript behaviors.

The cognitive load produced by these behaviors is mitigated in languages that use built-in tools based on type information. This insight motivated the creation of TypeScript, a superset of JavaScript with expressive type annotations [13]. It has become a widely used alternative among JavaScript developers, because it incorporates features that are helpful for developing and maintaining large applications [1]. TypeScript enables the early detection of several kinds of run-time errors and the integration of code intelligence tools like auto completion in an IDE.

Despite the advantages, it is unrealistic to expect the world to switch to TypeScript in a day. Therefore, existing JavaScript libraries can be used in a TypeScript project by adding a declaration file that describes the library's API in terms of types. The Definitely Typed repository [5] has been created as a community effort to collect declaration files for popular JavaScript libraries. At the time of writing it contains declaration files for more than 6000 libraries.

However, in order to provide useful type checking, it is beneficial for the compiler to know what types to expect from the JavaScript code. For this purpose, a subset of TypeScript is used for describing the types of functions and objects provided by libraries written in JavaScript. Such a description has been written for the TypeScript library, the library can be used by TypeScript programmers as though it were written in TypeScript. This language subset is used extensively and is an essential part of the TypeScript ecosystem. As an example, the Definitely Typed repository, which is a community effort to provide high quality TypeScript declaration files, at the time of writing contains declarations for over 200 libraries, comprising a total of over 100,000 effective lines of code in declaration files.

TypeScript declaration files are written by hand, and often by others than the authors of the JavaScript libraries. If the programmer makes a mistake in a declaration file, tools that depend on it will misbehave: the TypeScript type checker may report perfectly correct code, and code completion in IDEs may provide misleading suggestions. The TypeScript compiler makes an attempt to check the correctness of the declaration file with respect to the library implementation, and types are not checked at runtime, so any bugs in it

languages; for instance, the compiler can quickly catch obvious coding blunders, and editors can provide code completion. Types also carry documentation values, giving programmers a streamlined language with which to document APIs. TypeScript does not ensure type safety, in other words, the type system is unenforced by design. Even for programs that pass static type checking, it is possible that a variable at runtime has a value that does not match the type annotation. This is seen as a trade-off necessary to keep the type system unrestrictive.

TypeScript is designed to compile to JavaScript using a shallow translation process. Each TypeScript expression translates into one JavaScript expression, and a value in TypeScript is the same as a value in JavaScript. This makes it possible to mix TypeScript code with JavaScript code, in particular existing JavaScript libraries, without the use of a foreign function interface as known from other programming languages. However, in order to provide useful type checking, it is beneficial for the compiler to know what types to expect from the JavaScript code. For this purpose, a subset of TypeScript is used for describing the types of functions and objects provided by libraries written in JavaScript.

Such a description has been written for the TypeScript library, the library can be used by TypeScript programmers as though it were written in TypeScript. This language subset is used extensively and is an essential part of the TypeScript ecosystem. As an example, the Definitely Typed repository, which is a community effort to provide high quality TypeScript declaration files, at the time of writing contains declarations for over 200 libraries, comprising a total of over 100,000 effective lines of code in declaration files.

TypeScript declaration files are written by hand, and often by others than the authors of the JavaScript libraries. If the programmer makes a mistake in a declaration file, tools that depend on it will misbehave: the TypeScript type checker may report perfectly correct code, and code completion in IDEs may provide misleading suggestions. The TypeScript compiler makes an attempt to check the correctness of the declaration file with respect to the library implementation, and types are not checked at runtime, so any bugs in it

