Facilitating Program Analysis through Transformation

Zheng Gao

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

May 27, 2022

I, Zheng Gao, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Some of the work presented in this thesis has previously been published or submitted during my PhD study at University College London.

[1] Gao, Zheng, Christian Bird, and Earl T. Barr. ”To type or not to type: Quantifying detectable bugs in JavaScript.” 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 2017.

[2] Gao, Zheng, Alexandru Marginean, Mark Harman, and Earl T. Barr. ”Retype: Changing Types to Change Behaviour.” Submitted to OOPSLA 2016.

[3] Gao, Zheng, Martin Monperrus, Mark Marron, and Earl T. Barr. ”ValueScope: Tracking Values to Their Origin.” Submitted to FSE 2020.

In addition, the following work has been published during my PhD programme. These papers are relevant to my research. However they do not form part of the thesis content.

[1] Gao, Zheng. ”Numerical program analysis and testing.” Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014.

[2] Allamanis, Miltiadis, Earl T. Barr, Soline Ducousso, and Zheng Gao. ”Typilus: Neural type hints.” Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. 2020.

Abstract

Computer programs are reaching every corner of human lives, becoming ever more important. This, in turn, raises the cost of program malfunctions. Preventing or diagnosing these malfunctions requires extensive program analysis, but not every program is amenable to a specific analysis technique. For example, symbolic execution does not scale on programs involving floating point numbers or strings, due to the limitations of contemporary SMT solvers. This thesis attempts to advance the state of the art by utilising program transformation to facilitate analysis. To this end, it first establishes that program transformation is indeed invaluable when used to enable analysis. Specifically, it, departing from convention, views static typing as a form of program transformation and empirically quantifies the early detection of bugs static typing offers at the cost of type annotations. Then, it proposes TYPERITE, a tool that serves as an enabler to program analysis. TYPERITE allows developers to customise types via program transformation to detect domain-specific errors or permit analysis. Finally, this thesis presents VALUESCOPE, a dynamic program analysis that tracks a value of interest observed in an execution, like a `NaN` or a `null`, through its propagation chain back to its origin.

Impact Statement

The knowledge and methods of program analysis presented in this thesis are beneficial to the field of software quality improvement and could have impact on both software engineering research and industry.

In academia, our publication that quantifies static typing's benefits in early bug detection has received 83 citations at the time of submitting this thesis according to Google Scholar. Our tool, TYPERITE, can benefit researchers in computer science, where tailored, specific retype operation is common.

In industry, our empirical study on TypeScript has incorporated invaluable feedback from the language's development team in Microsoft. This work has assured the team that developing static typing for a dynamically typed language is worth every penny and they offered an opportunity on further collaboration. The paper also triggers heated online discussion on popular developer websites, including Reddit and Hacker News. Our tool, VALUESCOPE, is designed to ease debugging by tracking an erroneous value to its origins. It can help developers better understand the program and faster localise the bugs.

Acknowledgements

I would like to express the deepest gratitude to my principal supervisor, Prof. Earl T. Barr, who has not only guided me through this arduous doctoral journey, but also offered warm support when I was struggling emotionally.

I would like to thank my subsidiary supervisors, Prof. Mark Harman and Prof. Federica Sarro, for their valuable feedback on my research.

I would like to thank my family. Without their support, none of my achievement would be possible.

I thank my colleagues in the Software Systems Engineering (SSE) research group and the Centre for Research on Evolution, Search and Testing (CREST). The SSE reading groups and the CREST Open Workshops have exposed me to state of the art research topics and inspired my research.

I thank the Computer Science department of University College London for their financial support granted through my doctoral studentship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Born as calculating devices, computers are rapidly evolving and expanding their reach. Computerisation has been an unstoppable trend in recent decades and is likely to remain so in the foreseeable future. In academia, physicists and chemists use computer programs to simulate experiments; mathematicians perform numerical analysis in software; geneticists employ computers to study genes. In industry, more and more traditional, mostly manual processes are becoming computer-aided, from design to production, from quality assurance to maintenance. Even artists are using software, for example, to compose music or produce digital paintings.

The prevalence of computer programs raises the cost of a program malfunction. A study in 2002 commissioned by the Department of Commerce's National Institute of Standards and Technology revealed that software bugs cost the US economy \$59.5 billion annually [1]. More recently in 2017, a crash in British Airways' IT system caused the airline to cancel more than 700 flights and left tens of thousands passengers stranded, including the author of this thesis. The system meltdown was estimated to cost British Airways around £80 million [2]. In fact, a software failure may cause more than just economic damage. On February 25, 1991, a fixed-point truncation error prevented the Patriot missile system from intercepting an Iraqi Scud that killed 28 American soldiers and wounded approximately 100 others [3].

As an ancient Chinese proverb goes, "The water that bears the boat is the same that swallows it." While enjoying the benefits that computers bring, we have to carefully manage them so that we are not "swallowed". *Program analysis* aims to tackle precisely this problem. It is “the process of automatically analysing the behaviour of computer programs regarding a property such as correctness, robustness, safety and liveness” [4]. Program analysis not only can detect and eliminate functional defects, but can optimise non-functional properties. To maximise its utility, program analysis is often employed before software deployment. According to the Systems Sciences Institute at IBM [5], fixing an error identified in maintenance costs up to 100 times more than fixing one found during design.

Numerous program analysis techniques exist, such as type system, data-flow analysis, symbolic execution, and testing. However, many effective analysis techniques have limitations and target only a subset of all possible programs. For instance, techniques based on SMT solvers, including symbolic execution and model checking, struggle to scale when applied to programs that compute with floating point values, due to the immature solver support for floating point arithmetic.

Another notable example is static typing. Employed in almost all mainstream programming languages, type systems are one of the most successful program analysis techniques. Broadly, static and dynamic typing are the two distinct realisations of a type system. The former catches type errors early and does not impact the program’s runtime performance, at the cost of requiring developers to enforce a rigid type schema beforehand. The latter permits a more flexible and concise language syntax, obviating developers’ need to add type annotations to code and reason with types at compile-time. Therefore, programs composed in a language with dynamic typing are beyond the reach of static typing by definition; type errors hidden in those programs cannot be detected until runtime and are more likely to survive into deployment and maintenance.

1.1 Problem Statements

Countless researchers strive to invent program analysis techniques, either outperforming the state of the art for an existing problem or addressing a newly identified one. In contrast, how to transform a program under investigation so that it is amenable to existing program analysis is a field yet to be extensively explored. This thesis aims to fill the gap; the specific research problems that it tackles are discussed below.

Languages that implement dynamic typing, or dynamically typed languages, such as JavaScript, Python, and Ruby, are rocketing in various popularity rankings. Recent years, however, have witnessed an increasing uptake of optional type annotations in dynamically typed languages. For JavaScript, Microsoft devises TypeScript, a typed "superset" of JavaScript, and Facebook develops Flow, an optional type checker for JavaScript. Python incorporates optional type annotations in version 3.5. Industrial titans have followed suit: Facebook has released Pyre, Google Pytype, and Microsoft Pyright. The incentives of such huge investments remain unclear. This thesis asks the following questions:

What are the real-world benefits that optional type annotations have?

Types are invented to avoid the Russell's paradox arising in the naïve set theory [6]. They are now a fundamental concept in programming, preventing bugs and facilitating code comprehension, navigation, and completion. Some types, however, are more difficult to reason with, because of their vast value domains and the complex semantics of their supported operations. Floating point is among them, due to its unintuitive arithmetic (*e.g.*, the Associative and Distributive laws do not hold). For example, researchers in abstract interpretation have to devise novel, cumbersome abstract domains [7, 8, 9] to cope with floating point arithmetic. Strings are worse still, because checking a nontrivial property, *e.g.*, inclusion and equivalence, are undecidable over string operations [10]. This thesis raises the following question:

Can we design a transformation that replaces a type with another in a program to enable some existing analyses on the transformed program?

Programs compute with values. Most values reside in variables and may propagate from one variable to another. Variable to value bindings are not directly visible to developers. When a value of interest manifests itself in a variable, where the value was originated and how it propagated to this variable at the manifestation point are often opaque. In the context of debugging, tracking an erroneous value to its origin is a fundamental task, because it helps localise the bug's root cause. The most naïve approach to this task involves repeatedly inserting `print` statements and rerunning the program, which is tedious and inefficient. This thesis raises the following question:

> *Can we develop an analysis technique that tracks a value of interest through its propagation chain back to its origin?*

1.2 Goal and Objectives

The goal of this thesis is to advance the state of the art in program analysis by utilising transformation to make programs more amenable to analysis techniques. To achieve this goal, the following objectives are set.

1. To quantify the benefits of optional type annotations in bug detection. Though focusing on optional typing, a specific program analysis technique, this objective aims to address the more general research question: how beneficial can program transformation be after facilitating program analysis?

2. To formulate a general program transformation that replaces a type with another in a program so that some existing analyses are applicable to the transformed program. This objective instantiates the general idea of facilitating program analysis through transformation.

3. To devise a transformation that dynamically tracks a value of interest to its origins. This objective requires exploration in both fields: designing a novel program analysis and implementing the analysis via program transformation.

1.3 Contributions

The main contributions of this thesis are:

- This thesis quantifies the public bugs that static type systems detect and could have prevented: 15% for both Flow 0.30 and TypeScript 2.0, on average.

- This thesis introduces and formalises retyping a program. This formalism

1. injects the type-conversions that arise due to interactions with code that is not retyped;
2. allows the controlled introduction new behaviour through the new, target type; and
3. applies to both non-object-oriented languages and primitive types because it uses term rewriting, not the subtype relation.

- This thesis presents a realisation of RETYPE in the tool TYPERITE and validates it:

  1. demonstrating how it can be used to catch real world bugs, like an SQL injection in `hibernate`, and automates taint analysis on implicit information flow;
  2. showing that TYPERITE scales to Pidgin, an open-source online chatting client with 363K lines of code.
  3. automating retyping, eliminating the need for a developer to assess whether 120,096 operators need replacement or conversion when retyping float to fixed point in the special functions of the GNU scientific library.

- This thesis defines parameter relevance: A principled means to track of value of interest through computations that produce it by restarting on those values most relevant to the initial value of interest.

- This thesis presents presents VALUESCOPE for JavaScript, which automates a common debugging task: finding the origin and propagation of problematic values, like a `NaN`. The key to VALUESCOPE's performance is its capacity to restart on new values of interest, computed using parameter relevance.

- This thesis experimentally demonstrates that VALUESCOPE outperforms a state-of-the-art spectrum based fault localisation technique that uses the DStar [11] ranking measure and Orbs [12], a dynamic program slicer, in localising real world bugs.

1.4 Thesis Organisation

This thesis covers several interconnected topics surrounding facilitating program analysis through program transformation. The remainder of the thesis is organised as follows.

Chapter 2 reviews literature on program analysis, especially type system, symbolic execution and slicing, and summarises the research opportunities identified from the review.

Chapter 3 presents an empirical study that quantifies the benefits of static typing. Specifically, it investigates how many bugs in real-world JavaScript projects could have been caught, had two static type systems, Flow and TypeScript, been used during development.

Chapter 4 proposes TYPERITE, a general tool that changes the input program's semantics by changing its type schema, thereby facilitating program analysis or optimising the program's non-functional properties.

Chapter 5 describes VALUESCOPE, a debugging tool that employs program transformation to locate the origin of a value observed at a particular state. VALUESCOPE is conceptually an extreme development of type system, whose types are singleton sets.

Chapter 6 proposes future work and concludes the thesis.

Chapter 2

Literature Review

Program transformation is any operation that takes a computer program and generates another. Fundamentally, program transformation is an instance of term rewriting [13], which formalises replacing the subterms of a formula with other terms. A formal discussion of term rewriting, however, is beyond the scope of this thesis.

In many cases, the transformed program is required to be semantically equivalent to the original one. Notable examples of semantics preserving program transformation include compilation and refactoring. In other cases, program transformation changes the semantics of the original program in a predictable way [14]. The program transformation this thesis discusses is not restricted to be semantics-preserving.

While program transformation can be manual, it is often more practical to devise a system to automatically apply the required transformations. Indeed, compiler developers have been paying more attention to a modular design, where different components of a compiler, especially the front-end, can be cleanly exposed to the users, thereby providing various services, such as source-to-source transformation.

Program transformation has many applications. In particular, this thesis uses it to make the transformed programs amenable to program analysis. Broadly, two kinds of program analysis exist: static and dynamic. Nielson *et al.* [15] define static program analysis as "static compile-time techniques for predicting safe and computable approximation to the set of values or behaviours arising dynamically at run-time when executing a program on a computer." Static analysis operates at compile time, allowing developers to fix issues early in the development lifecycle. It also attempts to reason about many, if not all, executions of a program at once. For example, a program is guaranteed to be type correct with all inputs, if it passes a sound static checker; Section 2.1 elaborates this point. In practice, however, static analysis suffers from two problems. First, statically analysing an interesting property, such as correctness, of programs written in a reasonably complex language is hard and often undecidable. May alias analysis, the analysis of statically deciding whether two pointers point to the same storage location in *any* execution, is a notable example. By a reduction to the halting problem [16], Landi proves that "for languages with if-statements, loops, dynamic storage, and recursive data structure", may alias analysis, the fundamental building block of many static analysis techniques, is undecidable. To combat undecidability, static analysis often has to first build and then analyse some approximation to the input program's actual semantics. Approximation necessarily leads to false positives/negatives, which hampers the utility of static analysis. Second, many static program analysis techniques, such as static symbolic execution and model checking, struggle to scale to programs at industry scale.

Dynamic program analysis, on the other hand, executes programs on a real or virtual processor and analyse the observed runtime information. Testing is the most ubiquitous dynamic program analysis technique due to its efficiency and ease of use. The biggest drawback of dynamic program analysis is that it in general underapproximates an input program's semantics, because it covers only a limited number of executions. Dijkstra's famous observation that "testing can be used to show the presence of bugs, but never to show their absence" [17] generally applies to all dynamic program analysis techniques. Thus, for dynamic program analysis to be effective, the target program needs to be executed with sufficient and representative inputs to cover diverse execution paths.

Numerous program analysis techniques exist. This chapter reviews a selection of them, aiming to set the background of and motivates the research presented in this thesis. It starts with type system, a technique that resides in almost every language's compiler/runtime. Both Chapter 3 and Chapter 4 centre around type system: Chapter 3 empirically quantifies the benefit of static type system on code quality; Chapter 4 leverages type system to apply controlled program transformation. This chapter then studies symbolic execution, which motivates Chapter 4 to extend the application domain of symbolic execution. This chapter concludes with briefly reviewing four dynamic program analysis techniques that relate to Chapter 5.

2.1 Type System

Despite its simplicity and elegance, naïve set theory [18] has limitations. The Russell's Paradox [19] exemplified this fact by illustrating that some formalisations in naïve set theory could lead to a contradiction. To escape this paradox that had shaken the foundations of mathematics, Russell proposed ramified theory of types in 1908 [20]. At the core of this theory is a hierarchy of types; each mathematical entity would be assigned a type to prevent circularity in reasoning. Enlightened by Russell's work, many researchers have realised the significance of types and contributed to this field of study. Major landmarks include Ramsey's simple theory of types [21], Church's simply typed λ-calculus [22], Martin-Löf's constructive type theory [23, 24], and Berardi, Terlouw, and Barendregt's pure type systems [25, 26, 27]. Collectively, these researchers have established and nurtured *type theory*, which now refers to a class of formal systems that centred around types. Informally, types are a property of terms; they specify the available operations on the terms and therefore constrain the behaviours of the terms. For example, Frege's early work [28] distinguished between objects, predicates, predicates of predicates, *etc.*. Therefore, a first-order predicate applies to an object, but it cannot have a predicate as argument.

Because of its generality, type theory has applications in many disciplines, one of which is *type system* in computer science. Ever since its inception, type system has been extensively studied and become one of the most widely used lightweight formal methods. As with many terms shared by large communities, it is difficult to define "type system" in a way that is concise enough but also covers all its practical usage by programming language designers. Pierce made the following attempt [29]:

> A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the

kinds of values they compute.

Two points deserve comments. First, type system is a verification technique aiming to prevent some program behaviours which are often called *type errors*. Type error, however, is a concept dependent on the underlying type system and lacks a universal definition. To reflect the extent to which a programming language's type system forbids type errors, the term *type safety* is coined. Pierce summarised that a type safe language must "protect its own abstractions" [29]. Second, type system classifies terms according to the properties of the values that they will compute when executed. A type system can be regarded as an approximation to the run-time behaviours of the terms in a program. Though details vary, a type system typically consists of two stages, 1) *type assignment* that associate every construct in a program with a type, and 2) *type checking* that validates whether these constructs are used as permitted by their types.

Three pairs of concepts that often cause confusion need further explanation, namely static and dynamic typing, type annotation and type inference, and strong and weak typing.

Static and Dynamic Typing A type system can perform type checking statically before execution (*i.e.* static typing) or defer it to runtime (*i.e.* dynamic typing). Programming languages that employ static or dynamic typing are called statically or dynamically typed, respectively. Static typing analyses solely source code and offers many unique benefits. For example, it detects type errors early in the development cycle, thereby reducing bug costs; it improves program documentation and navigation by providing typed interfaces, thereby increasing productivity. On the other hand, static typing is rigid in enforcing type safety. Specifically, it desires developers to decide the types of all terms in their programs beforehand; once decided, the type of a term can no longer change. This design overhead, as opponents argue, slows down the development for daily programming tasks. Static type systems are also conservative: they can prove the absence of some bad program behaviours, but they cannot prove their presence. Hence, static type systems may reject programs that actually behave well at runtime. Formally, for a Turing-complete programming language, in order to maintain soundness (*i.e.* failing all programs with type errors) and therefore utility, static typing is inevitably incomplete (*i.e.* rejecting programs without any type errors). For example, a program like

```
if <complex test> then <normal execution> else <type error>
```

will be rejected as ill-typed, even the `<complex test>` may always evaluate to `true`, because a static analysis cannot determine the predicate's result.

These reasons have led to the birth of dynamic typing that uses runtime type tags to distinguish objects and update the tags when needed based on the def-use chain of the objects. Consequently, in a typical dynamically typed language, developers are free to change the type of a term. For instance, the code snippet below is rejected in Java, but not in JavaScript, regardless of the original type of variable `foo`:

```
foo = 0;
foo = "hello";
```

This flexibility exempts developers from making explicit upfront commitment to constraining the values an expression can consume or produce, which facilitates the writing of reflective, adaptive code. More importantly, dynamic typing is able to type check certain language features that are beyond the reach of static typing. Downcasting is a notable example, because whether the object of the base class holds a value of the derived class can be decided only at runtime.

Static Typing for Dynamically Typed Languges Dynamic typing gained much momentum before circa 2010, illustrated by the surging industrial uptake of dynamically typed languages back then, such as JavaScript, Python, and Ruby. In recent years, however, this expansion of dynamic typing seems to have hit a bottleneck, as dynamically typed languages set foot into the field of large scale, complex programming. Most newly-designed, popular industrial languages, such as Go and Rust, adopt static typing. Even within the dynamic typing community, static typing is rising. For example, starting from version 3.5, Python started to gradually introduce language features that allow developers to optionally annotate the types of local variables, function parameters and returns (PEP 484[1] and PEP 526[2]). Though Python

[1] `https://peps.python.org/pep-0484/`
[2] `https://peps.python.org/pep-0526`

remains dynamically typed, third-party type checkers, such as mypy[3], pytype[4] and pyre[5], collect these type annotations, statically type check them, and provide useful error messages. JavaScript is another notable example. Ever since Thiemann's first proposal [30], optional static typing for JavaScript has rapidly developed. Standalone type checkers such as Flow, compilers supporting optional type annotation and type inference such as Closure, and programming languages extending JavaScript with optional static typing such as TypeScript and Dart, have emerged.