The Academic Perspective on Type Mistakes



e.g. Reticulated Python



e.g. Typed Racket

Erasure

No dynamic checks for
type annotations

Stacktraces

Transient

Type assertions in typed code

Blames many

Natural

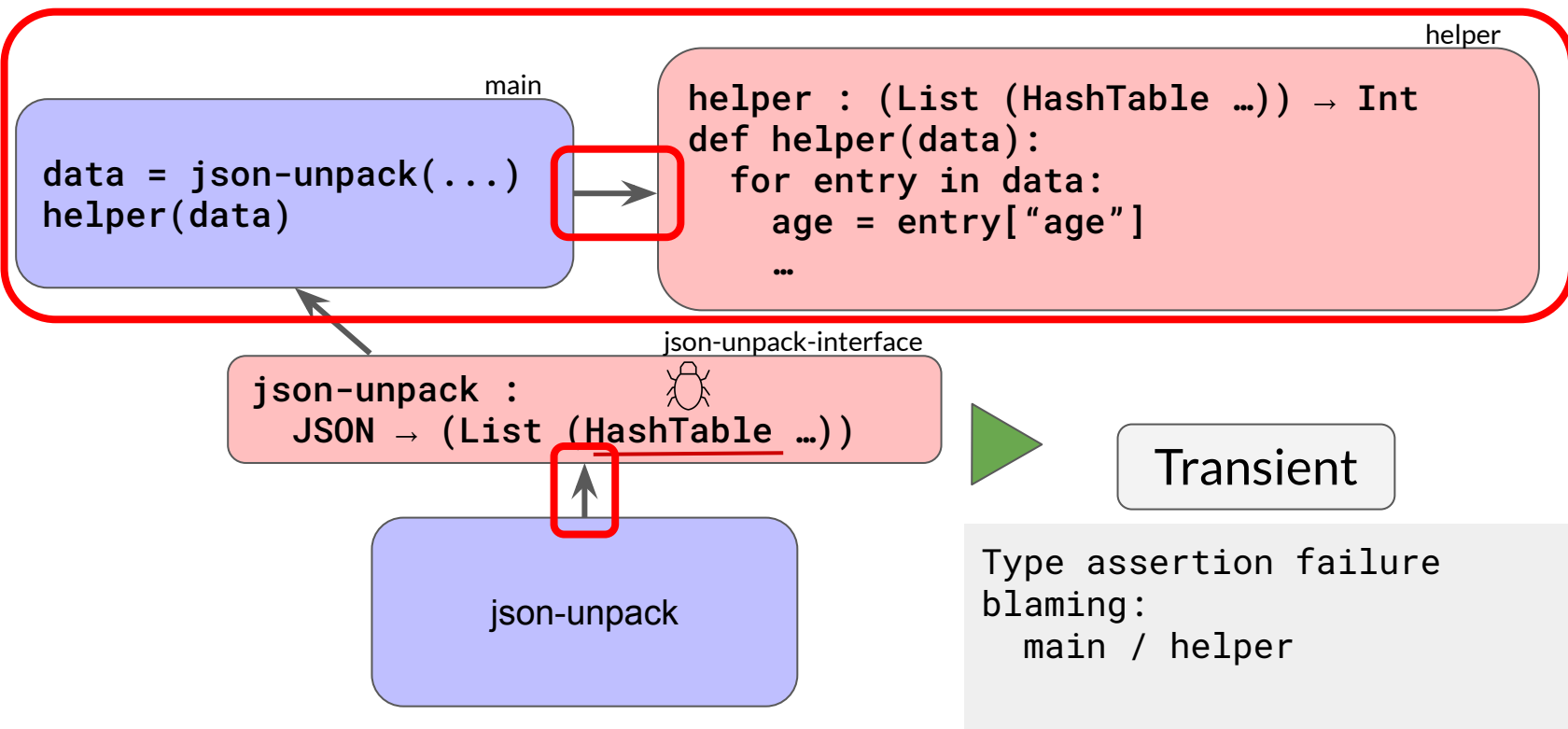
Higher order contracts

Blames one

Research Question (refined)

Given the same statics, which semantics for Gradual Typing provides better **error information** for systematically locating **type interface mistakes**?

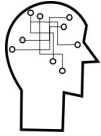
Comparing Each Semantics, by Example



Comparing Each Semantics, Systematically



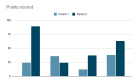
Hypothesis



Automated procedure(s)



Experiment testing procedures on real programs



Data providing evidence to support or refute the hypothesis

Does Blame Matter?

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

How to Evaluate Blame for Gradual Types

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

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

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

Additional Key Words and Phrases: gradual typing, blame

ACM Reference Format:
Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2021. How to Evaluate Blame for Gradual Types. *Proc. ACM Program. Lang.* 5, ICFP, Article 68 (August 2021), 29 pages. <https://doi.org/10.1145/3473737>

1 DOES BLAME MATTER

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

Industrial implementers of gradual typing systems have almost completely ignored blame assignment. Systems such as Flow, Hack, or TypeScript¹ exploit types for IDE actions and for finding

The original authors got this word wrong. A program has many components; blame helps identify a faulty one.

¹See <https://flow.org>, <https://hackmd.org>, and <https://www.typescriptlang.org>, respectively.

²Authors’ addresses: Lukas Lazarek, PLT @ Northwestern University, Evanston, Illinois, USA, lukas.lazarek@northwestern.edu; Ben Greenman, PLT @ Northeastern University, Boston, Massachusetts, USA, benjamin@greenman.org; Matthias Felleisen, PLT @ Northeastern University, Boston, Massachusetts, USA, matthiascs@cs.nyu.edu; Christos Dimoulas, PLT @ Northwestern University, Evanston, Illinois, USA, chris@northwestern.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner(s).

© 2021 Copyright held by the owner(s).
4475-4421/2021-08 ART64
<https://doi.org/10.1145/3473737>