What is the intuition behind this turn of the tide? Chapter 3 answers precisely this problem. It presents an empirical study that quantifies the benefit of optional typing on code quality.

Empirical Studies on Static and Dynamic Typing Researchers have empirically compared dynamic and static type systems [31, 32, 33, 34, 35, 36, 37]. These studies perform the comparison along different dimensions, including development productivity, code usability, and code quality. Prechelt and Tichy conducted a controlled experiment, in which 34 subjects were divided into four groups, two developing in C with a type system and other two without, to assess whether the type system would benefit developers [38]. Hanenberg's study [39] which ran for over a year and involved 49 students is a notable achievement. Hanenberg wrote his own language in two versions, one equipped with a static type system and the other with a dynamic one. The subjects were then divided into two groups, and required to write a simplified Java parser. For each student, Hanenberg recorded the development time of the scanner and the number of passed test cases for the parser. The results indicated that static type systems did not have a significant positive impact on both the development time and the code quality.

Nonetheless, questions remain. Developers are well-paid so it is expensive to study them. Thus, most studies are small scale or use students; while Hanenberg's scale is impressive, it still relies on students. Further, his custom language is itself another confounding factor. In contrast, our experiment samples from what developers

[3]`https://github.com/python/mypy`
[4]`https://github.com/google/pytype`
[5]`https://github.com/facebook/pyre-check`

naturally do in the course of their work, in particular from approximately 537,709 commits they create. Thus, it avoids the cost of dedicated developer participation. Moreover, the semi-automation and modular implementation of this experiment offers great reproducibility and adaptability.

Type Annotation and Type Inference Original static type systems collect type information from type annotations. Specifically, these type systems require developers to explicitly annotate their code and declare the types of self-defined objects, such as local variables and functions. In order to alleviate the annotation burden on developers, researchers have dedicated to devising a type inference algorithm that can automatically deduce the types of different constructs based on only program texts. Intuitively, such an algorithm starts with an implicit understanding of the types of various atomic values (*e.g.* `true` is a boolean value and `13` is an integer) and infers the type of an expression by aggregating the types of its sub-expressions. On the following code snippet, a typical type inference algorithm will first decide that variables `a` and `b` are of types `int` through knowing that values `1` and `2` are of types `int`, then assign function `foo` a type `int -> int -> int`, and finally deduce that variable `c` is of type `int`.

```
function foo (x, y) {
  return x + y;
}
a = 1;
b = 2;
c = foo(a, b);
```

The most well-known type inference algorithm is Algorithm W. The origin of this algorithm is the type inference algorithm for Curry and Feys' simply typed λ-calculus [40]. In 1969, Hindley extended this work and proved that their algorithm always inferred the most general type [41]. In 1978, Milner, independently of Hindley's work, provided an equivalent algorithm, Algorithm W [42]. In 1982, Damas and Milner finally proved that Algorithm W is complete and extended it to support systems with polymorphic references [43].

Complete type inference, however, is generally undecidable for an expressive, mainstream programming language. For example, type inference with polymorphic recursion is provably undecidable [44]. In these cases, explicit type annotations are required to help the underlying type systems resolve ambiguity.

Strong and Weak Typing The definitions of “strong” and “weak” typing are more contentious than that of type system. Their frequent appearances in discussions or even in literature, however, call for clarification. These terms originate from the work published in 1970s. According to Liskov and Zilles, a strongly-typed language must satisfy the requirement that “whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function” [45]. Jackson defined a strongly typed language to be one in which “each data area will have a distinct type and each process will state its communication requirements in terms of these types” [46]. Nowadays, “strong” and “weak” actually describe the extent to which a type system enforces type safety and are more appropriate in a comparative setting. Additionally, some may confuse strong typing with static typing. A statically typed language can be more weakly typed than their counterparts, *e.g.* C; a dynamically typed language can be strongly typed, *e.g.* Python.

Programming languages with certain features are often deemed less strongly typed. For example, type coercions are commonly assumed to be error-prone, so JavaScript is notorious for its permissiveness of type coercions [30, 47, 48, 49]. Pointers are another example. These beliefs, however, can be misleading. Pradel and Sen empirically studied the usage of type coercions in a large corpus of real-world JavaScript programs and revealed that most coercions (98.85%) were likely to be harmless [50]. Due to these controversies, this thesis recommends against the vague use of “strongly” or “weakly typed” to describe a programming language; instead, one should be more precise, for example, discussing the presence or absence of certain properties, like type safety or memory safety.

2.1.1 Specific Type Systems

Just relying on informal discussion is insufficient to truly understand specific type systems. To accurately account for them, this thesis must first pick a proper formalism. Though multiple formal models for type systems exist, such as term rewriting systems [51] and term graph rewriting systems [52], typed λ-calculus [27] remains the most widely used and therefore is employed throughout this section.

λ-Calculus As its name suggests, Typed λ-calculus is built upon λ-calculus, which this thesis briefly reviews here. Lambda-calculus is a formal system that was developed by Church [53, 54] and investigated by many his contemporaries, such as Turing [55], Kleene, and Rosser [56]. It models computation via function abstraction and application and supports variable binding and substitution. λ-calculus is proved to be Turing complete that it is able to simulate any Turing machine. Formally, in λ-calculus, a λ expression [6] is defined as follows:

$$\begin{array}{lrl} e ::= & x \;| & \text{variable} \\ & \lambda x.e \;| & \text{abstraction} \\ & e\;e & \text{application} \end{array}$$

A variable x is itself a λ expression; the abstraction of a variable x from a λ expression e is an expression; the application of a λ expression to another one is also an expression. A variable x that occurs within the scope of an abstraction $\lambda x.e$ is *bound*; all other variables are *free*. A variable is bound by the syntactically innermost enclosing λ, as in any block-structured programming language. An expression is *closed* if it contains no free variables, otherwise it is *open*.

λ-calculus allows three kinds of reduction on λ expressions:

$$\begin{array}{rcrr} \lambda x.e[x] & \rightarrow_\alpha & \lambda y.e[y] & \alpha\text{-conversion} \\ (\lambda x.e_1)\; e_2 & \rightarrow_\beta & e_1[e_2/x] & \beta\text{-reduction} \\ (\lambda x.(e\,x)) & \rightarrow_\eta & e & \eta\text{-conversion} \end{array}$$

[6]While some existing literature treats λ term and λ expression as synonyms [57, 58], some online article argues that a λ expression can be syntactically invalid and a λ term is a valid λ expression [59]. For simplicity, this thesis does not distinguish between them.

where $e_1[e_2/x]$ means replacing every occurrence of x in e_1 with e_2. Alpha-conversion, also known as α-renaming [60], allows the name of a bound variable to be changed. Beta-reduction substitutes formal with actual parameters. Eta-conversion can be applied only when e does not contain any free x and removes redundant abstractions. The term *redex*, refers to any sub-expression reducible by one of the reduction rules. An expression is in *normal form* if it contains no redexes.

Typing Rules Having revisited λ-calculus, this thesis now formally discuss type systems. A type system is essentially a set of typing rules built upon a core syntax. The typing rules are formulated with respect to an environment for the fragment being type checked. They have a set of judgements as premises and another set as conclusions. In general, a judgement has the form:

$$\Gamma \vdash \psi$$

where Γ denotes a typing environment that encodes the assignment of types to variables and ψ denotes an assertion whose free variables are declared in Γ. More formally, Γ is a set of pairs $x : \tau$, where x is a variable, τ its type, and the colon operator the has-type relation. When Γ is empty, it is denoted as $\emptyset$. This judgement asserts that given Γ, ψ holds. A more specific judgement that most typing rules involve, called the *typing judgement*, is typically of the form

$$\Gamma \vdash e : \tau$$

where e is a λ expression and τ a type. The typing judgement asserts that e has a type τ with respect to a typing environment Γ for the free variables of e. Other specific forms of judgements are also necessary. For example,

$$\Gamma \vdash \diamond$$

states that the typing environment Γ is *well-formed*, which intuitively means that Γ is constructed properly. An empty typing environment is well-formed. Given a type system, one may apply its typing rules to establish a derivation that is a tree of judgements with leaves at the top and a root at the bottom, where each judgement is obtained from the ones immediately above it by some rule of the system. A *valid* judgement is one that can be obtained as the root of a derivation. To verify the type correctness of an expression, one can check the existence of a derivation for the expression. If such a derivation does not exist, the expression is not typeable or *ill-typed.* In other words, it has a type error. Otherwise, it is *well-typed.* The types in a type system can be either *basic types* that are built-in, or new types that are constructed using a *type constructor* on old types. Basic types can be considered as ones built using nullary type constructors.

The remainder of this section reviews a list of major, specific type systems. It starts with the simply typed λ-calculus that is the canonical and simplest instance of a typed lambda calculus, and then investigates three extensions of the simply typed λ-calculus. The goal of this literature review is to present the direction to which contemporary type systems are heading, thereby identifying research opportunities. This section concludes with an informal, brief discussion of other advanced typing techniques.

Simply Typed λ-Calculus In 1940, Church introduced a typed interpretation of the λ-calculus, known as the simply typed λ-calculus or $\lambda_{\to}$ [22]. Figure 2.1 details its syntax. $\lambda_{\to}$ extends the syntax of untyped λ-calculus with a set of basic types G, type annotations for bound variables in $\lambda x : \tau.e$, and a single type constructor $\to$ to build function types. $\lambda_{\to}$'s syntax also includes a set of constants C, which, however, is not a necessity, because constants can be encoded using pure λ terms. For example, the Church encoding [61] represents *true* in:

$$\lambda t.\,\lambda f.\,t$$

This explicit inclusion of constants is mainly for notational convenience. More importantly, an expression that is valid with respect to the syntax of untyped λ-calculus is also actually well-behaved, which is not the case for $\lambda_{\to}$. For example, $\lambda x : \gamma.(x\,y)$ is not well-behaved, because x is of a basic type γ not a function type and

Variables	$x \in \mathbb{X}$		
Basic Types	$\gamma \in \mathbb{G}$		
Constants	$c \in \mathbb{C}$		
Types	τ	$::=$	$\gamma \mid \tau \to \tau$
Expressions	e	$::=$	$c \mid x \mid \lambda x : \tau . e \mid e\ e$

Figure 2.1: The formal syntax of the simply typed λ-calculus.

Judgement	**Interpretation**
$\Gamma \vdash \diamond$	Typing environment Γ is well-formed.
$\Gamma \vdash \tau$	Type τ is well-formed in Γ.
$\Gamma \vdash e : \tau$	Typing environment Γ entails that expression e is of type τ.

Table 2.1: The three judgements needed to compose typing rules for $\lambda_{\to}$.

cannot be applied to y. Therefore, the verification of a given expression's semantic correctness is left to type checking.

The simply typed λ-calculus's typing rules require only three simple judgements, shown in Table 2.1. The judgement $\Gamma \vdash \tau$ is actually redundant, since all syntactically correct types are well formed in any environment Γ by construction. This judgement, however, is necessary for more powerful yet complicated type systems, which will be discussed later.

Using the presented judgements, the simply typed λ-calculus's typing rules are presented in Figure 2.2. The rule (Env $\emptyset$) formalises the discussion above that an empty typing environment is well-formed. The rule (Env x) expands an environment Γ into a larger one $\Gamma, x : \tau$, where the comma operator extends behaves like set union, provided that τ is valid type in Γ. The rules (Type Basic) and (Type $\to$) construct types. While the former automatically include basic types into Γ, the latter builds new types using the $\to$ type constructor. The rule (Var) states that a variable x has whatever type we assume it to have. The rule (Abs) assigns type $\tau_1 \to \tau_2$ to a function abstraction, if the function body receives type τ_2 under the assumption that the formal parameter has type τ_1. The rule (App) associates a function application $e_1\ e_2$ with a type τ_2, if the function e_1 is of type $\tau_1 \to \tau_2$ and the actual parameter

$$\frac{}{\emptyset \vdash \diamond}\ \text{Env } \emptyset \qquad \frac{\Gamma \vdash \tau \quad x \notin dom(\Gamma)}{\Gamma, x : \tau \vdash \diamond}\ \text{Env } x$$

$$\frac{\Gamma \vdash \diamond \quad \tau \in Basic}{\Gamma \vdash \tau}\ \text{Type Basic} \qquad \frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \rightarrow \tau_2}\ \text{Type} \rightarrow$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{Var} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau . e \, : \, \tau_1 \rightarrow \tau_2}\ \text{Abs}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \, e_2 : \tau_2}\ \text{App}$$

Figure 2.2: The typing rules of the simply typed λ-calculus.

has type τ_1. Collectively, these typing rules form the theoretical basis of $\lambda_{\rightarrow}$.

Despite being a syntactical extension of the untyped λ-calculus, $\lambda_{\rightarrow}$ is in fact computationally weaker. In 1967, Tait demonstrated that β reduction of $\lambda_{\rightarrow}$ is strongly normalising [62]. In 1991, Berger and Schwichtenberg presented a purely semantic normalisation proof [63]. These important results show that $\lambda_{\rightarrow}$ is strongly normalizing, so every well-typed expression in $\lambda_{\rightarrow}$ is reducible to a normal form; that is, every program halts. Therefore, $\lambda_{\rightarrow}$ cannot simulate a Turing machine that may never halt and is not Turing-complete.

Hindley Milner Type System Introduced independently by Hindley [41] and Milner [42], Hindley Milner type system [64] is a milestone in the development of type systems. The core novelty of Hindley Milner type system is two fold: First, it allows parametric polymorphism [65] by introducing the *let* type constructor, and second, it proposes Algorithm W to fast deduce the most general type of a term in a program without requiring type annotations from the developer. Algorithm W is a recursive procedure: Given an expression whose type is to be inferred and a type environment, it decomposes the expression into subexpressions and handles them differently based on their kind. For example, if the subexpression is a variable, Algorithm W simply looks up its type in the type environment. If the subexpression is a function abstraction, Algorithm W adds a type variable for the parameter to the type environment, recursively infers the type of the function body, and uses what is learned about the function body to instantiate the parameter's type. While in a worst case scenario Algorithm W is DEXPTIME-complete in complexity [66], it can often

Variables	$x \in \mathbb{X}$			
Types	τ	$::=$	$\mathrm{T} \mid$	Type Variable
			$\tau \to \tau \mid$	Function Type
			$\forall \mathrm{T}.\tau$	Universally Quantified Type
Expressions	e	$::=$	$x \mid$	Variable
			$\lambda x : \tau.e \mid$	Abstraction
			$e\ e \mid$	Application
			$\lambda \mathrm{T}.e \mid$	Polymorphic Abstraction
			$e\ \tau$	Type Instantiation

Figure 2.3: The formal syntax of system F.

achieve linear complexity in practice [67], if its scanning of the type environment is well-engineered.

System F The simply typed λ-calculus is a first order formal system, as it lacks type parameterisation and type abstraction. Girard [68] and Reynolds [69] independently created system F, which extends the simply typed λ-calculus by introducing *universal quantification* over types. Hence, system F is a second order typed λ-calculus that formalises parametric polymorphism and serves as the theoretical foundation for programming languages such as Haskell and ML.

Figure 2.3 details the syntax of system F. Formally, system F includes a new form of term, $\lambda \mathrm{T}.e$, indicates an expression e that is parameterised with a type variable T that ranges over arbitrary types. With this new term, one may now write generic functions. For example, in $\lambda_{\to}$, the identity function of a fixed type τ is $\lambda x : \tau.x$; in system F, one can define the identity function of an arbitrary type as $\lambda \mathrm{T}.\lambda x : \mathrm{T}.x$. Then they can instantiate the type parameter of this parametric function with any given type τ, *i.e.* $(\lambda \mathrm{T}.\lambda x : \mathrm{T}.x)\ \tau$, which produces back $\lambda x : \tau.x$. However, $\lambda_{\to}$ is not able to assign a proper type to this term. Hence, system F adds the universally quantified types. The term $\lambda \mathrm{T}.e$ is of type $\forall \mathrm{T}.\tau$, meaning that for all T, the body e has type τ (here e and τ may contain occurrences of T. For example, the type of the parametric identity function is $\forall \mathrm{T}.\mathrm{T} \to \mathrm{T}$.

Though the judgements that system F use are exactly the same as those in $\lambda_{\to}$

$$\frac{}{\emptyset \vdash \diamond}\ \text{Env } \emptyset \qquad \frac{\Gamma \vdash \tau \quad x \notin dom(\Gamma)}{\Gamma, x : \tau \vdash \diamond}\ \text{Env } x \qquad \frac{\Gamma \vdash \diamond \quad \mathrm{T} \notin dom(\Gamma)}{\Gamma, \mathrm{T} \vdash \diamond}\ \text{Env T}$$

$$\frac{\Gamma \vdash \diamond \quad \tau \in Basic}{\Gamma \vdash \tau}\ \text{Type Basic} \qquad \frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2}\ \text{Type} \to$$

$$\frac{\Gamma, \mathrm{T} \vdash \tau}{\Gamma \vdash \forall \mathrm{T}\tau}\ \text{Type } \forall \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{Var}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau.e\ :\ \tau_1 \to \tau_2}\ \text{Abs1} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{App1}$$

$$\frac{\Gamma, \mathrm{T} \vdash e : \tau}{\Gamma \vdash \lambda \mathrm{T}.e : \forall \mathrm{T}.\tau}\ \text{Abs2} \qquad \frac{\Gamma \vdash e : \forall \mathrm{T}.\tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash e\ \tau_2 : [\tau_2/\mathrm{T}]\tau_1}\ \text{App2}$$

Figure 2.4: The typing rules of system F.

(Table 2.1), its typing rules are more complex. Figure 2.4 illustrates the complete typing rules for system F. Compared against $\lambda_{\to}$ (Figure 2.2), the additional rules are: (Env T), which extends the typing environment with a type variable; (Type $\forall$), which constructs a quantified type $\forall \mathrm{T}.\tau$ from a type variable T and a type τ which may be parameterised with T; (Abs2), which abstracts parametric polymorphic; and (App2), which instantiates a polymorphic abstraction with a given type, where the substitution $[\tau_2/\mathrm{T}]\tau_1$ replaces all the free occurrences of T in τ_1 with τ_2. As a simple yet instructive example, let us build the derivation for $id(\forall \mathrm{T}.\mathrm{T} \to \mathrm{T})(id)$, where *id* has type $\forall \mathrm{T}.\mathrm{T} \to \mathrm{T}$. Applying the rule (App2) gives $id(\forall \mathrm{T}.\mathrm{T} \to \mathrm{T})(id) \equiv ((\forall \mathrm{T}.\mathrm{T} \to \mathrm{T}) \to (\forall \mathrm{T}.\mathrm{T} \to \mathrm{T}))(id)$; then applying the rule (App1) gives $((\forall \mathrm{T}.\mathrm{T} \to \mathrm{T}) \to (\forall \mathrm{T}.\mathrm{T} \to \mathrm{T}))(id) \equiv \forall \mathrm{T}.\mathrm{T} \to \mathrm{T}$.

First Order Type System with Subtyping Subtyping, a kind of polymorphism, intuitively captures the inclusion relation between types. An entity of a type can be deemed an entity of any of its supertypes. Thus, this entity should be safely used where an entity of its supertype is expected. Though not until 19080s had subtyping been formalised by Reynolds [70] and Cardelli [71], it is core to the object-oriented programming paradigm, so the underlying type systems must handle it. Broadly, two kinds of subtyping exist: *nominal subtyping*, in which a type is another's subtype only when the developers explicitly declare so, and *structural subtyping*, in which the structure of two types determines whether or not one is a subtype of the other.