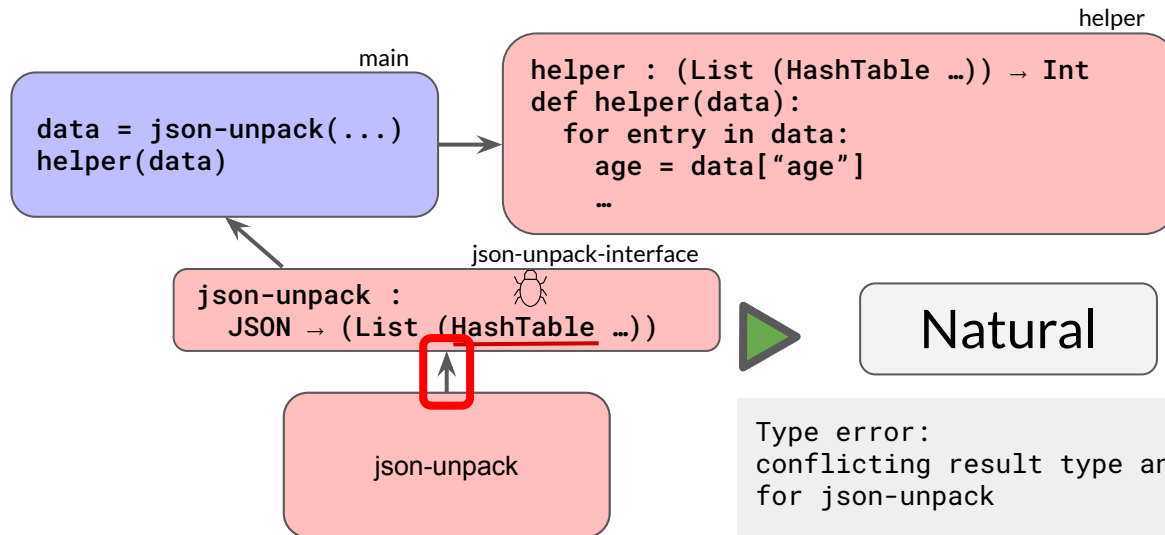
Proc. ACM Program. Lang., Vol. 5, No. ICFP, Article 68. Publication date: August 2021.

A Hypothesis, by Example



Hypothesis

Using the type system, error information can be translated into the location of type interface mistakes



Natural

Type error:
conflicting result type annotations
for json-unpack

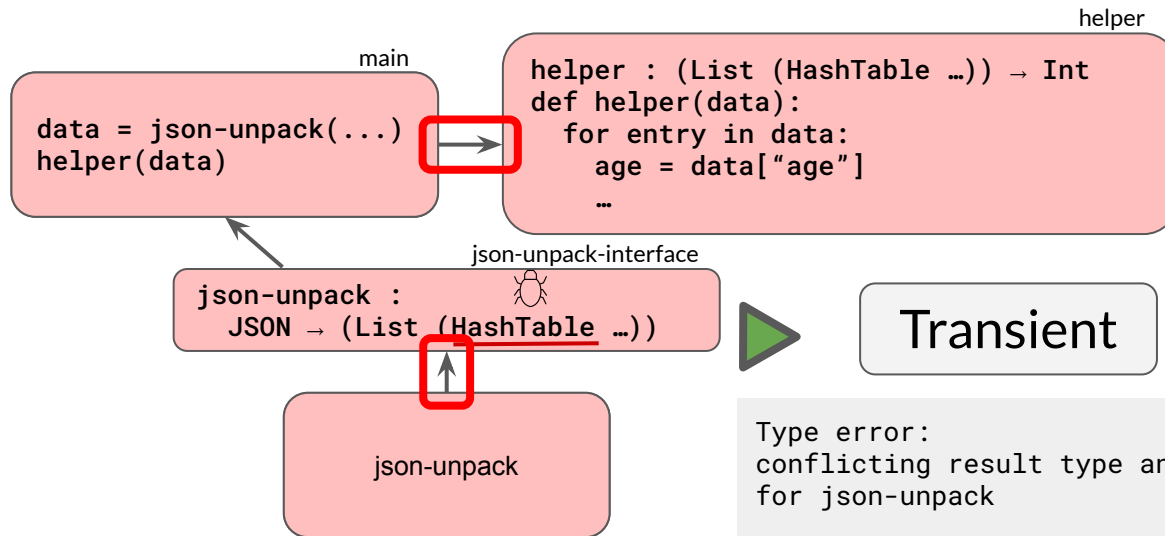


A Hypothesis, by Example



Hypothesis

Using the type system, error information can be translated into the location of type interface mistakes