Judgement	Interpretation
$\Gamma \vdash \diamond$	Typing environment Γ is well-formed.
$\Gamma \vdash \tau$	Type τ is well-formed in Γ.
$\Gamma \vdash e : \tau$	Typing environment Γ entails that expression e is of type τ.
$\Gamma \vdash \tau_1 <: \tau_2$	Type τ_1 is a subtype of τ_2 in Γ.

Table 2.2: The four judgements needed to compose typing rules for a type system that supports subtyping.

To model subtyping, $<:$, the subtyping relation operator must be added; $\tau_1 <: \tau_2$ asserts τ_1 is a subtype of τ_2. Consequently, a type system with subtyping utilises an additional judgement, called the *subtyping judgement*, shown on the last row in Table 2.2. One of the simplest type systems with subtyping is an incremental extension of $\lambda_\to$, which we call $\lambda_\to^{<:}$ here. The syntax of $\lambda_\to^{<:}$ remains mostly unchanged with respect to that of $\lambda_\to$, except for the inclusion of a type $\top$ that denotes a supertype of all types. The typing rules for $\lambda_\to$ can also be ported without modification. However, $\lambda_\to^{<:}$ does require extra typing rules to reason subtyping, as Figure 2.5 depicts. These rules include: (Sub Refl) and (Sub Trans), which guarantee the reflexivity and the transitivity of the subtyping relation; (Type $\top$), which extends a well-formed typing environment with type $\top$; (Sub $\top$), which formally states that $\top$ is the supertype of all types; (Subsumption), which connects a typing judgement with a subtyping judgement, meaning that if an expression e is of type τ_1 and τ_1 is a subtype of τ_2, then e is of type τ_2; and finally (Sub $\to$), which deserves in-depth discussion. The key concept behind this rule is type variance, which dictates how subtyping between complex types relates to subtyping between their components. If the constructor of a complex type preserves the subtyping relations of the simple types, it is *covariant*; if it reverses the subtyping relations, it is *contravariant*. Here, the function type constructor $\to$ is contravariant in its parameter type but is covariant in its return type. It is again Reynolds and Cardelli who first formally stated and later popularised this rule [70, 71].

Gradual Typing Static and dynamic typing have complementary strengths and are not necessarily exclusive. In practice, programming languages like Java that rely heavily on static typing may still implement dynamic typing to check constructs

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau <: \tau}\ \text{Sub Refl} \qquad \frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3}\ \text{Sub Trans}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \top}\ \text{Type } \top \qquad \frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}\ \text{Subsumption}$$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau <: \top}\ \text{Sub } \top \qquad \frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_3 <: \tau_4}{\Gamma \vdash \tau_2 \to \tau_3 <: \tau_1 \to \tau_4}\ \text{Sub} \to$$

Figure 2.5: The additional typing rules of $\lambda_{\to}^{<:}$ with respect to those of $\lambda_{\to}$.

Variables	$x \in \mathbb{X}$		
Basic Types	$\gamma \in \mathbb{G}$		
Constants	$c \in \mathbb{C}$		
Types	τ	$::=$	$\gamma \mid ? \mid \tau \to \tau$
Expressions	e	$::=$	$c \mid x \mid \lambda x : \tau . e \mid e\ e$
			$\lambda x . e \equiv \lambda x : ? . e$

Figure 2.6: The formal syntax of the gradually typed λ-calculus.

whose types cannot be resolved at compile time, *e.g.* downcasting. Many computer scientists have dedicated to the integration of both typing disciplines. Notation work in this field includes: Dynamic Typecase that extends statically typed languages with a type, often named Dynamic or Any, together with explicit forms for injecting and projecting values into and out of the Dynamic type [72, 73]; and Soft Typing that applies static analysis to dynamically typed programs for the purposes of optimisation [74] and debugging [75].

Gradual typing is yet another academic attempt to combine static and dynamic typing within a single programming language [58]. The authors presented a formal type system that lets the developers 1) control which program regions are statically or dynamically typed and 2) add or remove type annotations without any unexpected impacts on their program. Intuitively, gradual typing introduces a special type to represent statically-unknown types, and relaxes type equality into type consistency, a symmetric but not transitive relation, to accommodate the unknown type. Formally, the gradually typed λ-calculus extends the simply typed λ-calculus with an unknown type, denoted as ?. A function whose parameter is not annotated is syntactic sugar for

$$\text{(CREFL)}\ \tau \sim \tau \quad \text{(CUNR)}\ \tau \sim ? \quad \text{(CUNL)}\ ? \sim \tau$$

$$\text{(CFUN)}\ \frac{\tau_1 \sim \tau_2 \quad \tau_3 \sim \tau_4}{\tau_1 \to \tau_3 \sim \tau_2 \to \tau_4}$$

Figure 2.7: The definition of the type consistency relation $\sim$.

$$\frac{\Gamma\, x = \lfloor \tau \rfloor}{\Gamma \vdash x : \tau}\ \text{GVar} \qquad \frac{\Delta\, c = \tau}{\Gamma \vdash c : \tau}\ \text{GConst} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau . e\ :\ \tau_1 \to \tau_2}\ \text{GAbs}$$

$$\frac{\Gamma \vdash e_1 : ? \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : ?}\ \text{GApp1} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_3 \quad \tau_3 \sim \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}\ \text{GApp2}$$

Figure 2.8: The typing rules of the gradually typed λ-calculus.

a function whose parameter is of type ?. Figure 2.6 shows the complete syntax for the gradually typed λ-calculus. The key novelty of gradual typing, however, lies on its type consistency relation $\sim$, defined in Figure 2.7. The (CFUN) rule states that given two functions, if their parameters and returns are type consistent, these two functions are type consistent. Figure 2.8 details the typing rules for gradually typed λ-calculus. The authors, however, overloaded the typing environment Γ to be a function from variables to optional types ($\lfloor \tau \rfloor$ or $\perp$). The type system is parameterised on a signature Δ that assigns types to constants. The rule (GAPP1) states that when the type of a function is unknown, applying it returns also an unknown type. The rule (GAPP2) allows a function to be applied to an expression which is type consistent with the function's formal parameter. Based on the discussed formalisation, the authors finally proved that the gradually typed λ-calculus is type safe.

In their subsequent work [76], Siek and Taha further adapted the to object-oriented programming languages. In 2015, these authors observed that the term "gradual typing" seemed to have been abused. To mitigate this issue, they published a paper [77] in which they proposed a list of refined criteria for a type system to qualify as gradual typing. Some argue that these requirements are too strict to be met. Whether this is true remains an open question.

Optional type systems [78], which require type annotations to be omittable and have no runtime semantics, are another attempt to integrate static and dynamic typing. Like gradual typing, common implementations of an optional type system also introduce the special type ? representing an statically unknown type. It differs from gradual typing in that it does not have any soundness guarantee. Therefore, researchers often also call it a form of unsound gradual typing. Recent years have witnessed an increasing industrial uptake of optional typing, but how it is enforced varies across languages. For example, TypeScript checks and then erases all type annotations when transpiling to JavaScript, whereas Python's type annotations simply decorate the code, waiting to be used by separate type checkers.

2.1.2 Other Advanced Type Systems

In the field of type systems, what have been reviewed above are only a tip of the iceberg and many other advanced typing techniques exist. This section briefly and informally discusses some of them.

Intersection types Early type systems limit a term to have only one type. To model function overloading which requires a term to have more than one type, Coppo *et al.* introduced the intersection type constructor "$\wedge$" in the late 1970s [79] . They presented a type assignment system *CD*, where the used context is implicit, and proved that for a type τ and a λ-term in the β-normal form N, whether or not $N : \tau$ (N has the type τ) is decidable. However, the *CD* is flawed. It was later proved [80] that type inference for intersection type is not decidable, for terms that are not in β-normal form. Also, the CD system is closed under β-equality only for λI-calculus [53], a restriction of the full λ-calculus. Finally, in the CD system, not all terms under β-extension for the full λ-calculus can be typed [81].

In 1981, Coppo *et al.* extended the CD system with the *CDV* type assignment system [82]. They formalised the sequence type formation object, previously expressed only in the Curry's type system [79], and made the context explicit in a sequent style presentation. Because of this, an additional deduction rule, elimination for the intersection type constructor $\wedge$, was required. They introduced the universal type constant ω, first described by Girard [83], which can be assigned to any λ-term. The introduction of ω provides some important results. CDV is closed under β-equality for the full λ-calculus, while all the terms that have a *head* normal form [84] are typeable, as opposed to CD systems, where only the terms with normal form where typeable. All the proprieties proved for CD [79] systems, still hold for CDV system. Bucciarelli *et al.* [85] compared the computational properties of the λ-terms typeable with CDV type assignment system, with the ones typeable with Curry-style type systems. There main result is the prove that all the functions uniformly [86] definable by using intersection types, are also definable by using Curry types.

Neither the CD nor the CDV system is, however, complete, mainly because they require a function to always return an atomic type or ω. Barendregt *et al.* [87] introduced the BCD type assignment system, and proved its completeness, by allowing the intersection type constructor to also appear on the right hand side of the function type constructor $\rightarrow$ (*i.e.* intersection types at function returns). However, BCD systems have more than one way to infer a type, which makes it difficult for computation. Rocca *et al.* [88] later proved the principal type schemes theorem for BCD systems. Van Bakel [89] presented a restricted version of BCD systems, where the BCD system was restricted to strict types. In this work, the types that are strictly needed to assign a type to a term in BCD are considered strict types.

In general, type inference for intersection type based systems is undecidable. Researchers have put a lot of effort in imposing restrictions to obtain decidability [90, 91, 92, 93, 85, 94, 95, 96, 97, 98, 99, 100, 101], the most studied being rank 2 intersection types [102]. Rank 2 intersection type systems assume that intersection types might appear only on the left hand side of an function type constructor ($\rightarrow$) [92]. Alves *et al.* [102] proved that any term typeable by a simple type in the rank 2 intersection type system can be typed into a new term in Curry-style system, with the same type, by systematically transformations of intersection types in simple types, using fresh variables. Coppo *et al.* [90] suggested the combination of intersection type with universal quantification. Terauchi *et al.* [99] also treated the problem of rank-2 intersection types with polymorphic recursion. In this work, the authors proved the undecidability of rank-2 intersection type with polymorphic recursion, and thus answered the open question posed in Jim's work [93]. Coppo *et al.* [94] defined another decidable restriction of the intersection types, without using the rank 2 restriction. In their approach, they limited the notion of intersection, for allowing a term variable to have different types in its different occurrences. The authors also present an unification based decidable type inference algorithm , and proved the principal type property for their system. Later, Kfoury *et al.* [96] defined a new intersection type system. The authors proved that for any rank-n restriction of there system, where n is a finite number, the principal typings property, as well as decidability hold. In the previous results, only restrictions up to rank-2 are proved to be decidable. For decidable type inference in their system, they proposed a new unification algorithm, called β-unification. Kfoury *et al.* [100] also defined a methodology for computing compositional type inference, by using intersection type. In their approach, the inference for a particular fragment is obtained only from the inference results of its immediate subfragments, and thus obtained a very efficient implementation of intersection types inference algorithms.

Union types Union types, the dual notion of intersection types, are implemented in many programming languages, such as C, FORTRAN, ALGOL68, Pascal, and Caml [103]. Barbanera *et al.* [104] first introduced a type system that uses both intersection and union types, and proved that there exist translations from type systems with union types to ones without. They demonstrated that anything one can prove for a type system without union types can also be proved for the equivalent system that uses union types. Since then, many researchers have realised the strength of union types and made important contributions [105, 106, 107, 108, 109, 110, 111, 112, 113, 114].

2.2 Symbolic Execution

Symbolic execution (symex) binds symbols to variables during execution. When it traverses a path, it constructs a path condition that define inputs (including environmental interactions) that cause the program to take that path. Symex has some well known limitations [115] including path explosion, coping with external code, and out-of-theory constraints.

Harman *et al.* [116] introduced the concept of testability transformations, *source-to-source* program transformations whose goal is to improve test data generation. Following Harman *et al.*, Cadar speculated that program transformations might improve the scalability of symbolic execution in a position paper [117]. For example, a program transformation can rewrite a program to allow symbolic execution to cover portions of the program's state space that current symex engines cannot efficiently reach. The core idea is to transform expressions in a program that produce *out-of-theory* constraints into expressions that produce *in-theory* constraints. The transformed program under-approximates the input program's behaviour: it is precise over every component of the program's state it binds to a value in G and reifies its ignorance of component's actual value when it binds $\perp$ to that component.

2.2.1 Handling Unknown via Concretisation

Solver unknowns bedevil symex. One way to handle them is to resort to concrete execution. *Dynamic Symbolic Execution* (DSE) [118] does this by lazily concretising the subset of the symbolic state for which the solver return unknown. Concolic testing [119, 120, 121] or *white-box fuzzing* [122] is an extension to DSE that initially follows the path that a concrete input executes. Concolic testing requires concrete inputs from the user and then searches a neighbourhood around the path executed under its concrete input by negating the values of the current branch conditions from the current followed path. First concolic testing flips the closest branch to the end of execution and then, it continues to do so upwards to the entry point of the program. To flip the path condition, concolic testing uses an SMT solver. Whenever reaching a constraint that the solver cannot solve, or for which the solver times out, the concretisation methods, such as DSE and concolic testing, get a concrete value that can be either random, or obtained under different heuristics [121].

2.2.2 Constraint Encoding

During its execution, a program under analysis produces constraints in the different domains of its data types, such as constraints over strings, or floats. Before calling the solver, a symex engine needs to encode the constraints in a language that the solver understands. Some of these constraints are difficult for the solver: the literature provides a couple of constraint encoding mechanism to cope with them.

Strings Quine proved that the first-order theory of string equations is undecidable [123]. Since then different techniques have emerged implementing decision procedures over fragments of string theory. All these approaches have limitations: either they do not scale; they support a fragment of string theory that is not well-aligned with string expressions developers write; they require fixed length strings; or they require a maximum length and loop through all possible lengths up to that maximum. There are three main categories of string solvers: automata, bit-vector, and word-based.

Automata-based solvers [124] use regular languages or context-free grammars to encode the string constraints. The idea is to construct a finite-state automata that accepts all the strings that satisfy the path conditions in a program. Building this automata requires handcrafted building algorithms for the set of string operations that we intend to support and the set of values that the program accepts as inputs [125]. When a new string constraint is added to the path condition, these approaches refine the automaton to not accept the strings that violate the newly added constraint. The refinement process is automatic: we remove from the set of values that the automata accepts the ones that are not valid for the new constraint. Infeasible paths construct automata that do not accept any strings. For string constraints, the automaton becomes the solver. Automata-based string solvers tend not do not combine strings with other data types [126], as combining string automata with other data types and operations over them, requires handcrafted initial automata for the particular data types and operations that the program under analysis uses in conjunction with the string operations. Yu *et al.* [127] tackles the problem of using an automaton to handle both string and integer constraints, but no other data types.

Bit-vector based symex engines convert string constraints into the domain of bit-vectors [128]. The bit-vector solvers require a maximum string length and lack scalability. Specifying a maximum length per each query that symex sends to the solver would require comprehensive annotations, so symex engines use a global maximum. In general, string length is domain-specific and specifying a maximum length for an entire program is problematic: for example, a database query might be hundreds of characters long, while a command line flag can occupy only one character. The string length restriction means that users must specify a single length for all strings in the program. When too big, this length slows the solver; when too small, it limits the analysis to small strings. Thus, these engines typically execute a program over different length strings [128]. The bit-vector encoding of values causes exponential blow up in model size, 2^n where n is the length of a bit-vector, since each bit becomes a propositional variable for the underlying SAT solver and hampers scalability.

Word-based string solvers define a subtheory that uses rewriting rules and axioms tailored to a fragment of string theory. Word-based string solvers escape Quine's result by not handling all string expressions. CVC4 [129] and S3 [130] support constraints over unbounded strings restricted only to length and regular expression membership operators. Z3-STR [131] supports unbounded strings together with the concatenation, string equality, substring, replace, and length operators.

Floating Point Bit-blasting is the most widely used technique for solving floating-point constraints. Bit-blasting converts floats into bit-vectors and encodes floating point operations into formulae over these bit-vectors. Bit-blasting floating-point constraints generates formulae that require a huge number of variables. For example, Brillout *et al.* [132] showed than when using a precision of only 5 (mantissa width) addition or subtraction over floating point variables requires a total of 1035 propositional variables to be encoded in bit-vector theory. In a similar scenario, multiplication or division requires a total of 1048 propositional variables. Because of this, bit-blasting for floating point constraints does not scale to large programs [133].

Testing is also used to solve floating point constraints. Microsoft's Pex symex engine can use the FloPSy [134] floating point solver. FloPSy transforms floating point (in)equalities into objective functions. For example, the predicate if ((Math.Log(a) == b) becomes the objective function $|Math.Log(a) - b|$ for which we want to find values of a and b that yield 0. FloPSy uses hill climbing [135] to find values for a and b. Fu *et al.* proposes Mathematical Execution (ME) [136]. They capture the testing objective through a function that is minimised via mathematical optimisation to achieve the testing objective. Given a program under test P, they derive another program P_R called representing function. P_R represents how far an input $x \in$ dom(P) is from reaching the set $x|P(x)$ is wrong. P_R returns non-negative results and P_R is the distance between the current input and an error triggering input. Further, they minimise P_R. Souza *et al.* [137] integrate CORAL, a meta-heuristic solver designed for mathematical constraints, into PathFinder [138].

Klee-FP [139] and Klee-CL [140] replaces floating-point instructions with uninterpreted functions. Klee-CL and Klee-FP apply a set of canonising rewritings and further do a syntactic match of floating-point expressions trees. Both Klee-CL and Klee-FP solve floating point constraints by proving that them are equivalent (or not) with an integer only version of the program. They do this by matching the equivalent expression trees. Thus, the main limitation is the requirement of having the both version of the programs available: the floating point and the integer implementation.

Other approaches use constraint programming [141] to soundly remove intervals of float numbers that cannot make the path condition true. Botella *et al.* [142] solve floating point constraints using interval propagation. Interval propagation tries to contract the interval domains of float variables without removing any value that might make the constraint true. These approaches are either imprecise or do not scale.

Despite these attempts, constraint solving for string and floating point remains difficult and does not scale to programs at industry scale. Chapter 4 proposes a general program transformation that utilises types to change a program's semantics in a controlled way, thereby facilitating existing program analysis, such as symbolic execution, for programs that compute with strings or floats.

2.2.3 The State Of the Art in Symbolic Execution

Significant research has tackled the problem of applying symex to real world programs [115]. Some approaches aim to improve a symex engine's interactions with the constraint solver. Klee [118] and Green [143] cache solved constraints. Lazy initialisation delays the concretisation of symbolic memory thereby avoiding the concretisation of states that additional constraints make infeasible [144]. Memoized symex [145] and directed incremental symbolic execution [146] reuse query results across different symex runs.

Other approaches improve the scalability of symex by mitigating the path explosion problem. Veritesting [147] proposes a path merging technique that reduces the number of paths being considered as a result of reasoning about the multiple merged paths simultaneously. MultiSE [148] perform symbolic execution per method, rather than per entire program. MultiSE merges different symex paths into a value summary, or a conditional execution state.

For an unsolvable constraint, symex concretises its variables, causing incompleteness: it cannot reason on the rest of the state space. Pasareanu *et al.* [149] delay concretisation to limit the incompleteness. They divide the clauses into simple and complex ones. When a simple clause is unsatisfiable, the entire PC becomes unsatisfiable. This can avoid reasoning about the complex clauses. Khurshid *et al.* [150] concretises objects only when they need to access them.

A recent study [151] show that 33 optimisation flags in LLVM decrease symex's coverage on coreutils . Overify [152] proposes a set of compiler optimisations to speed symex. Sharma *et al.* [153] exploit these results and introduce undefined behaviour to trigger various compiler optimisations that speed up symex [117]. Under-Constrained symex [154] operates on each function in a program individually. Abstract subsumption [155] checks for symbolic states that subsume other ones and remove the subsumed ones. Ariadne transforms numerical programs to explicitly check exception triggering conditions in the case of floats [156].

2.3 Other Related Dynamic Analysis Techniques

Despite being a fundamental software engineering task, debugging in practice is slow and prone to producing bad fixes that introduce new errors or only partially fix the problem [157]. To remedy the situation, researchers have proposed a significant number of techniques. This section reviews previous dynamic debugging techniques, which are related to Chapter 5.

Value Tracking Bond *et al.* [158] pioneered research into tracking erroneous values to their origin. By modifying a Java virtual machine, they developed a dynamic analysis that pinpoints the assignment of a `null` that further leads to a `null` dereference. In the tool called Casper, Cornu *et al.* [159] improve Bond *et al.*'s work by also recording how a `null` propagates from its production to the dereference site.

State-Based Fault Localisation Wong *et al.* classify fault localisation techniques into eight categories [160], in which state-based fault localisation, as its name suggests, uses program states (*i.e.* variable to value pairs) to localise bugs. Delta Debugging [161] systematically trims failure-triggering data, like a program input. It can also be applied to an execution trace, which inherently has two dimensions: along one dimension lies program states; the other dimension is temporal, representing a series of state transitions. When Zeller [162] first applied Delta Debugging to program states, he sought to narrow the state difference between a passing and a failing execution. Later, Cleve and Zeller applied Delta Debugging to state transitions to isolate a chain of transitions that cause a failure [163].

Dynamic Slicing Given a slicing criterion, a triple of inputs to a program p, a location l in p, and a variable x, dynamic slicing [164] returns a subsequence of p such that executing it preserves the behaviour for x at l. A typical dynamic slicer first builds a dependence graph from executing p and then computes a slice for the slicing criterion by traversing the dependence graph. Observation-based slicing [12] is a variant of dynamic slicing, which realises behaviour preservation by requiring the execution of p's sliced variants to halt and the values of x to remain unchanged.

Dynamic Taint Analysis Tracking and auditing the flow of suspicious data is a critical problem in security. Dynamic taint analysis [165] tackles this problem by introducing and monitoring a special mark, which it calls taint. Specifically, it augments data from a potentially untrusted source, such as the network, with a taint field. When the program is executed, dynamic taint analysis uses some predefined rules to propagate the taint field and checks whether data consumed at important locations are tainted. Devised to combat security problems, dynamic taint analysis has been used in debugging. Multithreaded programs are typically non-deterministic.

To debug them, a debugger needs to record events of interest at runtime, which poses an overhead. Ganai *et al.* use dynamic taint analysis to identify inputs that can affect control flow or data flow of a multithreaded program [166]. They demonstrate that identifying these inputs effectively mitigates the overhead of runtime logging and increases coverage.

Despite the rich literature, debugging remains “the dirty little secret of computer science” [167]. Chapter 5 presents a dynamic program analysis technique that tracks a value of interest through its propagation chain back to its origin, answering two fundamental debugging questions: “Where did this value come from?” and “How did it get here?”.

2.4 Chapter Summary

This chapter reviews the literature on program analysis, from which a research opportunity arises: Can we utilise program transformation to extend the reach of existing program analysis techniques? For example, both static typing and symbolic execution are effective for a particular set of programs, but they have limitations: static typing is inapplicable to programs written in a dynamically typed language, where type annotations are omitted and type inference is undecidable; symbolic execution suffers from various problems, one of which is unsupported theories. These limitations can, in fact, be addressed via program transformation. Optional type annotations, a degenerate case of program transformation, enable static typing on the typed regions of a program. For symbolic execution, a well-controlled program transformation can convert unsupported theories to supported theories. Indeed, researchers have already taken this route. Ariadne [156] uses symbolic execution to detect floating point exceptions, after replacing floating point to arbitrary precision rationals, whose constraints contemporary solver are able to solve.

Chapter 3

To Type or Not to Type: Quantifying Detectable Bugs in JavaScript

As Section 2.1 discusses, static typing is resurgent. Even within the community of dynamically typed languages, static typing, particularly optional typing, is gaining ground. This chapter empirically investigates this phenomenon, using JavaScript as its subject.

JavaScript is a multi-paradigm programming language that experiences increasing popularity and importance. Indeed, it is often called the assembly of the web [168]; it is the core language of many long-running projects with public version control history. JavaScript's prevalence in the web and wider use in general programming have drawn much attention. Richards *et al.* [169] and Ratanaworabhan *et al.* [170] concurrently analysed the run-time behaviour of real-world JavaScript applications. The former invalidated some common assumptions and called for more flexible static analysis, while the latter focused on performance and concluded that the existing benchmarks were not representative. Richards *et al.* further investigated the use of `eval`, a powerful but controversial function [171]. The authors showed that `eval` was pervasive and justified its importance. Pradel and Sen carefully examined another dubious feature of JavaScript, implicit type conversion [50]. Similar to `eval`, implicit type conversion is widely used, but much less harmful than commonly assumed. Nikiforakis *et al.* extensively studied the real-world uses of JavaScript's remote library inclusion and revealed four types of vulnerabilities [172]. Apart from research on the language, while Ocariza *et al.* and Hanam *et al.* classified JavaScript bugs and investigated their root causes [173, 174, 175], Selakovic and Pradel restricted themselves to performance issues [176].

JavaScript is also a dynamically typed language with permissive typing rules. For example, it deems the expression `"1" - 1` type correct and evaluates it to `"10"`. Academia and industry have proposed many program analysis techniques to help developers combat this permissiveness, especially when they are composing large and complex programs in JavaScript. Three companies, in particular, have viewed static typing as important enough to invest in static type systems for JavaScript: first Google released Closure[1], then Microsoft published TypeScript[2], and most recently Facebook announced Flow[3]. What impact do these static type systems have on code quality? More concretely, how many bugs could they have reported to developers, if they had been used during development?

The fact that long-running JavaScript projects have extensive version histories, coupled with the existence of static type systems that support optional typing and can be applied to JavaScript programs with few modifications, permits an experiment for us to under-approximately quantify the benefit of static type systems on code quality. Specifically, this benefit is measured in terms of the proportion of bugs that were checked into a source code repository that might not have been if the committer were using a static type system that reported an error on the bug.

In this experiment, we sample public software projects, check out a historical version of the codebase known to contain a bug, and add type annotations. We then run a static type checker on the altered, annotated version to determine if the type checker errors on the bug, possibly triggering a developer to fix the bug. Unlike a controlled human subject experiment, our experiment studies the effect of annotations on bugs in real-world code-bases not the human annotator, just as surgery trials seek to draw conclusions about the surgeries, not the surgeons [177], despite our reliance on human annotation. More generally, decision makers can use

[1]`https://developers.google.com/closure/compiler/`
[2]`http://www.typescriptlang.org/`
[3]`http://flowtype.org/`

this "what-if" style of experimentation on software histories to help decide whether to adopt new tools and processes, like static type systems.

We employ two static type checkers, Flow and TypeScript, for three reasons. First, Flow and TypeScript were created and are actively maintained by two industrial giants. In particular, TypeScript's community has developed DefinitelyTyped, a mature collection of annotated interfaces for popular JavaScript libraries, for which Feldthaus and Møller built an analysis tool [178]. In contrast, other tools, such as Roy[4], have not been actively updated. Second, to use Flow and TypeScript, one simply needs to add type annotations to an existing JavaScript program. Many of the alternatives require a complete rewrite, because their grammar differs from that of JavaScript. Finally, Flow and TypeScript largely share annotation syntax, which allowed us to reuse some annotations.

This experiment targets bugs that are *public*, actually checked in and visible to other developers, potentially impacting them; public bugs notably include field bugs, which impact users. We consider public bugs because they are observable in software repository histories. Public bugs are more likely to be errors understanding the specification because they are usually tested and reviewed, and, in the case of field bugs, deployed. Thus, this experiment under-approximates static type systems' positive impact on software quality, especially when one considers all their other potential benefits on documentation, program performance, code completion, and code navigation.

This chapter makes the core contribution of quantifying the public bugs that static type systems detect and could have prevented: 15% for both Flow 0.30 and TypeScript 2.0, on average. Our experimentation artefacts are available at: `https://2-type-not-2.github.io/`.

3.1 Problem Definition

This section defines the research problem that we aim to answer.

Definition 3.1.1 (*ts*-detectable)**.** *Given a static type system ts, a bug b is ts*-detectable

[4]`http://roy.brianmckenna.org/`

when adding or changing type annotations causes the program p containing b to error on a line changed by a fix and the new annotations are consistent *with f, a fixed version of p.*

We assume a fix increases the adherence of p to the specification and we consider only its effect on resolving b. The added or changed type annotations may affect several terms, or only one. These annotations are *consistent* if 1) when they are carried to f, f type checks, and 2) the type of every annotated term in p is a supertype of that term's type when an oracle precisely annotates it in f. In this experiment, we can only strive to achieve consistency, because we do not have the ideal oracle that precisely annotates f and in general, a fix may only partially resolve the bug or introduce new bugs. One can download our experimental data to verify how well we have reached this goal. Consistency implies that we do not intentionally add ill-formed type annotations. For example, when `b` and `c` have type `number`, changing `var a = b + c` to `var a:boolean = b + c` incorrectly annotates `a` as `boolean`, triggering a type error. If such ill-formed annotations are not ruled out, one can use them to "detect" any bug, even type-independent failures to meet the specification.

Let L be a programming language, like JavaScript, and L_a be a language based on L with syntactical support for type annotations, like Flow or TypeScript. Let $P = \{p_1, p_2, \cdots, p_m\}$ denote a set of buggy programs. Let a be an annotation function that transforms a program $p \in L$ to $p_a \in L_a$. Finally, let tc be a type checking function that returns `true` if an annotated program p_a type checks and `false` otherwise.

We annotate each buggy program p_i that is in P and written in L, and observe whether it type checks. We calculate the percentage of bugs that a static type system detects over all collected ones. Our measure of a static type system's effectiveness at detecting bugs follows:

$$\frac{|\{p_i \in P \mid \neg tc(a(p_i))\}|}{|P|} \tag{3.1}$$

Equation 3.1 reports the portion of bugs that could have been prevented had a type system, like Flow or TypeScript, reported type errors that caused a developer to notice and fix them. Depending on the error model of a static type system, a might be the identity function, i.e. add no annotations. For instance, both Flow and TypeScript are able to detect errors in reading an undefined variable without any annotation.

3.2 Methodology: Time-Travel Experiment

An experiment is an empirical procedure conducted to investigate the causal impact a manipulated independent variable, or treatment, has on an outcome. Usually, the subjects of an experiment are divided into an experimental group that receives the treatment and a control group that does not; ideally, this should be the sole difference. On one end of the experiment spectrum are powerful controlled ones, in which experimenters tightly control both the experimental environment and the independent variable, assigning it either randomly (randomised control trial [179]) or using a non-random criterion (quasi experiment [180]), like illness severity. At the other end are experiments with loose control. An extreme case is natural experiment [181], in which nature, not researchers, assigns the treatment.

Leveraging the version history of software projects, we propose a experimentation methodology: time-travel experiment. We “travel back” in time to a particular historical version of a project, assign a “treatment” to that version, and compare its outcome with the reality that lacks the treatment. The concept of a treatment in a time-travel experiment is broad: it can be any technique or tool that helps with the software development, *e.g.*, a program analysis technique or an IDE. In this chapter, we substantiate it with static typing.

Being the gold standard, a randomised control trial randomly allocates treatment to mitigate selection bias, as Figure 3.1 shows. Our time-travel experiment also randomly assigns treatment; its key distinction is that the subjects in the experimental and the control groups are actually the same, freeing it from intrinsic confounding factors, such as subject individuality, as Figure 3.2 illustrates. Imagine a drug trial in which you could compare a patient’s outcome, both with and without the drug treatment, across two universes! When applied to software engineering, this

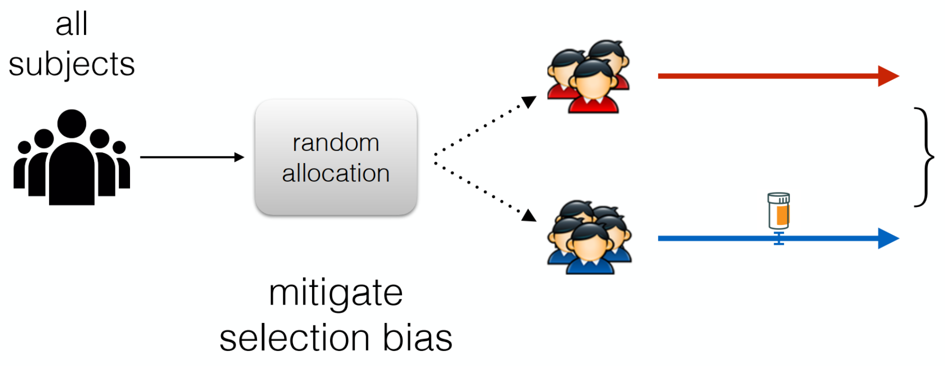


Figure 3.1: The random controlled trial.

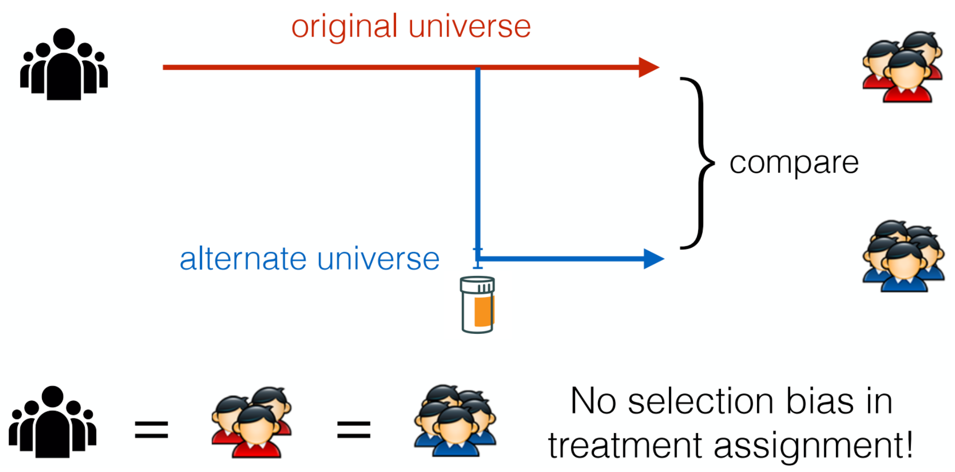


Figure 3.2: Time travel experiment.

experimental methodology shines in its scalability and reproducibility, and may allow us to tackle and more confidently answer questions that have traditionally been out of reach due to the cost of human subject studies on developers.

Historical Treatment Methodology Previous work has mined software repositories to measure a property of an existing, mined version, like the number of vulnerabilities a static analysis detects [182] or warnings a lint tool reports [183]. Most of these studies are observational; a tool is used on history, but neither the history nor the historical artefacts are modified. Work does exist that applies a treatment to the history and/or artefacts in the history (e.g. changing the source code) and measures the impact of such an intervention. Le Goues *et al.* automatically modified source code to fix bugs [184]. Brun *et al.* introduced different orderings of historical branch merges from repository histories to determine how early conflicting changes could have been detected [185]. Bird and Zimmermann replayed development activity on alternative branch structures to identify branches that acted as bottlenecks to code movement [186]. We posit that this methodological approach of applying a treatment to a historical snapshot of a project and measuring its impact can be a powerful empirical tool.

3.2.1 Leveraging Fixes

Bug localisation often requires running the software and finding a bug-triggering input. Code bit rots quickly; frequently, it is very difficult to build and run an old version of a project, let alone find a bug-triggering input. Worse, many of our subjects are large, some having as many as 1,144,440 LOC (Table 3.1). To side-step these problems, we leverage fixes to localise bugs. For $p \in L$, we assume we have a commit history as a sequence of commits $C = \{c_1, c_2, \cdots, c_n\}$. When $c_i \in C$ denotes a commit that attempts to fix a bug, the code base materialised from at least one of its parents c_{i-1} is buggy. A fix's changes help us localise the bug: we minimally add type annotations only to the lexical scopes changed by a fix. We add annotations until the type checker errors or we decide neither Flow nor TypeScript would error on the bug. This partial annotation procedure is grounded on optional typing, which both Flow and TypeScript employ. These two type systems are permissive. When they cannot infer the type of a term, they assign the wildcard `any`, similar to Abadi *et al.*'s *Dynamic* type [72], to it.

This procedure allows us to answer: "How many public bugs could Flow and TypeScript have prevented if they had been in use when the bug was committed?", under the assumption that one knows the buggy lines. By "in use", we mean that developers comprehensively annotated their code base and vigilantly fixed type errors. The assumption that developers knew the buggy lines is not as strong as it seems because, under the counterfactual that developers were comprehensively and vigilantly using one of the studied type systems, the bug-introducing commit is likely to be small (median of 10 lines in our corpus) and to localise some of the error-triggering annotations, while the rest of the annotations would already exist in the code base.

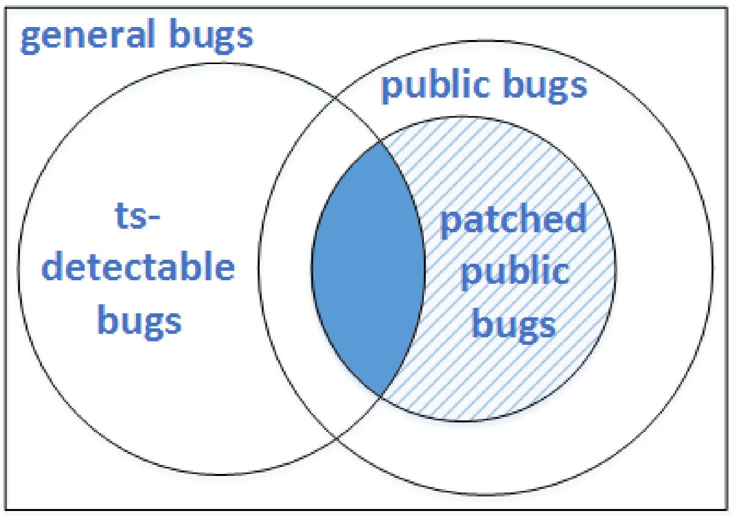


Figure 3.3: The error model of this experiment.

Limitations Four limitations of our approach are 1) a "fix" may fail to resolve and therefore localise the targeted bug, 2) a minimal, consistent bug-triggering annotation may exist outside the region touched by the fix, 3) we may not succeed in adding consistent annotations (3.1.1), and 4) the annotation we add may cause the type checker to error on a bug unrelated to the bug targeted by the fix. Further, considering only fixed, public bugs introduces bias. We restrict our attention to these bugs for the simple reason that they are observable. We have no reason to believe this bias is correlated with *ts*-detectability. Section 3.6 discusses other threats to this work.

3.2.2 Error Model

The subjects of this experiment are identified and fixed public bugs. As Figure 3.3 shows, we aim to classify these bugs into those that are *ts*-detectable (the solid partition of fixed public bugs) and not (the hashed partition of fixed public bugs).

Type systems cannot detect all kinds of fixed public bugs. What sorts of bugs do our type systems detect and may prevent? Type systems eliminate a set of bad behaviours [29]. More specifically, Flow or TypeScript detects and may prevent type mismatches, including those normally hidden by JavaScript's coercions, and undefined property and method accesses. Additionally, both Flow and TypeScript identify undeclared variables.

```
1 function addNumbers(x, y) {
2     return x + y;
3 }
4 console.log(addNumbers(3, "0"));
```

(a) The buggy program.

```
1 function addNumbers(x, y) {
2     return x + y;
3 }
4 console.log(addNumbers(3, 0));
```

(b) The fixed program.

```
1 function addNumbers(x:number, y:number) {
2     return x + y;
3 }
4 console.log(addNumbers(3, "0"));
```

(c) The annotated, buggy program.