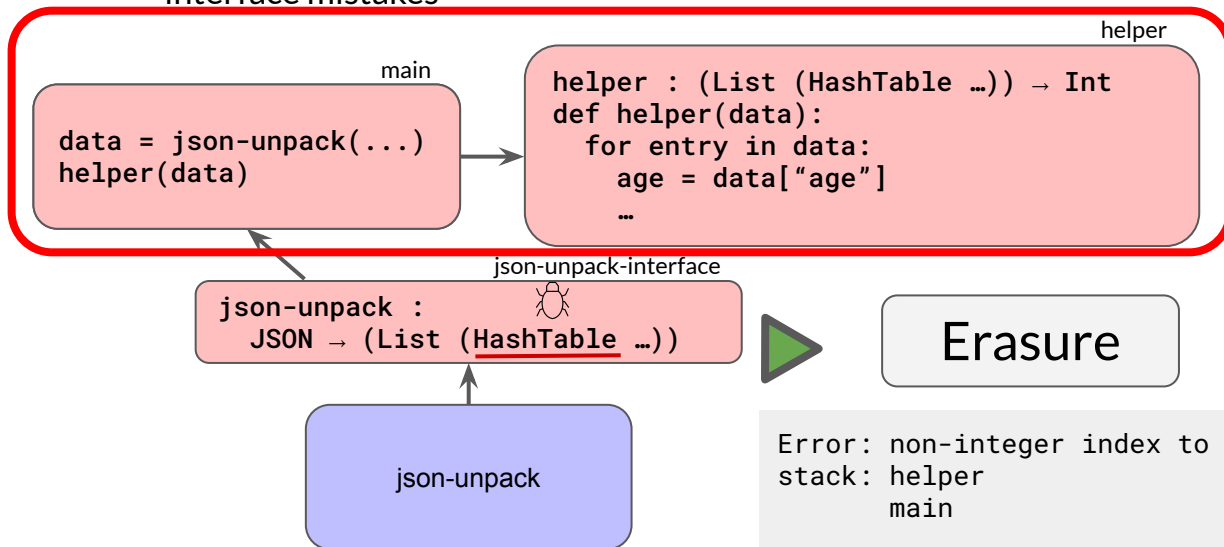
Transient

Type error:
conflicting result type annotations
for json-unpack

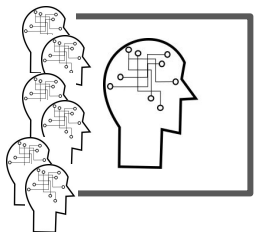
A Hypothesis, by Example



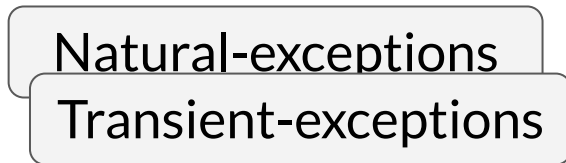
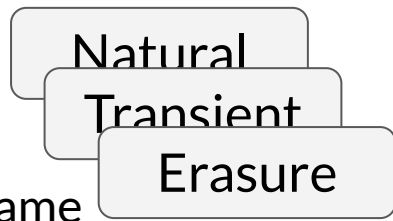
Hypothesis Using the type system, error information can be translated into the location of type interface mistakes



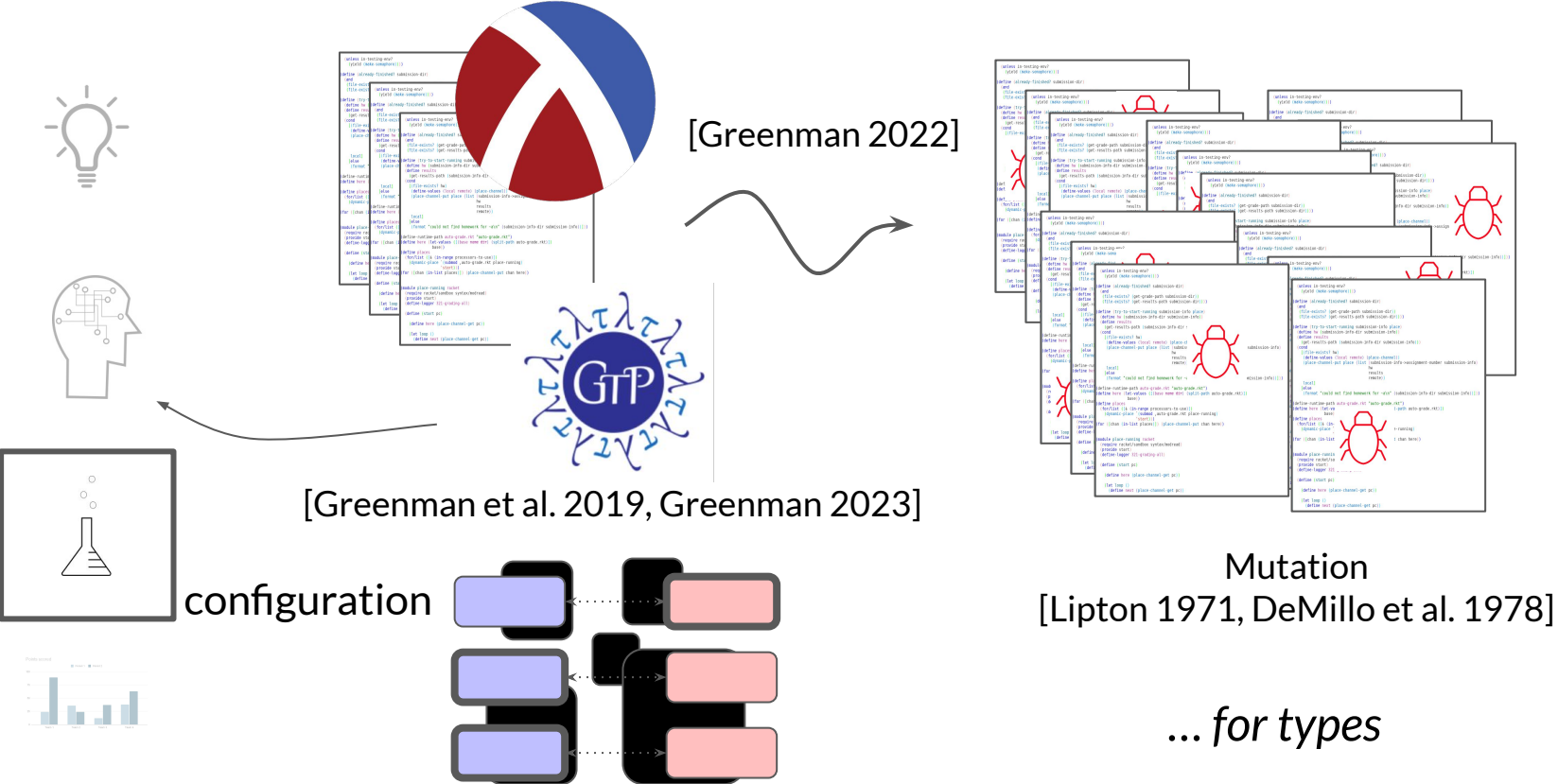
A Procedure Reifying Our Hypothesis



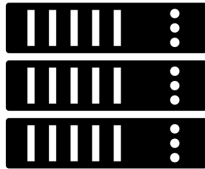
1. Run the program with Natural semantics to get blame
2. Identify the (untyped) blamed component
3. Try to type that component* (may fail)
4. Type-check the program
 - 4.1 If it type-checks: go to 1
 - 4.2 Otherwise: stop (success)



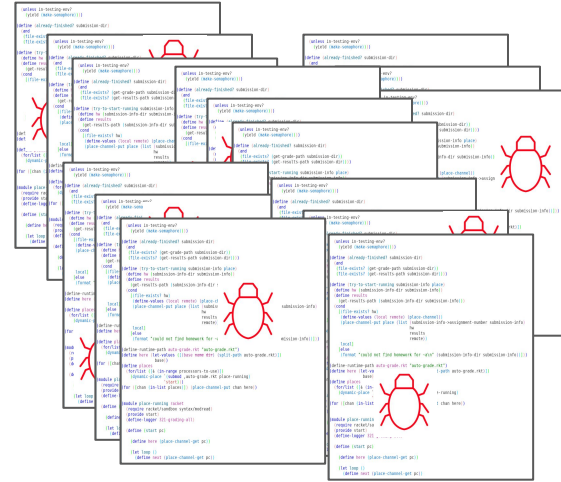
Creating an Experiment to Test Our Hypothesis



Creating an Experiment to Test Our Hypothesis

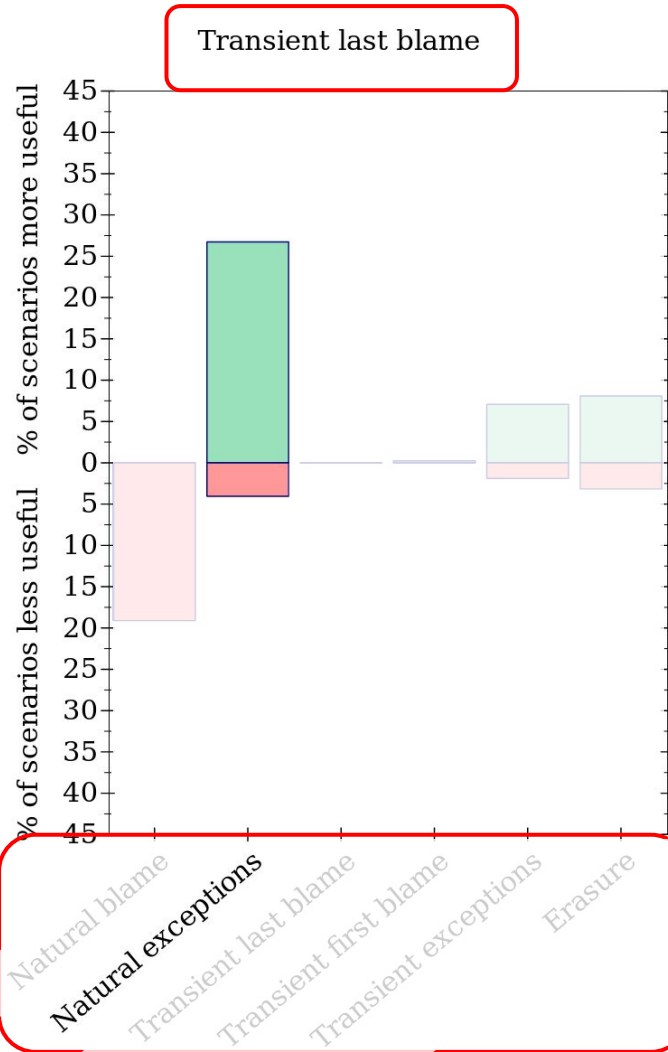
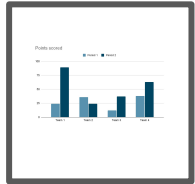


[... details omitted ...]