Figure 3.4: JavaScript coerces `3` to `"3"` and prints `"30"`. From the fix, we learn that this behavior was unintended and add annotations that allow Flow and TypeScript to detect it.

3.2.3 Example

Assume `addNumbers` in Figure 3.4a is intended to add two numbers, but the programmer mistakenly passes in a string `"0"`. Because of coercion, a controversial feature that enriches a language's expressivity at the cost of undermining type safety and code understandability [187], + in JavaScript can take a pair of `number` and `string` values. Thus, Figure 3.4a converts the number to a string, concatenates the two values, and prints `"30"`. By reading the fixed program in Figure 3.4b, we infer that both parameters are expected to have type `number`. We partially annotate the program, shown in Figure 3.4c, enabling Flow and TypeScript to signal an error on line 4 and detect this bug. If, in addition to this bug, we had shown four other bugs to be undetectable, Equation 3.1 would evaluate to $\frac{1}{5}$.

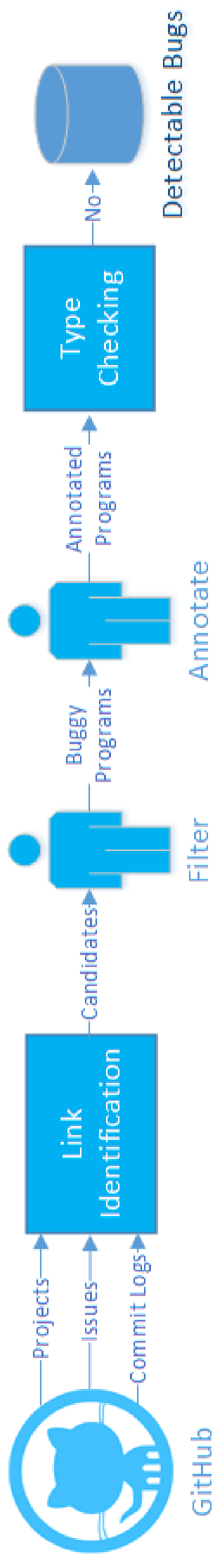


Figure 3.5: The workflow of our experiment.

3.3 Experimental Setup

Our experimental setup is similar to that of Le Goues *et al.* [184]. They aimed to determine, for a sample of real world historical bugs sampled from GitHub projects, what proportion of bugs would been fixed through automatic program generation (Defects4J [188] enables similar studies and evaluations on real world bugs for Java-targeted tools). We perform a sampling of historical real world JavaScript bugs and attempt to determine what proportion of bugs would have been detected using static JavaScript type systems if the authors had been using them.

Figure 3.5 depicts the general process of our study, which comprised many phases, methodological decisions, investigations, and techniques. This section describes the types of data we gathered and how we selected the data to use, discusses potential threats and how we mitigated them, reports on preliminary investigations, and presents our annotation process and various tactics used.

3.3.1 Corpus Collection

We seek to construct a corpus of bugs that is representative and sufficiently large to support statistical inference. As always, achieving representativeness is the main difficulty, which we address by uniform sampling. We cannot sample bugs directly, but rather commits that we must classify into fixes and non-fixes. Why fixes? Because a fix is often labelled as such, its parent is almost certainly buggy and it identifies the region in the parent that a developer deemed relevant to the bug. To identify bug-fixing commits, we consider only projects that use issue trackers, then we look for bug report references in commit messages and commit ids (SHAs) in bug reports. This heuristic is not only noisy; it must also contend with bias in project selection and bias introduced by missing links.

Missing Links A link interconnects a bug report and a commit that attempts to fix that bug in a version control system. Historically, many of these links are missing, especially when the developer must remember to add them, due to inattentiveness, distractions, or fire drills. Naïve solutions to the missing link problem are subject to bias [189]. GitHub provides issue tracking functionality in addition to source code management and provides tight integration to ease linking. In addition, when pull requests or commit messages reference bugs in the issue tracker, GitHub automatically links the source code change to the bug. For these reasons, projects that use pull requests, issue tracking, and source code management suffer far less from the linking problem [190].

To validate this and assess the missing link problem in the context of GitHub ourselves, we collected eight JavaScript projects, using a set of criteria including project size, popularity, number of contributors, and the use of Node.js and jQuery. We manually inspected them and observed that because project norms dictate that developers refer to bugs in requests and commits to enable GitHub's automatic linking, the overwhelming majority complied with the practice, thus mitigating the missing link problem.

Identifying Candidate Fixes Figure 3.6 depicts our procedure for identifying candidates of bug-resolving commits. For a project, we extract all bug ids from the issue tracker, then search for them in a project's commit log messages; concurrently, we extract all SHA from the version history, and search for them in the project's issues. GitHub allows developers to label issues as bug reports, but we choose not to use this functionality and consider all tracked issues, as we were uncertain what bias this labelling could introduce. When we find a match, we have a candidate fix that we store as a triple consisting of the SHA of the candidate, the SHA of its parent, and the bug report ID. We cross-check matches from commit logs against matches from the bug reports. If a fix has more than one parent, the algorithm stores a distinct triple for each parent for later human inspection. For every automatically identified candidate, we manually assess whether it is actually an attempt to resolve a bug, rather than some other class of commit, like a feature enhancement or code refactoring. We also filter bug reports written in a language other than Chinese and English, and fixes that do not modify JavaScript.

The resulting set of bugs is a biased subset of all fixed public bugs. GitHub may not be representative of projects, since proprietary projects tend not to use it. While we have argued the problem is less acute, missing links persist. Finally, we may not correctly identify bug-fixing commits. We contend, however, there is no reason,

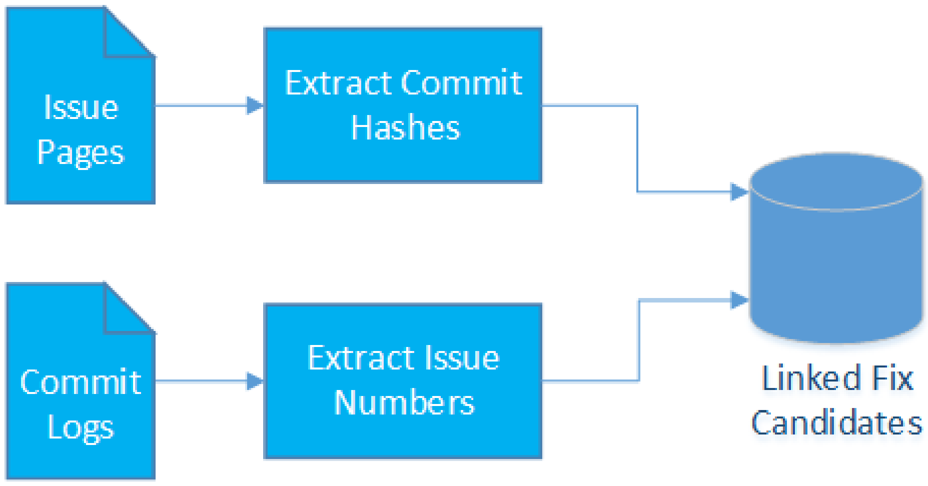


Figure 3.6: The automatic identification of fix candidates that are linked to bug reports.

from first principles, to believe that there is a correlation between the ability of Flow or TypeScript to detect a bug and the existence of a link between that bug and the fixing commit. Thus, any link bias in the subset is unlikely to taint our results.

Corpus To report results that generalise to the population of public bugs, we used the standard sample size computation to determine the number of bugs needed to achieve a specified confidence interval [191]. On 19/08/2015, there were 3,910,969 closed bug reports in JavaScript projects on GitHub. We use this number to approximate the population. We set the confidence level and confidence interval to be 95% and 5%, respectively. The result shows that a sample of 384 bugs is sufficient for the experiment, which we rounded to 400 for convenience.

To collect these 400 bugs, we took a snapshot of all publicly available JavaScript projects on GitHub, with their closed issue reports. We selected a closed and linked issue uniformly at random, using the procedure described above and stopped sampling when we reached 400 bugs. The resulting corpus contains bugs from 398 projects, because two projects happened to have two bugs included in the corpus.

Table 3.1 shows the size statistics of the corpus. The project size varies largely, ranging from 32 to 1,144,440 LOC, with a median of 1,736. The smallest project is `dreasgrech/JSStringFormat`, a personal project with a single committer. It minimally implements .NET's `String.Format` that inserts a string into another based on a specified format. We sampled from GitHub uniformly at random so our corpus

	Max	**Min**	**Mean**	**Median**
Project	1144440	32	18117.9	1736
Fix	270	1	16.2	6

Table 3.1: The size statistics in LOC of the projects and fixes in our corpus, which includes 398 projects[6].

contains such small projects roughly in proportion to their occurrence in GitHub. For a commit, GitHub's Commits API[5] does not return a diff; it returns summary data, notably a pair of numbers, the count of additions and deletions. From this pair, the number of modifications can only be implicitly bounded by `min(#adds, #dels)`. Because developers think in terms of modified lines, not lines of diff, we counted the line in which Git's word diff reported modifications. Most bug-fixing commits were quite small: approximately 48% of the fixes touched only 5 or fewer LOC, and the median number of changes was 6. We did not explicitly track the number of scopes; that said, most of the fixes modified a single scope. The complete corpus can be downloaded at: `https://2-type-not-2.github.io/`.

3.3.2 Preliminary Study

To quantify the proportion of public bugs that the two static type systems detect, and could have prevented, our study must 1. find a time bound on per-bug assessment and annotation in order to make our experiment feasible, 2. establish a manual annotation procedure. Additionally, our study also aims to classify *ts*-undetectable bugs. To speed the main experiment, we wanted to define a closed taxonomy for undetectable bugs. To these ends, we conducted a preliminary study on 78 bugs, sampled from GitHub using the above collection procedure.

A histogram of our assessment times showed that, for 86.67% of the bugs, we reached a conclusion within 10 minutes, despite the fact that we were simultaneously defining our annotation procedure. Thus, we set M, the maximum time that we can spend annotating a bug, to be 10 minutes.

[5]`https://developer.github.com/v3/repos/commits`

[6]Of the 398 projects, only 375 were still available on GitHub at the time of this experiment.

Algorithm 1 Manual Type Annotation.

```
Input: $M$, the maximum time to spend annotating a bug
Input: $B$, the list of sampled bugs
Output: $O$, the assessment of all sampled bugs
 1: while $B \neq []$ do
 2:     $b :=$ head $B$; $B :=$ tail $B$;
 3:     Check out $p$, the buggy program that contains $b$
 4:     for all $ts \in \{\text{Flow}, \text{TypeScript}\}$ do
 5:         start := now(); $O_{ts}[b] :=$ Unknown;
 6:         while now() $\leq$ start + $M$ do
 7:             Read $p$, $b$'s fix, and the bug report
 8:             $p_a := a(p)$
 9:             if $tc_{ts}(p_a)$ then
10:                 $O_{ts}[b] :=$ True; break
11:             end if
12:             if we deem $b$ $ts$-undetectable then
13:                 Justify the assessment
14:                 Categorise $b$ using the taxonomy below
15:                 $O_{ts}[b] :=$ False; break
16:             end if
17:         end while
18:     end for
19: end while
```

Taxonomy of Undetectable Bugs To build a taxonomy of bugs that Flow and TypeScript do not currently detect, we used *open coding*. Open coding is a qualitative approach for categorising observations that lack a priori organisation [192]. The researchers assessed each observation and iteratively organised them into groups they deem similar. Starting from JavaScript's error model, we refined the taxonomy. At the end of our preliminary study, our taxonomy contained JavaScript's `EvalError`, `RangeError`, `URIError`, and `SyntaxError`. To these, we added `StringError`, such as malformed SQL queries. The logical errors we encountered caused us to add `BranchError`, `PredError` that are caused by incomplete or wrong predicates, `UIError`, and `SpecError`, a catch-all for other failures to implement the specification. Regular expressions are built into and widely used in JavaScript, so we included `RegexError`. Finally, we added `ResError` to handle resource errors, like out of memory, and `APIError` to capture errors such as using a deprecated call.

3.3.3 Annotation

Procedure 1 defines our manual type annotation procedure. Because we annotate each bug twice, once for each type system, our experiment is a within-subject repeated measure experiment. As such, a phenomenon known as learning effects [193] may come into play, as knowledge gained from creating the annotations for one type checker may speed annotating the other. To mitigate learning effects, for a bug b in B, we first pick a type system *ts* from Flow and TypeScript uniformly at random, so that, on average, we consider as many bugs for the first time for each type system. If b is not type related "beyond a shadow of a doubt", such as misunderstanding the specification, we label it as undetectable under *ts* and categorise it based on item 3.3.2, skipping the annotation process. If not, we read the bug report and the fix to identify the *patched region*, the set of lexical scopes the fix changes.

Combining human comprehension and JavaScript's read-eval-print loop (REPL), *e.g.* Node.js, we attempt to understand the intended behaviour of a program and add consistent and minimal annotations that cause ts to error on b. We are not experts in type systems nor any project in our corpus. To combat this, we have striven to be conservative: we annotate variables whose types are difficult to infer with `any`. Then we type check the resulting program. We ignore type errors that we consider unrelated to this goal. We repeat this process until we confirm that b is *ts*-detectable because *ts* throws an error within the patched region and the added annotations are consistent (Section 3.1), or we deem b is not *ts*-detectable, or we exceed the time budget M.

3.3.4 Annotation Tactics

The key challenge in carrying out Procedure 1 is efficiently annotating the patched region. As previously stated, we rely on optional typing to allow us to locally type a patched region. Sometimes, we must eliminate type errors so the type checker reaches the patched region. In practice, this means we must handle modules. With modules out of the way, we use a variety of tactics to gradually annotate the patched region. The first, and most important, tactic is to read the bug-fixing commit. For example, the fix of `naugtur/transitionrunner:1` (using `author/project:issue` to refer to our dataset) assigns the empty string to the variable `initialClass` when it is `null`. Therefore, we add an annotation to indicate `initialClass` can be `null`. We also use online documentation, when it exists. For example, accessing a non-existing property triggers bug `Gozala/narwhal-xulrunner:5`. We read the documentation of `nsIOutputStream` at Mozilla Developer Network to learn and and inject the appropriate annotation. To handle globals, we use type shims, which we describe below. As noted, we have striven to add type annotations that are consistent (Section 3.1) with the the ideal, fully annotated, and fixed version of the buggy program.

Modules For a subject buggy program, we first run the type checker without any type annotations. Often the type checker reports an error before reaching the patched region due to failures to import modules. We search for the declaration of the variables in the fix and try to see whether they use any module methods, like jQuery's `$`. Finding variable declarations can be nontrivial in JavaScript, precisely because a lack of types hindered our understanding of the program. If we deem a missing module to be unrelated to the bug, we annotate it as `any` to eliminate such type errors. For example,

```
// Flow and TypeScript cannot properly
// import express.js
var express = require('express');
var app = express.createServer();
```

becomes

```
var express:any = { };
var app = express.createServer();
```

For TypeScript, if we deem a missing module related to the bug and it exists in DefinitelyTyped[7], we include it. If the bug stems from a misuse of a library that has an annotated interface in DefinitelyTyped, we reuse DefinitelyTyped's annotations. For example, deprecated internal data models in `ember-cli` caused the bug `sivakumar-kailasam/broccoli-leasot:55`. To solve it, we borrowed DefinitelyTyped's annotations for `ember-cli`. If the missing module is related but lacks an

[7]A project from the TypeScript community that provides annotated interfaces for popular JavaScript libraries, at `https://goo.gl/xvDaSI`

interface in DefinitelyTyped or we are using Flow, we construct a type shim for it, manually inferring the types from the module's documentation or its code base.

Type Shims In general, a patched region contains free identifiers that we need to annotate. Introducing casts would increase the annotation tax. Our workaround is to introduce a *type shim*[8], a set of type bindings for the free identifiers. From within the patched region, the rest of the program can be viewed as a module for which we can define a shim as a set of interfaces. We have aimed to construct consistent type shims (Section 3.1); when a shim includes a property or method that is unrelated to the bug, we annotate it with `any`. For example, by using a shim, the code snippet

```
var t = {x: 0, z: 1};
t.x = t.y; // y does not exist on t
t.x = t.z; // z exists on t, but is unrelated
```

becomes

```
interface T {
  x:number;
  z:any; // z has the type any
}
var t: T = {x: 0, z: 1};
t.x = t.y;
t.x = t.z;
```

3.4 Results

The main results rest on a manual assessment of 400 buggy JavaScript programs. To help me calibrate, two collaborators on this experiment, Prof. Earl T. Barr and Dr. Christian Bird, and I jointly assessed a subset of the bugs. First, we present the inter-rater agreement of that three-way assessment, before presenting our main result that static type systems find a significant number of public bugs.

[8]We overload shim here, which traditionally means code that normalises the functionality of an existing API across different browsers.

3.4.1 Inter-Rater Agreement

To calculate inter-rater agreement, we uniformly selected 20 bugs for calibration, then all three of us annotated and classified each bug, using Procedure 1. Once all 20 bugs were processed, we collectively resolved each one on which they were not unanimous.

After this calibration step, we uniformly selected an additional subset of 80 buggy versions. Once we had independently classified each of the 80 bugs, we calculated the inter-rater agreement. There was full agreement among us for 86.4% of the issues, indicating a high level of agreement. Because there were more than two raters, Cohen's κ is not an appropriate statistic [194]. Instead, we use Gwet's AC_1 agreement coefficient, because it accommodates more than two raters and is more stable than Fleiss κ when the distribution of ratings is highly skewed (as in our case where over 80% of cases were rated as undetectable) [195, 196]. The AC_1 statistic for the 80 ratings by the three collaborators is 0.89 which indicates "almost perfect" agreement [197, 198]. In an effort to compare our ratings to a baseline rater that simply classifies each bug as undetectable (i.e. always choosing the majority class) we calculated the AC_1 statistic three times, each time replacing one of the collaborators with such a baseline rater. The resulting AC_1 statistics were statistically lower, 0.82, 0.85, and 0.83.

In discussing unknowns, we learned that each of us independently had categorised a bug as unknown when we thought it was detectable, but could not show it, before we ran out of time. To see the impact of this implicit agreement, we relabelled "unknown" as "detectable" and recomputed AC_1: it increased to 0.90; perfect agreement rose to 90%.

3.4.2 Detecting Public Bugs

Research Question: On what percentage of public bugs does Flow 0.30 or TypeScript 2.0 report errors?

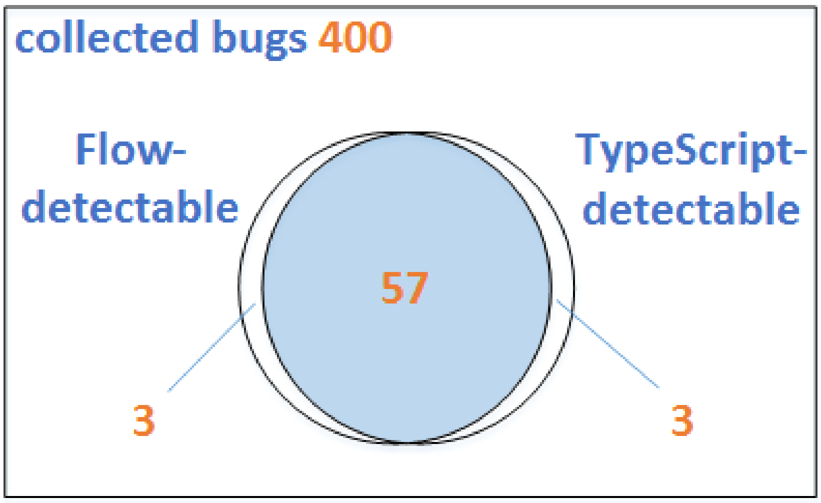


Figure 3.7: Venn Diagram of Flow- and TypeScript-detectable bugs.

Of the 400 public bugs we assessed, Flow successfully detects 59 and TypeScript 58. We, however, could not always decide whether a bug is *ts*-detectable within 10 minutes, leaving 18 unknown. The main obstacles we encountered during the assessment include complicated module dependencies, the lack of annotated interfaces for some modules, tangled fixes that prevented us from isolating the region of interest, and the general difficulty of program comprehension. For these 18 bugs, we spent as much time as needed to resolve each one. We patiently imported all relevant modules by using interface management tools like Typings[9], annotated interfaces as appropriate, and read the code base and official documentation when necessary. We used simple experiments to validate a *ts*-undetectable assessment, as necessary.

As a result, we successfully labelled all 400 bugs as either detectable or undetectable under Flow and TypeScript. Flow detected one more for a grand total of 60; TypeScript catches two more and also reaches 60. Running the binomial test on the results shows that, at the confidence level of 95%, the true percentage of detectable bugs for Flow and TypeScript falls into $[11.5\%, 18.5\%]$ with mean 15%. Figure 3.7 shows that Flow and TypeScript largely detect the same bugs. Section 3.5 describes the bugs on which they differ in detail. Together, Flow and TypeScript detect a total of 63 bugs, of which 7 (11%) are field bugs. This proportion of field bugs is

[9]`https://github.com/typings/typings`

approximate: to compute it, we manually counted *ts*-detectable bugs open across releases. Some projects do not tag releases; we conservatively deemed their bugs non-field. The time spent assessing each of the initially unknown 18 bugs varied, ranging from 8 minutes to more than 1 hour of dedicated time. Surprisingly, 3 bugs took us only around 10 minutes to decide their *ts*-detectability on a fresh restart, which, we reckon, is due to our increasing expertise.

Our experimental methodology and results extend previous efforts to measure the effectiveness of static typing, which have relied on programming assignments written by students [39] or have performed aggregate statistical analyses comparing two large disjoint sets, one composed of statically typed programs and the other dynamically typed programs [36, 37]. Our study complements these efforts by quantifying the bug-detection effectiveness of static types on bugs in real world projects on the same subject program. We have aimed to study the expressivity and power of type annotations, not the skill of the annotators. This is why we defined Procedure 1, defined and agreed the annotation tactics that III.D details, and compute the inter-rater agreement to measure the degree to which we have succeeded in consistently and uniformly devising and applying annotations. In this way, we have striven to emulate surgery trials, which seek to draw conclusions about surgeries, not the surgeons [177].

This result probably greatly understates the impact of static typing, since we designed our experiment from its inception to under-approximate the impact of static typing:

1. We study only publicly visible bugs. Anecdotally, static type systems eliminate many bugs during development and also obviate certain classes of testing. We do not measure either effect. Many public bugs are due to misunderstanding the specification, which type systems cannot detect.

2. Static type systems have other strengths, such as facilitating program understanding, improving performance, and enabling better code completion and navigation.

3. Our experiment uses only two relatively weak type systems, Flow and Type-

Script; stronger type systems could perform better.

4. Our limited expertise in Flow and TypeScript (and JavaScript) means that we may have incorrectly deemed a bug to be undetectable or unknown.

At first glance, 15% may not appear to be a large number. In practice, however, even small changes in the number of checked-in bugs can be quite valuable. When we presented the results to an engineering manager at Microsoft, he responded *"That's shocking. If you could make a change to the way we do development that would reduce the number of bugs being checked in by 10% or more overnight, that's a no-brainer. Unless it doubles development time or something, we'd do it."*. We have shown that Flow and TypeScript meet and exceed the 10% bar; we discuss the cost in our discussion of the annotation tax in Section 3.5.

3.5 Case Study

Based on three criteria, we select bugs for further manual assessment: ones whose Flow- or TypeScript-detectability is not agreed upon, ones whose Flow- and TypeScript-detectability differ, and ones that are TypeScript-detectable under version 2.0 but not under 1.8.

Disagreements Of the 80 uniformly-sampled bugs that we used to calculate inter-rater agreement, each rater needed to make 160 decisions in total, 80 for Flow-detectability and 80 for TypeScript-detectability. 138 of these 160 decisions were unanimously labelled. We define a *strong* disagreement as a disagreement in which one rater deems the bug detectable while another deems it undetectable. Of the 22 disagreements, 12 are strong.

Let U denote unknown, D detectable, and $\overline{D}$ undetectable. We manually assessed each disagreement without a time bound and found that, in each case, weak disagreements resolved as follows: $UUD \to D, UU\overline{D} \to \overline{D}, UDD \to D, U\overline{D}\overline{D} \to \overline{D}$. In other words, the rater who confidently assessed ts-(un)detectability within the time bound was correct every time in our experiment. Our 12 strong disagreements had three patterns of labels: 2 were $D\overline{D}U$, 2 were $\overline{D}\overline{D}D$, and 8 were $DD\overline{D}$. After manually resolving all of them, we found that whenever two raters agreed, they were correct. Among the 10 strong disagreements where a rater disagreed with the other two, rater one dissented in 8 cases and rater two in 2 cases. With hindsight, we would have improved our assessment protocol. We should have specified that each rater consider whether or not added logic was manual type checking. We would have agreed on whether or not to consider typos in library names *ts*-detectable. These changes alone would have eliminated 7 of the 12 strong disagreements. Please visit our project page for more details.

Classifying ts-undetectable Bugs Figure 3.8 categorises bugs that are undetecatble under both Flow and TypeScript, after the 18 unknowns were resolved. Recall that, while `BranchError`, `PredError`, and `URIError` are logic errors in implementing the specification, `SpecError` captures all other specification errors. Unsurprisingly, `SpecError`, with 186 bugs, accounts for 55% of the total bugs and significantly outweighs other categories. Errors implementing specification, as a group, overwhelmingly constitute 78%. This result, yet again, demonstrates the importance of specifications.

Despite the dominance of errors implementing specification and the fact that only public bugs are considered, there still exists a non-specification-related opportunity for type systems: `StringError`. Ranked second in the histogram, `StringError` is a broad concept that represents errors caused by the incorrect content of a string, such as a wrong URL. The reason why `StringError` is so common, we conjecture, is two-fold: first, the `string` type itself is extremely popular; second, JavaScript is rooted in web applications that extensively use hyperlinks. However, the `string` type is opaque to most static type systems, and how to effectively refine it remains challenging, although promising work is emerging in this direction [199].

Measuring TypeScript 2.0 null Handling Improvement TypeScript 2.0 was released during this study, giving us the opportunity to measure how effectively it handles `null` and `undefined`. Prior to 2.0, all types were nullable in TypeScript [200]. In Flow, all types, except `any`, `void`, and `null`, are non-nullable by default; one prefixes them with `?` to make them nullable. This design choice enables Flow to elegantly catch incorrect `null` / `undefined` usage. TypeScript 2.0 added the compiler

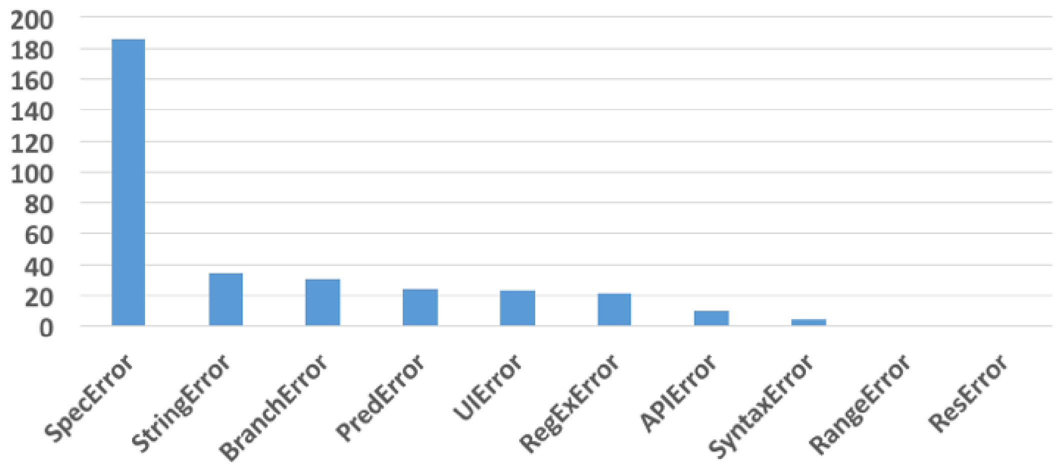


Figure 3.8: The histogram of undetectable public bugs under both Flow and TypeScript.

option `--strictNullChecks`, which, when enabled, makes most types nonnullable, allowing the user to **or** `null` into a type annotation to specify nullability. For instance, `var s: string | null = "foo"` defines `s` to be a nullable string.

We reviewed our corpus and found that 22 bugs, an increase of 58%, are detectable under TypeScript 2.0 but not under TypeScript 1.8. This result decisively and quantitatively demonstrates the value of TypeScript 2.0's strict null checking.

Comparing Flow and TypeScript Though sharing a similar annotation syntax, Flow and TypeScript differ in terms of expressivity and type variance. These dimensions are hard to quantify. Thus, we compare Flow and TypeScript in terms of their ability to detect and potentially prevent public bugs had they been used when those bugs were introduced and the costs of the requisite annotations.

As discussed in Section 3.4.2, Flow and TypeScript both catch a nontrivial portion of public bugs. In our dataset, the bugs they can detect largely overlap, with 6 exceptions: 3 bugs are only Flow-detectable and 3 only TypeScript-detectable.

All three Flow-detectable bugs share a common feature that reveals a weakness in TypeScript's recently introduced `null` handling: TypeScript does not error when concatenating a possibly `undefined` or `null` value to another of type `string`. For example, TypeScript remains silent on the following statement:

```
var x = " " + null + " ";
```

whereas Flow reports a type error:

```
1: ' ' + null + ' '
         ^^^^ null. This type cannot be added to
1: ' ' + null + ' '
   ^^^^^^^^^^ string
```

Without knowing whether TypeScript intentionally allows this behaviour, we cannot judge this decision, but its cost is substantial: TypeScript could have detected 3 more bugs, which amounts to an increase of around 5%.

Though bug `arrowrowe/es6-playground:2` is detectable under both Flow and TypeScript, it is worthy of attention. Originally, we reckoned that it was only Flow-detectable: Flow natively supports Node.js' `require()` function, which imports modules, and reports that the module named as a argument of `require()` does not exist; TypeScript lacks such support. The TypeScript team, however, helped us realise that, by using a TypeScript-specific module-importing syntax we had overlooked, `import foo = require("foo")`, this bug is, in fact, TypeScript-detectable. Similarly, we also thought that Flow had support for JavaScript's native functions, like `parseInt()` in `pupil-monitoring/pupil:14`. Here, the TypeScript team brought our attention to the `--noImplicitAny` option, with which enabled, TypeScript will error when it fails to infer a variable's type.

Two of the three bugs that are only TypeScript-detectable arise due to Flow's incomplete support for a popular JavaScript idiom, using a string literal as an index. For example, TypeScript detects the bug `conanbatt/OpenKaya:45` when `i0` and `i1`, two variables used as indexes, are annotated with `undefined | string | number`; Flow fails with the same annotation. The remaining bug, `sandeepmistry/node-core-bluetooth:1`, arises because of Flow's permissive handling of the `window` object. Below is its error message:

```
node-core-bluetooth/lib/central-manager-delegate.js:146
  }.bind(mapDelegate(self), mapPeripheral(identifier), error));
                     ^^^^
ReferenceError: self is not defined
```

In JavaScript, `self` generally refers to the global object, `window`. This bug is caused by a operating system upgrade, after which the system no longer recognises `self` and forces the developer to use `$self`. Therefore, the fix simply replaces `self` with `$self`.

Both Flow and TypeScript are able to infer that `self` has type `Window`. By reading the issue report and the code, we are able to infer that function `mapDelegate` accepts values of only `string` or `number` type. In TypeScript, we add the following annotation to `mapDelegate`'s definition:

```
function mapDelegate(self:string | number) {
```

Upon type checking, TypeScript signals a type error:

```
central-manager-delegate.ts(146,22): error TS2345: Argument of type 'Window'
    is not assignable to parameter of type 'string | number'.
```

Flow, on the other hand, even with the same annotation, does not regard `self` being passed to `mapDelegate` as a type error.

The per-Bug Annotation Tax Everything comes at a price. To enjoy the benefits that a static type system brings, a developer often needs to annotate their program. Directly measuring the effort programmers must expend to annotate their programs for a static type system would requires a large-scale, invasive study of two teams of developers, one using static types and other dynamic types, with all the attendant cost and confounds such a large user study would entail. Thus, we resort to under-approximating the annotation tax with two simple, expedient proxies: *token tax*, the number of tokens in the added type annotations, and *time tax*, the time spent adding annotations.

The token tax rests on the intuition that each token must be selected, so this proxy measures the number of decisions a programmer must make when adding type annotations. For a *ts*-detectable bug, we define the token tax as the number of tokens in the annotation needed to trigger a type error on a line involved in causing the bug. Let Δ be a function that returns the syntactical difference between two code snippets and $||\cdot||$ be a function that calculates the number of tokens in a code snippet. Then, the token tax is $||\Delta(a(b_i),b_i)||$. To report the time to annotate, we recorded how long we spent annotating each buggy version in the commit message, creating an electronic laboratory notebook [201].

These measures of the annotation tax are per-bug and underestimate the annotation effort in time and tokens relative to the whole code base, because our experiment leverages the fix to localise our annotation effort, as detailed in Section 3.2.1. With this knowledge, we locally annotate the region aimed at this specific bug, ignoring unrelated type errors. The developers who originally committed the buggy code lacked this knowledge and, in the worst case, may have needed to annotate the entire program. However, under the assumption that the project has fully embraced using a static type checker, the codebase would already be annotated prior to the bug-introducing change. Our annotation tax metric measures just the time and tokens required for the additional annotations at the time the bug-introducing change was made. Thus, our measure captures the case of incrementally adding and annotating patches to an already annotated code base. It is also likely that the project developers will be more knowledgeable about the codebase than us and take even less time to add the needed annotations.

Using this measure, we answer the question "What is the per-bug annotation tax in number of tokens of Flow and TypeScript, without considering the definition of shims?", finding that on average Flow requires 1.7 tokens to detect a bug and TypeScript 2.4. Two factors contribute to this discrepancy: first, Flow implements stronger type inference, mitigating its reliance on type annotations; second, Flow's syntax for nullable types is more compact. As discussed previously, to denote a variable is nullable in Flow, one simply needs to add a `?` before the type annotation, like `?number`, whereas TypeScript requires the use of union type operator, like `number | null | undefined`. The benefit of type inference in saving type annotations is also shown in the median values. Table 3.2 exhibits a sharper difference in time tax between Flow and TypeScript. Thanks to Flow's type inference, in many cases, we do not need to read the bug report and the fix in order to devise and add a consistent type annotation, which leads to the noticeable difference in annotation time.

Cross Pollination In our experiment and case studies, handling modules was the most time-consuming aspect of annotating buggy versions. Flow has builtin support for popular modules, like Node.js, so when a project used only those modules, Flow worked smoothly. Many projects, however, use unsupported modules. In these cases, we learned to greatly appreciate TypeScript community's DefinitelyTyped

	Token Tax		Time Tax (s)	
	Mean	Median	Mean	Median
Flow	1.7	2	231.4	133
TypeScript	2.4	2	306.8	262

Table 3.2: Under-approximation of the annotation tax in tokens and seconds for bugs detected by either Flow or TypeScript.

project. Flow would benefit from being able to use DefinitelyTyped; TypeScript would benefit from automatically importing popular DefinitelyTyped definitions. Flow would also benefit from supporting the use of string literals as array indices. TypeScript should borrow more null-handling tactics from Flow, as discussed above, like preventing the + operator from simultaneously taking null and string as operands.

3.6 Threats to Validity

To address the standard threat to external validity, we uniformly sampled issues from JavaScript projects on GitHub, a vast repository of software project and conservatively identified fixes (Section 3.3). Our experiment rests on public bugs, studies two static type systems for JavaScript, and relies on non-expert humans to annotate programs to determine whether Flow or TypeScript could have detected and prevented them. In each case, we have designed our experiment to under-approximate the effect of static type systems: since we always run the type checker on our annotated version of the buggy (Line 9 of Procedure 1), the only way we can generate a false positive and report a bug detectable when, in fact, it is not is if we fail to write consistent annotations (3.1.1). Private bugs occur in a buffer in a developer's editor, or survive to file saves or to commits to a local branch. Capturing these bugs is intrusive, requiring an instrumented IDE [202], so this work considers only public bugs, not private bugs. Type systems, however, can and do effectively detect and prevent private bugs. Because we have studied only Flow and TypeScript, our experiment reflects their strengths and weaknesses, and under-approximates the benefits of general type systems. We may have incorrectly determined a bug to be undetectable when Flow or TypeScript could have, in fact, reported an error given correct annotations and alerted a developer to prevent the bug. If we could not construct type annotations that caused the type system to report an error, we marked the bug as unknown. For each such bug, we made notes that we hope will be useful for the designers of type systems. While we are not experts in Flow or TypeScript, we have, through working on this project, become informed lay-people and therefore if we labelled a bug as unknown, it is a safe bet that many industrial practitioners would as well. We leveraged fixes to localise our annotation efforts; Section 3.2.1 details the attendant threats and limitations.

3.7 Chapter Summary

In this chapter, we evaluated the code quality benefits that static type systems provide to JavaScript codebases. The results are encouraging; we found that using Flow or TypeScript could have prevented 15% of the public bugs for public projects on GitHub. As far as we are aware, this is the first work to empirically evaluate the efficacy of static type systems for JavaScript on mature, real-world code bases. As such, our study will help practitioners decide whether to adopt a static type system for JavaScript by categorising bugs that Flow and TypeScript can and cannot detect at the time of this experiment and summarising differences between the two systems. All experimentation artefacts can be found at: `https://2-type-not-2.github.io/`.

Chapter 4

Retype: Changing Types to Change Behaviour

Chapter 3 empirically establishes that program transformation, in the form of optional type annotation, is effective in facilitating program analysis, in the form of static typing. This chapter takes a step further, exploring the possibility of designing a general, automated program transformation to extend the reach of existing program analysis techniques. Specifically, this chapter aims to facilitate symbolic execution. As Section 2.2.2 details, constraint solving for complex theories, like string and floating point, is still not scalable. The transformation presented in this chapter converts unsupported theories to supported theories, thereby sidestepping the problem.

Programmers often wish to widen, narrow, or change types within a specified region of source code. For example, narrowing to a subtype may improve the performance characteristics of the affected region, while widening to a supertype may afford greater precision to some specific computation. Type changes may be required to integrate different regions of code, each of which operates on shared data, but using different internal representations. In all three cases, the affected type must be safely converted at the boundary between the affected region in the context within which it resides.

As a motivating example, consider a financial legacy system that uses Binary Coded Decimal (BCD) and which is to be incorporated into an updated system that does not; a common reverse engineering scenario [203, 204, 205]. Financial business logic may be captured by a code region *A* that used BCD, which is to be integrated with a newer separate unit, *B* that uses integers, while both *A* and *B* reside in a wider context *C*, that uses integers. In order to compose *A* with *B*, the programmer retypes *B* to use BCD. This is a highly tedious task, because it involves identifying all points at which integers can flow into and out of BCD-computations, and applying the correct type-conversion operator at all such “type interfaces”. This is a task which clearly cannot be left to the type coercion system, since BCD encoding is simple invisible to the type coercion system. It is also an error-prone task, because any type-interface conversion overlooked by the programmer will likely compile, yet behave incorrectly. Furthermore, retyping introduces the need for type overloading, which when incorrectly applied by the programmer will also compile, yet may introduce potentially subtle bugs.