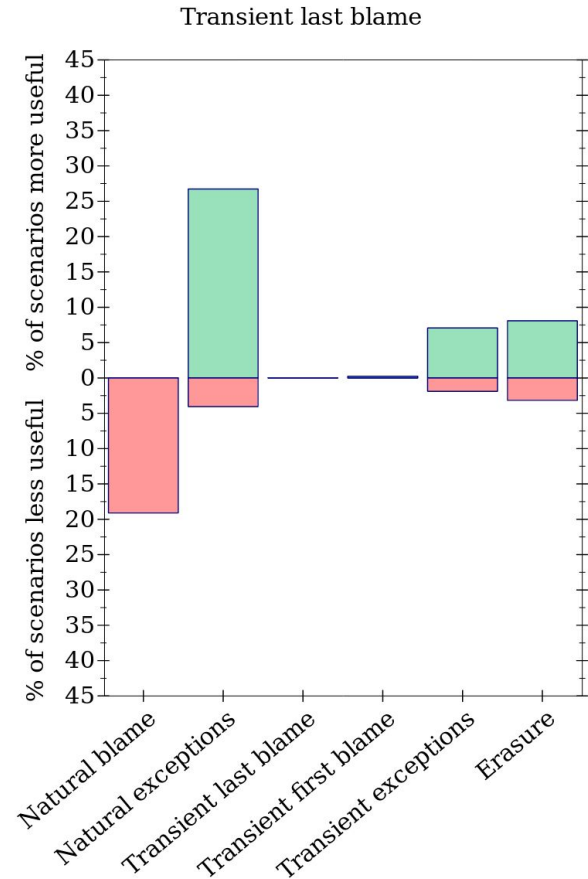
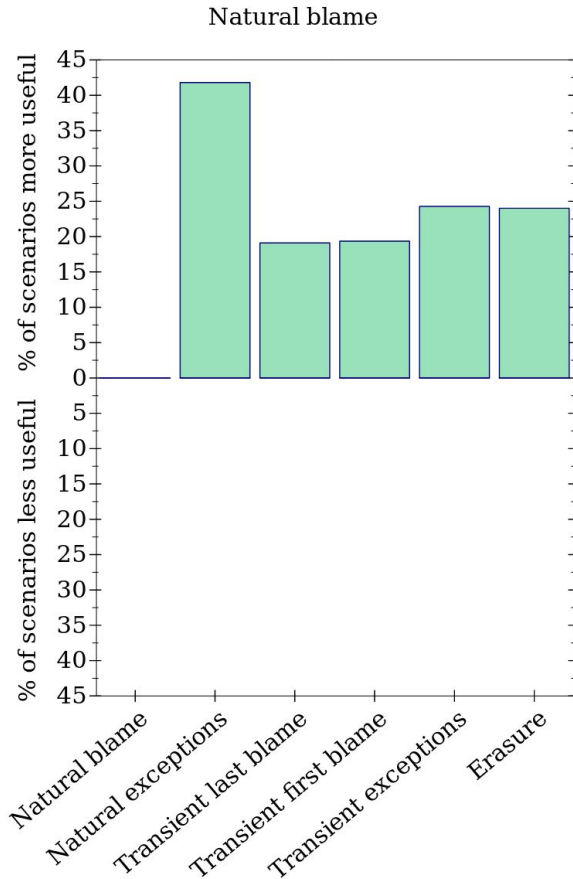
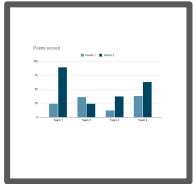


Mutation
[Lipton 1971, DeMillo et al. 1978]

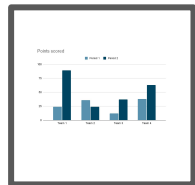
Results



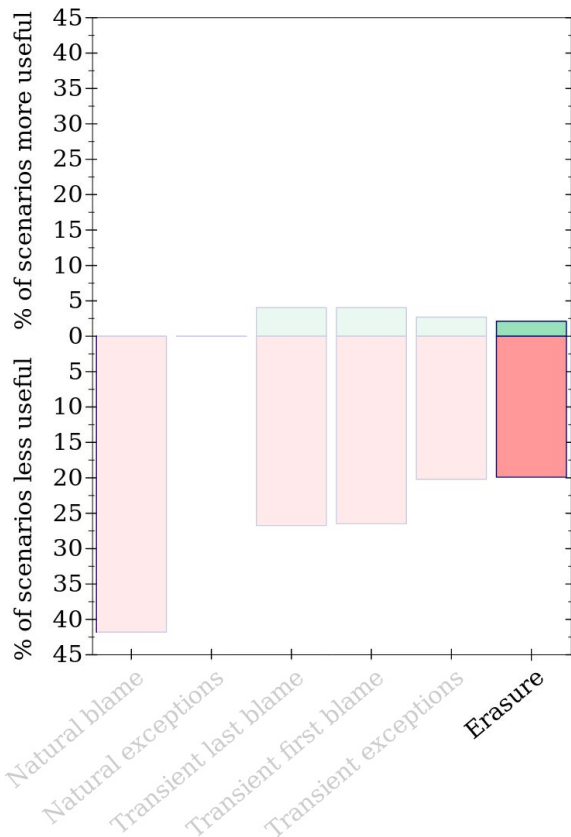
How Blame Stacks Up



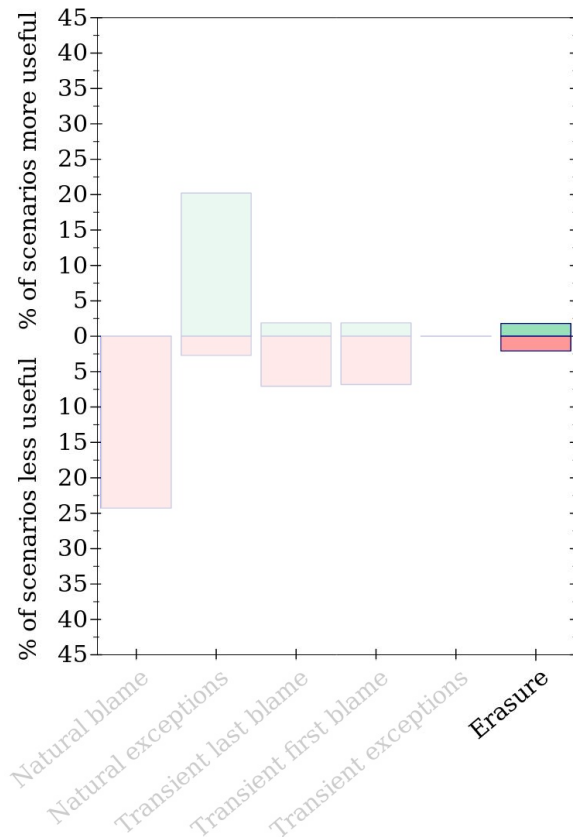
Checks *Without Blame* Don't



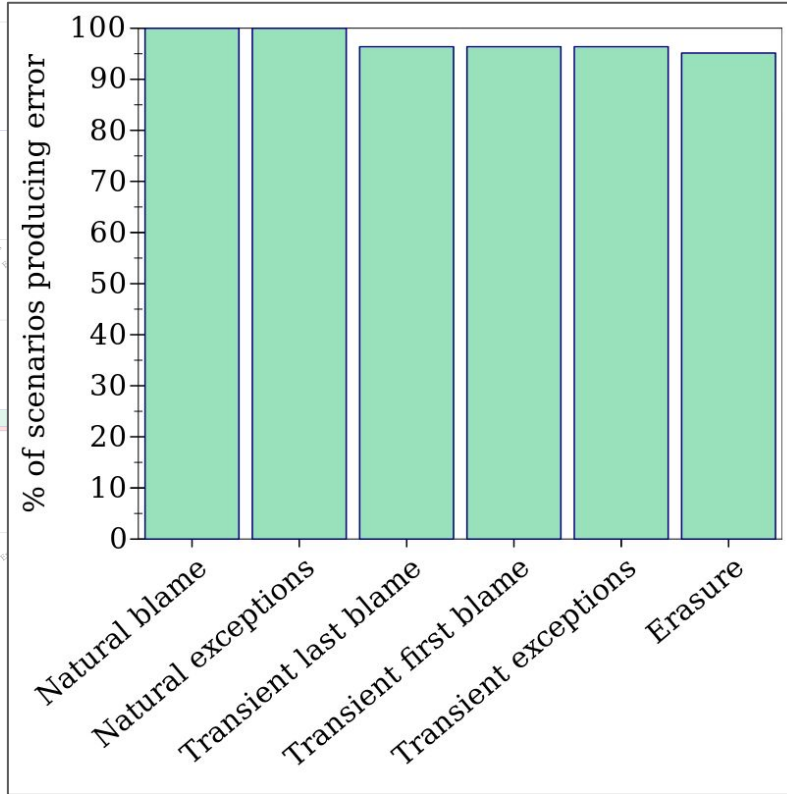
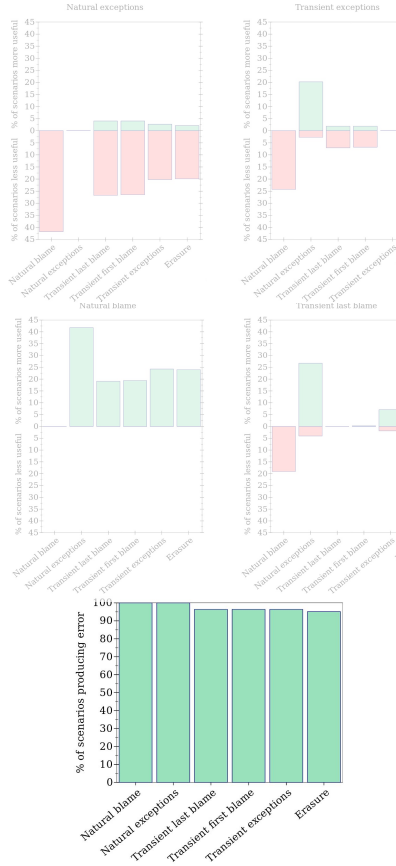
Natural exceptions