Retyping is thus currently a tiresome and error-prone manual process. Indeed, the burden of manual retyping maybe so great, that the programmer simply chooses to decline otherwise valuable opportunities for performance improvement and precision enhancement. In situations where retyping cannot be avoided, such as legacy system updating, the manual process is likely to be unnecessarily costly. Automated retyping avoids these unnecessary costs and missed opportunities.

More precisely, to “retype a program” means that the programmer identifies a specified region of interest with the program within which the following retyping activities are fully automated:

1. the replacement of occurrences of one type annotation with another,
2. the replacement of method calls that consume the old type, and
3. the conversion of the type and, if necessary, the values of affected variables that enter or exit the retyped region.

The rewritten program type checks by construction.

Programs cannot be entirely retyped, due to external libraries, operator definitions built into the language runtime, or system calls. It is often natural and convenient for the developer to reuse an operator name, like ’+’, when retyping. In this case, both the old and new definitions of an operator may be needed. Thus, retyping often overloads operator and function definitions. Further, data flows across the boundary between the retyped and unretyped regions must be handled. Both tasks are manual and error-prone, beyond the reach of regular expressions. RETYPE, our approach to retyping, handles both automatically: adding new operator definitions and replacing the operator applications that need the new definition and injecting type-conversions at the boundary between the retyped and unretyped regions.

Our approach to retyping extends both previous scientific research and practical manifestations of retyping in popular Integrated Development Environments (IDEs). Our theory and overall approach follows the seminal work of Balaban et al. [206, 207, 208], who introduced an elegant framework for replacing a derived Java class with another that shares a common ancestor in the Java inheritance hierarchy. This previous work focused on retyping Java, based on a “migration specification” that defines the mapping between the method calls and field accesses in the two derived classes. Balaban *et al.* also extend the type constraint system of Schwartzbach and Palsberg [209] to type check the retyped program. Furthermore, since their work is specific to Java, they are also able to handle Java synchronisation seamlessly within their approach. We extend this previous work to retype programs written in non-Objective-Oriented programming languages, handling type overloading and the instantiation of type-conversion operators at the boundary between the retyped code region and the context within which it resides.

Our tool, TYPERITE, extends the practical manifestations of retyping found in the two popular IDEs, Eclipse and IntelliJ, each of which partially automates a special case of retyping for Java: Eclipse supports the replacement of a Java type with its supertype, while IntelliJ can retype a single variable at a time (but fails to change the operator applications into which the new type flows, neither does it handle constants relying, instead, on the underlying builtin type coercion mechanism. TYPERITE removes reliance on type coercion, generalises to reasonably complex, statically typed programming languages, and automates the remaining manual retyping, otherwise left to the programmer by these existing IDE implementations. When the programmer fails to supply all operators needed to retyped the program, TYPERITE automatically identifies those that remain to be supplied. When TYPERITE reports no such error, the resulting program type-checks by construction. Unlike previous approaches and implementations, TYPERITE can replace a type with a subtype in languages that do not support subtyping, like C.

We comprehensively evaluate TYPERITE in three different scenarios, each of which involve application to real-world systems. Specifically, we apply TYPERITE to 15 different retyping tasks in the instant messaging system `Pidgin`[1], a 363K LOC open source system, with several million users worldwide. We also compose two applications of TYPERITE to two C and three Java programs, migrating them to a new type and then back again. This demonstrates that this composite retyping operation, which is idempotent in theory, is also idempotent in practice, for our implementation. Finally, we illustrate an application of TYPERITE to the task of integrating two different data type representations, by retyping between fixed and floating point arithmetic.

The main contributions of this chapter follow:

1. We introduce and formalise retyping a program. This formalism
   (a) injects the type-conversions that arise due to interactions with code that is not retyped;
   (b) allows the controlled introduction of new behaviour through the new, target type; and
   (c) applies to both non-Object-Oriented languages and primitive types because it uses term rewriting, not the subtype relation.

2. We present a realisation of RETYPE in the tool TYPERITE and validate it:
   (a) demonstrating how it can be used to catch real world bugs, like an SQL injection in `hibernate`, and automates taint analysis on implicit information flow;

[1]`https://pidgin.im`

(b) showing that TYPERITE scales to Pidgin, an open-source online chatting client with 363K lines of code.

(c) automating retyping, eliminating the need for a developer to assess whether 120,096 operators need replacement or conversion when retyping float to fixed point in the special functions of the GNU scientific library.

4.1 Motivating Example

When a developer replaces a normal pointer with a reference counting one to improve the effectiveness of garbage collection, they are retyping their programs. When a developer replaces `double` with `float` to trade precision with performance, with new operators that guard against potential underflows and overflows, they are retyping their programs. Indeed, bespoke attempts to semi-automate retyping are common. Ariadne [156], a symbolic execution engine that automatically finds exception-triggering inputs for floating-point programs, extensively uses a tailored, float to arbitrary precision rational, retyping. SEEDS [210] is a general framework that aims to reduce a program's energy usage. The authors instantiate SEEDS by retyping a subclass of Java's Collection Interface to the most energy-efficient implementation.

Many embedded systems lack a Floating-Point Unit (FPU). TYPERITE eases the porting of a floating-point program to the domain of embedded systems, by automatically replacing a floating-point type with other real encodings that embedded systems support. The GNU Scientific Library (GSL) provides a wide range of mathematical routines that largely rests on floating-point arithmetic. Without loss of generality, Figure 4.2 shows a code snippet, adapted from `airy_der.c` (line 742 – 747) in GSL. We demonstrate how TYPERITE makes the program applicable to embedded systems without a FPU, by retyping `double` into `fixed`, a fixed point encoding[2]. Figure 4.1 defines the operators over `fixed` required by the retyping, abridging the actual implementation.

```
$ typerite airy_der.c float fixed.c
```

One might think that retyping is just a glorified find-and-replace, for which

[2]https://github.com/kmowery/libfixedtimefixedpoint

```
fixed fix_mul(fixed x, fixed y) {···};
fixed fix_mul_d(double x, fixed y) {···};
fixed convertToFixed(double x) {···};
```

Figure 4.1: The definitions of fixed's operators in a file fixed.c.

```
742 const double atr = 8.7506905708484;
743 const double x2 = x*x;
744 const double sqrtx = sqrt(x);
745 ...
746 const double z = 2.0*x...
747 cheb_eval_mode_e(&bif2_cs, z, mode, &result_c0);
```

Figure 4.2: Input code for TYPERITE in `GSL specfunction`.

regular expressions are sufficiently useful. However, most popular programming languages are at least context free, beyond the reach of regular expressions. Specifically, for two reasons a regular expression based find-and-replace cannot perform the retype operation illustrated above. First, functions, such as `cheb_eval_mode_e()` at line `747`, and literal constants, such as `"8.7506905708484"`, are defined outside the retypeable region and are therefore unretypeable. TYPERITE injects a type-conversion operator that converts between objects of the original and new type to reconcile the conflict. Regular expressions, however, cannot decide where to apply the type-conversion operator. Second, regular expressions cannot handle operator overloading that naturally arises when the retypeable region is not the whole program. For example, after the operands have been retyped, the operator $*$ is called with two different type signatures. At line `743`, $*$ should take two fixed as arguments; at line `746`, it should take a double and a fixed. For these two call sites, regular expressions cannot identify which new operator to use.

When retyping, a developer may overlook some operators to which the new type flows. To ease the developer's burden, TYPERITE implements an useful error mechanism. For example, if the developer forgets to provide a new operator for `sqrt` and the type-conversion operator, TYPERITE reports an error:

```
ERROR: "sqrt", application "1" not handled!
APPLICATION: sqrt(x)
LOCATION: airy_der.c, line 744
```

Reacting to this message, the developer can provide a new operator `fix_sqrt` that

```
742   const fixed atr = convertToFixed(8.7506905708484);
743   const fixed x2 = fix_mul(x,x);
744   const fixed sqrtx = fix_sqrt(x);
745   ...
746   const fixed z = fix_mul_d(2.0,x)...
747       cheb_eval_mode_e(&bif2_cs, convertToDouble(z), mode, &result_c0);
```

Figure 4.3: etyped code for Figure 4.2. Type annotations and operator applications modified by TYPERITE appear in bold.

takes a `fixed` as input and returns a `fixed`. Assuming the developer provides all the operators for fixed in Figure 4.1 and `fix_sqrt`, the right-hand-side of the assignment at line `746` in fact has two possible rewrites, `fix_sqrt(x)` or `sqr(convertToFixed(x))`. TYPERITE's rewriting, however, is two staged (Section 4.2.5); that is, it first applies the available new operators, and then, as a last resort, injects the type-conversion to preserve type-correctness. Therefore, TYPERITE transforms the input program into one shown in Figure 4.3. As GSL is composed in `c`, this example also demonstrates that TYPERITE is applicable in non-Object-Oriented languages.

4.2 Formalism

This section opens with definitions, then presents the RETYPE's core algorithm. This algorithm is applied to a running example, with its critical steps discussed in detail in the following subsections.

4.2.1 Definitions

Figure 4.4 defines the core language that RETYPE targets. Each variable has a type, which is bound to the variable at its declaration. The language allows developers to construct customised types using keyword `struct`. From the core syntax, we compose a toy program, shown in Figure 4.5. Suppose we wish to retype int to TInt defined in Figure 4.6, whose operators include two new pluses with different type signatures, and a conversion that takes a TInt and returns an int. In the input program, function read() and foo(), whose definitions are not in the retyping region, cannot be retyped, as the grey background indicates. We use the running example to illustrate our formalism throughout this section.

We now define a type context, and illustrate how RETYPE generates the type

$$\begin{array}{rrl}
\text{Program} & P & ::= dcl^{*}\ e \\
\text{Expression} & e & ::= \textbf{skip} \mid n \mid b \mid l \\
 & & \mid x \mid x = e \mid f(e^{*}) \\
 & & \mid \textbf{ret}\ e \mid e_0 ; e_1 \\
 & & \mid e_0 \oplus_a e_1 \mid e_0 \oplus_b e_1 \\
 & & \mid \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \\
 & & \mid \textbf{while}\ e_0\ \textbf{do}\ e_1 \\
\text{Declaration} & dcl & ::= fndcl \mid vardcl \mid stdcl \\
\text{Function Declaration} & fndcl & ::= t\ f(vardcl^{*})\ \{vardcl^{*}\ e\} \\
\text{Struct Declaration} & stdcl & ::= \textbf{struct}\ sn\ \{vardcl^{*}\} \\
\text{Variable Declaration} & vardcl & ::= t\ x \\
\text{Type} & t & ::= sn \mid \textbf{int} \mid \textbf{void} \mid \textbf{bool} \\
 & & \mid \textbf{float}
\end{array}$$

$$\begin{array}{rlrl}
n & \in \mathbb{Z} & b & \in \mathbb{B} \\
l & \in \text{locations} & sn & \in \text{struct names} \\
x & \in \text{variable names} & f & \in \text{function names} \\
\oplus_a & \in \text{arithmetic operations} & \oplus_b & \in \text{boolean operations}
\end{array}$$

Figure 4.4: The core syntax of a language. RETYPE rewrites declarations, operator uses and function calls.

context for the running example.

Definition 4.2.1. *A **type context** Γ is a finite function from statements in S (*i.e.* terms) to types and locations:*

$$\Gamma = \ \{(x_1 : \tau_1, l_1), \cdots, (x_n : \tau_n, l_n)\},$$

where $x_i \in S$, $\tau_i \in t$, and $l_n \in \mathbb{N}$.

A type context contains definitions and function calls. Definitions include function and variable definitions. To rewrite type annotations and calls in line 2 in Algorithm 2, we adapted the standard definition of the type context [211] to include

```
void read(int var)

int foo(){
    int i;
    i = 3;
    ret i * i * i;
}
void main (){
    int a b c;
    a = 11;
    read(b);
    b = a + foo();
    c = a + 1;
    b = b + c;
}
```

Figure 4.5: Input program of the running example in RETYPE's core language.

```
Struct TInt {
    int i_val;
    bool b_val;
}
TInt opPlus(TInt x, TInt y) {···};
TInt opPlus(TInt x, int y) {···};
int convertToInt(TInt x) {···};
```

Figure 4.6: The definition of τ_n in our running example, including its type and operators.

the location of the term, using path, file, line number as our coordinates, as l_i. We use the location information in error messages and to identify where to rewrite the input program to produce the retyped program (Section 4.2.6).

4.2.2 The RETYPE Algorithm

Algorithm 2 defines RETYPE's highlevel algorithm. RETYPE takes, as input, P the program to retype, τ_o the 'old' type to replace, and τ_n the 'new' type with which to replace τ_o.

RETYPE also takes two functions. The function $r : S \rightarrow \mathcal{B}$ specifies which $t \in P$ should be retyped and $m : F \rightarrow 2^F$ maps methods in τ_o to methods in τ_n; it returns a subset because τ_n may overload the function in P. The function r defaults to all terms internal to P; m defaults to the identity function. The new type τ_n must be a subtype of τ_o with respect to τ_o's retypeable uses in P. 4.2.2 relaxes this constraint. In the case of non-OOP languages, τ_o and τ_n share definitions: they are not encapsulated. Thus, we do not introduce classes in non-OOP languages. Both τ_o and τ_n denote both a type name and its definition; we specify which when it is not clear from context.

First, Algorithm 2 (line 1) infers Γ_o, the type context in P. On line 2, it applies the r on Γ_o, and obtains Γ_o', the subset of Γ_o to contain only retypeable terms. Further, on line 3, RETYPE applies Φ, RETYPE's term rewriting system to retype Γ_o' to Γ_o''. Finally, on line 4, Algorithm 2 checks the correctness of the retyping. First, we

Algorithm 2 $P' = \textsc{Retype}(P, \tau_o, \tau_n, r = * \to \mathbf{T}, m = id)$

```
Input: $P$, the input program
 1:     $\tau_o$, the old type to replace
 2:     $\tau_n$, the new type
 3:     $r$ identifies retypeable terms
 4:     $m$ maps names in $\tau_o$ to $\tau_n$
 5: $\Gamma_o := infer(P)$                                ▷ Section 4.2.3
 6: $\Gamma'_o := r(\Gamma_o)$                            ▷ Section 4.2.4
 7: $\Gamma''_o := \Phi(\Gamma'_o)$                       ▷ Section 4.2.5
 8: $P' := tce(P, m(\Gamma''_o, infer(\tau_n))$           ▷ Section 4.2.6
```

map the old operator names to the new ones by using m, and next we check if the definitions in τ_n match the applications in Γ''_o. If this is true, we return the retyped program.

4.2.3 Type Inference

To retype a program, Retype must infer the type context Γ_o from P and from τ_n. Retyping parts of a program overloads an operator when that operator's old definition is used outside the retyped region. For example, at line 13 : c = a + 1; and 14 : b = b + c; in Figure 4.5, the operator '+' has two call sites. The first call has an unretypeable operand, the constant '1', while both operands are retypeable in the second one. Under these conditions, Retype infers two different types for + in P', as the column Γ' shows for P', in Figure 4.8: $TInt \to TInt \to TInt$, and $TInt \to int \to TInt$.

For Figure 4.5, we identify all the usages of variables and operators involving `int` in the input program. Further, we extract the type context Γ for the identified usages. τ_n contains the definitions for `TInt`'s operators shown in Figure 4.6.

Blindly adding the new definition into an existing type context would produce:

$$\{opPlus : (TInt \to TInt \to TInt),$$
$$opPlus : (TInt \to int \to TInt), \cdots\}.$$

This multiset is *not* a type context because it contradicts 4.2.1, since `op2` appears in two different assignments.

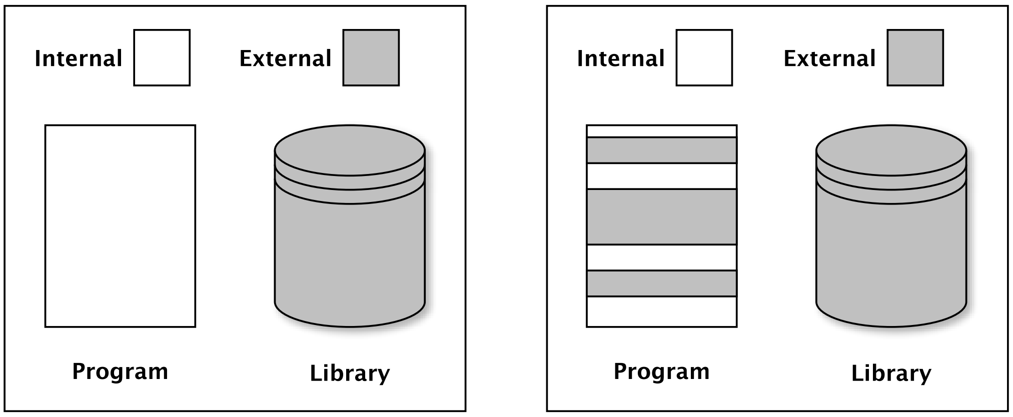


Figure 4.7: Code that is retypeable (white) and not (gray).

To solve this problem, we can use fresh names or Intersection types (∩-types) [82]. Fresh names introduce a new variable name, and thus a new operator definition, for each overload of an operator. ∩-types, on the other hand, allow a single symbol to simultaneously have multiple types. We choose ∩-types over fresh names, because ∩-types produce more compact formulae in the original namespace, which, when there is an error in retyping, generates more understandable error messages.

Inference over ∩-types is undecidable in general, because ∩-types assign the same type to β–equivalent terms, but the β–equivalence of two terms is undecidable [212]. Retype escapes this undecidability result by inferring ∩-types only over programs that type check in the base language's type system, which does not use ∩-types and is in β normal form. Essentially, RETYPE puts two type systems in sequence.

4.2.4 Retypeable Region

A program is a sequence of statements in S, or of terms, in the sense of lambda calculus [211], that are either internal and mutable, or external and immutable, like constants, or system calls or third party library calls. External terms cannot be retyped. By default, all internal terms are retypeable. Figure 4.7 shows this. In the leftmost figure, the entire input program is retypeable, while the library is not. In the figure on the right, only portions of the input program are retypeable (the white ones) while the library remains unretypeable.

When retyping a program, developers will often want to specify which internal terms can and cannot be retyped, *i.e.* explicitly externalise some internal terms. For instance, they may wish to retype some performance critical functions or just a block to which they have tracked down a bug. To control which internal terms are retyped, RETYPE allows its user to define $r : S \to \mathcal{B}$, which marks some internal terms as *retypeable*, implicitly externalizing the rest. Thus, a term can be naturally external, like a system call, or externalised by a developer overriding the default definition of r. In Figure 4.5, the `read()` call on Line 11 is a system call and naturally external; the developer externalised `foo`, as its grey background denotes, so its call on Line 12 cannot be replaced.

4.2.5 Rewriting Type Contexts

During execution, data flows across the border between the retyped and unretypeable portions of the input program. Some of these data flows may involve τ_n, as when a system call returns an instance of τ_o to a variable retyped to τ_n or an instance of τ_n is passed as a parameter to a function call that the developer specified should not be retyped. In these cases, the flow must be converted: *e.g.*, 12 : b = a + foo(); External functions like library and system calls are one source of these terms; regions the developer decided not to retype are the other.

Here, we present Φ, the rewriting rules to accomplish this task. RETYPE leverages the fact that, of the myriad of types, Φ's rewriting rules need to consider only two, constant types: τ_o and τ_n. Thus, Φ is a ground term rewriting system over two constants.