Transient exceptions



Takeaways



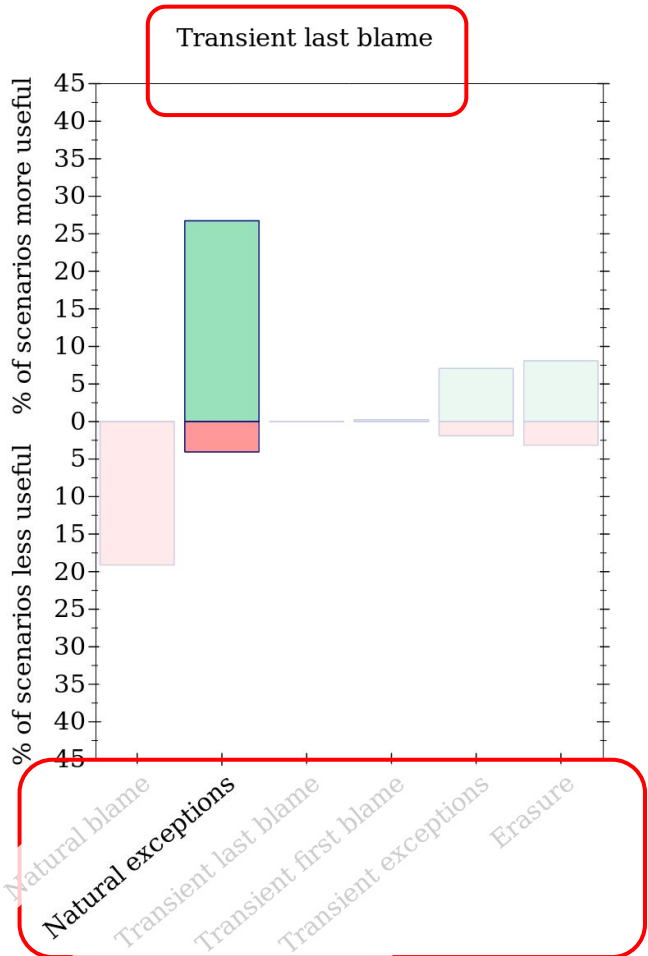
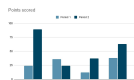
make sense?

the location of bugs.

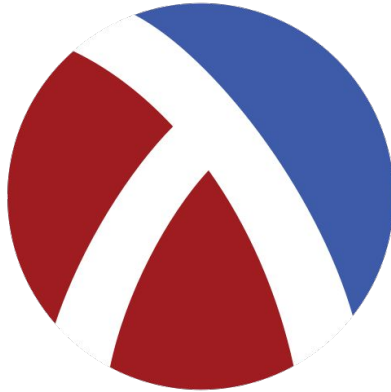
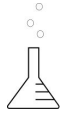
bugging mode be a useful



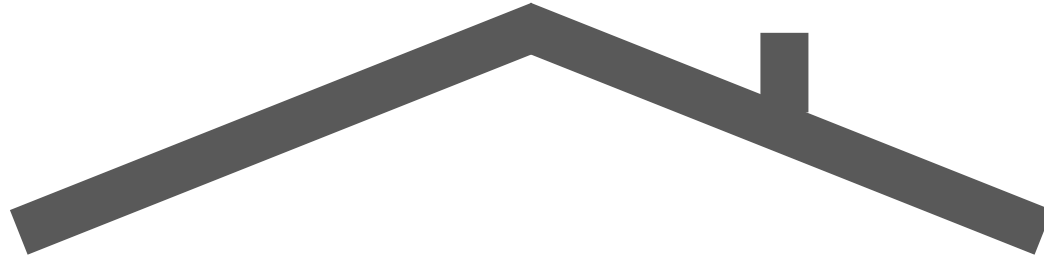
Results



Creating an Experiment to Test Our Hypothesis



[Greenman 2022]



Erasure

Transient

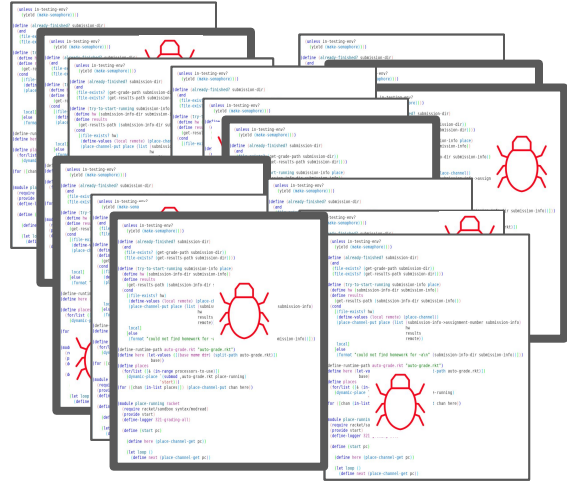
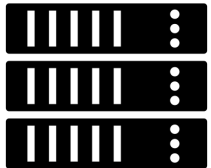
Natural



Creating an Experiment to Test Our Hypothesis



30,000 scenarios



~300 mutants

2 million scenarios
(mutant X configuration)



Erasure Detects Most of the Bugs