τ_n must define type-conversions, so that Φ can inject them at the appropriate points. Φ prefers operators in τ_n to those in τ_o, with type-conversions. Only when not a suitable replacement is found, Φ uses operators in τ_o with casts. Casts relabel a variable; they are a substitution operator on type annotations. Casting is insufficient for RETYPE, since the narrowing, widening, or reencoding that it supports may require arbitrary computation, subject to the restrictions defined in Section 4.2.7. Consider for example extracting values from an ADT in a downcast in C++: class `Dog` is derived from class `Animal`; a downcast in C++ takes an `Animal` object, and converts it into `Dog`. Thus, our type-conversions, which we denote $\leftarrowtail$, are, in general, Turing-complete; they must also relabel:

$$\leftarrowtail(x) = \begin{cases} \tau_o & \text{if } x = \tau_n \\ \tau_n & \text{if } x = \tau_o \\ x & \text{otherwise} \end{cases}$$

Because $\leftarrowtail$ relabels as shown, it subsumes casting. The point of retyping is to introduce new operators over τ_n. Type-conversion make a trivial sort of retyping possible that does not use operators over τ_n, but instead converts all instances of τ_n to τ_o and only applies operators over τ_o. Thus, type-conversions are a last resort. They are necessary for handling constants, as shown in . Figure 4.5. The constant `1` at line `13` is inside a retypeable region, but cannot be retyped since it is a constant, so we use type-conversion operator: `convertToTInt(1)` to convert it to *TInt*.

Φ handles externals interactions by applying type-conversions to terms at the boundary between retypeable and unretypeable terms. Given the term $t \in S$, the *external*?(t) selects constants and external calls, but does not match user-supplied operators.

$$R_1 = \tau_o \longrightarrow \leftarrowtail(\tau_o) \text{ if } \mathit{external?}(t) \wedge t : \tau_o$$
$$R_2 = \tau_n \longrightarrow \leftarrowtail(\tau_n) \text{ if } \mathit{external?}(t) \wedge t : \tau_n$$

To illustrate how Φ works, we continue using the `+` operator in our running example, which appears both in Figure 4.5 and the upper left corner of Figure 4.8. In Figure 4.8, when the type-conversion $\leftarrowtail$ is needed, $c(\tau)$ denotes the new type constructed by $\leftarrowtail$. The box Γ_o'' in Figure 4.8 shows the result of applying Φ on Γ_o', the type context of the filtered terms in P. R_1 introduced the $\uplus$ types, while R_2 introduced the conversions c in Figure 4.8. Note that we do not apply Φ on Γ_n'', the type context of τ_n.

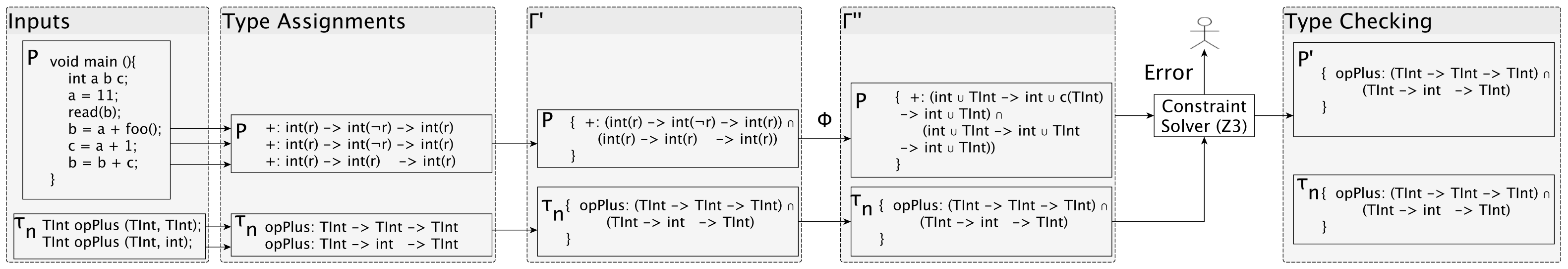

Inputs
P void main (){
int a b c;
a = 11;
read(b);
b = a + foo();
c = a + 1;
b = b + c;
}
τn TInt opPlus (TInt, TInt);
TInt opPlus (TInt, int);
Type Assignments
P +: int(r) -> int(¬r) -> int(r)
+: int(r) -> int(¬r) -> int(r)
+: int(r) -> int(r) -> int(r)
τn opPlus: TInt -> TInt -> TInt
opPlus: TInt -> int -> TInt
Γ'
P { +: (int(r) -> int(¬r) -> int(r)) ∩
(int(r) -> int(r) -> int(r))
}
τn { opPlus: (TInt -> TInt -> TInt) ∩
(TInt -> int -> TInt)
}
Φ
Γ''
P { +: (int ∪ TInt -> int ∪ c(TInt)
-> int ∪ TInt) ∩
(int ∪ TInt -> int ∪ TInt
-> int ∪ TInt))
}
τn { opPlus: (TInt -> TInt -> TInt) ∩
(TInt -> int -> TInt)
}
Error
Constraint
Solver (Z3)
Type Checking
P' { opPlus: (TInt -> TInt -> TInt) ∩
(TInt -> int -> TInt)
}
τn { opPlus: (TInt -> TInt -> TInt) ∩
(TInt -> int -> TInt)
}

Figure 4.8: Retyping the example program in the core language : get type context, extract the type assignments, construct Γ, and apply Φ.

4.2.6 Type Checking

The type context formed from merging the rewritten type context Φ produces and τ_n's may not type check for two reasons: 1) missing definitions in τ_n and 2) overload conflicts. An overload conflict occurs when two or more overloads of an operator have the same type signature, and can be used in an operator application. Then, *tce* cannot decide which definition to use for that operator application.

RETYPE uses constraint solving for type checking. One of the reasons for using constraint solving is that this allow us to provide precise error messages. Another the reason is that the fresh names generate constraints that grows multiplicatively in the number of applications, and operators, while $\cap$-types is linear in the number of operators. While type checking '+" in our running example in Figure 4.5, RETYPE with fresh name would call an SMT solver four times for the uses of '+' on lines 13 and 14. Using $\cap$-types allows RETYPE to call the solver only twice.

For each retypeable $t \in \Gamma''_o$, RETYPE calls the constraint solvers to check if the operators in τ_n contain a unique solution for the constraints generated from t. If none of them is the solution, we apply the type-conversion operators.

Constraint solving can be very slow. RETYPE mitigates this problem by generating very simple constraints. $\cap$-types distribute over $\rightarrow$. This fact permits rewriting the type assignments over one parameter of a function at a time. For example, `opPlus` in Figure 4.6: $opPlus : (TInt \rightarrow TInt \rightarrow TInt) \cap (TInt \rightarrow int \rightarrow TInt)$, can be rewritten as (op_1, op_2), and its return (ret)): $(op_1 : TInt) \rightarrow (op_2 : int \cap op_2 : TInt) \rightarrow (ret : TInt)$.

We consider only a variable whose initial type is τ_o, so the only pertinent question is whether it has been retyped or not. We map the retyped ones to `T`, and the others to `F`. Under this encoding, R_1. introduces a constraint that is always true: $T \vee F$. Thus, our constraints are just a sequence of conjunctions over the scope of a variable declaration and over function applications.

Coming back to the `opPlus` in Figure 4.6, we consider the constraints generated for its second parameter here. In the input program, it is called at the lines: 13 : c = a + 1; and 14 : b = b + c. In the call on line `13`, the second parameter is not retypeable, but *TInt* defines the type-conversion, so we map it to `T`. On `14`, it is retypeable, so we map it to `T`. This means that the type of the second parameter of the operator `+` is *TInt* in the retyped program. Thus, the constraint generated for these two usages is: $T \wedge T$. This is simplifies to T, and so the second parameter of the new operator `+` should be of type *TInt*.

RETYPE identifies a missing operator definition when *tce* cannot find a definition in τ_n that consumes the types of an operator application in the input program. This search always succeeds when τ_n defines type-conversion operators, since they can always be applied, to each parameter and return value. For example, we have shown that the solution to `opPlus`'s constant in Figure 4.6 for its second parameter is T, so its type must be $TInt$. Similarly, the solution for the types of its first parameter and return value are also T. For the retype to succeed, τ_n must define *opPlus* with the type signature: *TInt* $\rightarrow$ *TInt* $\rightarrow$ *TInt*. If this is not present, *tce* reports a missing operator.

Merging Namespaces P and τ_n may not use the same names for equivalent operators. Their namespaces must be normalised and merged. The map m parameter of Algorithm 2 connects terms in the input program P to their replacement in τ_n. In our running example, m maps `+` from P to `opPlus` in τ_n.

Emitting the Retyped Program When rewriting P to P' in line `3`, Algorithm 2, RETYPE treats variable definitions and function definitions differently. Specifically, it changes the annotations in variable definitions, but injects the operator definitions in τ_n at the end of the file, since the old function definition might still be used by unretyped parts of the input program. RETYPE adds new function definitions to the end of the source files, so that l remains identical between the input program and the retyped one.

4.2.7 Semantics-Preservation under RETYPE

Retyping injects new behaviour into the retyped program by the means of the new operator definitions in τ_n. The relationship between the program's original and retyped behaviour depends on these definitions, whose semantics RETYPE does not constrain. When a user retypes an operator to one that crashes the program, for instance, RETYPE inevitably fails to preserve the original behaviour. However, when the users constrain the new behaviour of the operators in τ_n, we can prove semantics-preservation properties. First, we consider the case where τ_n adds operator definitions that introduce and operate only on fresh state, then we consider the case where the new operators change the program's original behaviour, but maintain a user-defined predicate over the original program's traces and those of its retyped incarnation.

For the first case, we use the Liskov substitution principle (LSP) [213]. Unlike nominal and structural subtyping, behavioural subtyping is defined on semantics instead of syntax and guarantees semantic interoperability of types in a hierarchy. Informally, LSP states that if S is a behavioural subtype of T, then objects of type T in a program may be replaced with objects of type S without affecting any provable property over objects of type T. Under LSP, retyping a variable from S to T preserves behaviour.

Let $\Sigma = X_0 \times X_1 \times \cdots \times X_k$ be the state space of the program P and $\Sigma' = \Sigma \times X_{k+1} \times \cdots X_j$ be the state space of a retyped variant of P. For $\vec{\sigma} \in \Sigma$ and $\vec{\sigma}' \in \Sigma'$, $=_\Sigma$ is program state equality restricted to Σ: $\vec{\sigma} =_\Sigma \vec{\sigma}' \equiv \bigwedge_{\sigma_i \in \vec{\sigma}} \sigma_i = \sigma_i'$. Let $trace(f(i)) = \sigma_0, \sigma_1, \cdots$ be the states generated by f's execution on i. Let ψ be a predicate over traces that filters out states internal to retyped operators, producing the traces $t = \sigma_0, \cdots, \sigma_k$ and $t' = \sigma_0', \cdots, \sigma_j'$, then returns $i = j \wedge \bigwedge_{\sigma_i \in t} \sigma_i =_\Sigma \sigma_i'$.

Theorem 4.2.1 (Semantics-Preservation under the Liskov Substitution Principle)**.** *If τ_n is a subtype of τ_o under the Liskov substitution principle, then* $\forall i \in \mathbf{dom}(P)$, $\psi(trace(P(i)),$
$trace(\text{RETYPE}(P, \tau_o, \tau_n, m)(i)))$.

Proof Idea LSP implies that any new behaviour τ_n defines operates on fresh data, not Σ. Noninterference (NI) partitions a program's inputs and outputs into low or high [214]. A program exhibits NI when changes in its high inputs do not change its low outputs. To apply NI to 4.2.1, we map the program's old state to low variables, and the new, "fresh" state, introduced by τ_n to high variables. We then prove noninterference by induction over applications of the new operators introduced by τ_n. In applying induction, we follow Focardi *et al.* in establishing step-wise NI, a stronger property that trivially implies NI [215].

In the base case, we show that all transitions from initial states in the retyped program are noninterferencing with respect to "fresh" state. We do so by showing that high states cannot flow into low ones across any operator in τ_n. This must hold under LSP, because, if not, failure for NI to hold is a provable property that would distinguish the retyped program's state space from the original program's state space when objects of type S replace objects of type T. Next, we assume that the noninterference holds over k applications of the new operators, and prove, again under LSP, that it holds for any $k+1$ application. Thus, no high or "fresh" state affects any low state: all evaluations outside of the retyped operators are low, or, in our case, original state equivalent. In particular, the evaluation of control point expressions produce low state, so control flow outside of retyped operators is unchanged.

The LSP assumption handles naturally arising retype use cases, like taint analysis, refactoring, replacing an array with a list, logging, or instability detection in floating-point programs [216]. It is, however, too strong in many practical cases. Sometimes, we wish to retype a program to change its behaviour. For example, a developer may change `float` to `double` to increase a computation's accuracy. Under this retyping, the retyped program may take different execution paths on the same input, as when the original program exited upon overflowing a variable.

To handle these cases, we relax 4.2.1 in two ways. First, we allow the user to specify ψ to capture those semantics of the original program that τ_n preserves. Second, we only require ψ to hold over a non-empty subset $\hat{I} \subseteq \mathbf{dom}(P)$. When retyping `float` to `double`, a developer could define ψ to tolerate a bounded error, thus implicitly handling rounding errors (INEXACT exceptions) up to its bound. Consider Ariadne [156], which retypes a numerical program's floating-point variables to arbitrary precision rationals, reifies floating-point expressions to program points, then asks whether these program points are reachable. To use RETYPE, Ariadne would define ψ to match control expression evaluations and to ignore other program

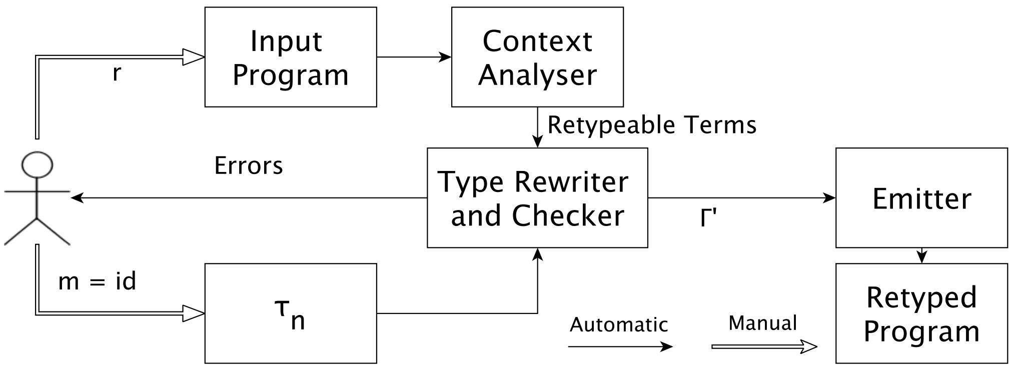


Figure 4.9: The Architecture of TYPERITE.

states except the final states of the two traces, which must be the same floating-point exception.

Theorem 4.2.2 (Semantics-Preservation modulo ψ). *If τ_n is a subtype of τ_o modulo ψ, then $\forall t \in \hat{I}$, $\psi(trace(P(i)), trace($
$retype(P, \tau_o, \tau_n, m)(i)))$.*

Proof Idea We use induction over operator applications, using case analysis to show that ψ holds over the τ_n's new operators at the base case and the inductive step.

4.3 Implementation

We realise our approach to retyping in TYPERITE, composed of a Context Analyser, a Type Rewriter and Checker, and an Emitter, shown in Figure 4.9. The **Context Analyser** infers the type context Γ (Section 4.2.3), augmented with locations in P later used in the Emitter, and implements r to identify the retypeable region within P to produce Γ' (Section 4.2.4). The **Type Rewriter and Checker** rewrites Γ' (Section 4.2.5) and type checks it, via constraint solving using Z3[3]. Last, the **Emitter** takes Γ' and rewrites P's operator applications and variable definitions to form P'.

TYPERITE is built on ROSE [217], a compiler infrastructure on which developers can perform customised source-to-source transformation, analysis, and

[3]`https://z3.codeplex.com`

optimisation. TYPERITE currently supports C and C++. One should be able to add support for statically-typed languages that ROSE supports, such as Java and Fortran, to TYPERITE with ease. Each component in Figure 4.9 is in fact a ROSE-based compiler.

Two alternatives to ROSE we considered are TXL [218] and LLVM[4]. We decided against TXL because ROSE scales much better and because ROSE transforms a program's AST so its rewritings are language agnostic, while TXL requires transformation rules for each language. Like ROSE, LLVM scales to industrial programs; unlike ROSE, LLVM transforms a program to its intermediate representation, so it is language agnostic too. At the time of implementing TYPERITE, however, LLVM did not provide an easy way to output the rewritten source code from the rewritten intermediate representation. One of our design goals for TYPERITE is to provide the retyped source code to the user so they could verify the change and check it into version control. Thus, we choose ROSE over LLVM. TYPERITE needs rewrite and output headers. Header handling is not among ROSE's most heavily used features, so we encountered some bugs, such as relative path handling in `#include` and the failure to emit header files.

Specifying What to Retype TYPERITE permits developers to explicitly specify which program terms to retype (and which not). The developer can pass TYPERITE a file containing lists of the names of functions or files. When defining the retypeable region using functions, TYPERITE allows the user to specify whether or not to follow calls, and to what depth. For a depth `k`, TYPERITE retypes all the functions up depth `k` depth in the call graph of the program, rooted in the current function; depth `*` means an infinite depth. For example, '`foo 1`' indicates that `foo` and its direct callees be retyped. When the user names a file, TYPERITE simply treats it as the set of functions it defines.

Within a file, the developer can also annotate sequences of lines or even annotate snippets within a line. These annotations delimit the region to retype. Conceptually, they are not in the alphabet of programming language. In practice, we realise them

[4]`http://llvm.org/`

using a unique sequence of characters, whose exact definition is configurable. Here, we use `[*` and `*]`. To annotate a line, one can write

$[*\ \mathrm{w} = \mathrm{a} + \mathit{foo}();\ *]$.

To annotate variables inline, one can write

int $[*\ \mathrm{a}\ *]$,b,w,x,$[*\ \ \mathrm{y}\ \ *]$, z;.

Usage To use TYPERITE from the command line, one issues either

```
$ typerite <switches> P.c float fixed.c # OR
$ typerite      Makefile    float fixed.c
```

In the first mode, TYPERITE takes compiler switches, and then accepts a source file as a program, the type to replace, here `float`, and the name of file, here `fixed.c`. In the second mode, TYPERITE accepts a `Makefile`. The filename `fixed.c` represents τ_n. This file must compile, and start with the definition of τ_n, followed by the definitions of the new operators and type-conversions. TYPERITE accepts two optional arguments: the filename to contain the mappings m; and the filename to contain the retypeable regions r.

Error Handling TYPERITE can fail for two reasons: 1) missing operator definitions; and 2) colliding overloads. Colliding overloads refers to overloads that share the same type signature, so TYPERITE cannot decide which one to choose. When operators are missing, the developer might wish to get the list of missing operators or learn how often the missing operators are used, to decide whether to provide a definition for a given operator or just use type-conversion. Assume the operators `+` and `sqrt` are undefined. To list the missing operators, the developer issues

```
$ typerite P float fixed.c --missing
ERROR: "op+": fixed x fixed -> fixed
ERROR: sqrt: fixed -> fixed
```

To list all uses of missing operators, the developer issues

```
$ typerite P float fixed.c --unhandled
ERROR: "op+":
1) APP: a+b not handled!; foo.c:23
2) APP: x+5 not handled!; bar.c:2343
```

If two user provided definitions for replacing the operator `+` collide, TYPERITE reports

```
ERROR: COLLISION: "plus_1"; "plus_2":
1) APP: a+b collides.; foo.c:23
```

To enhance TYPERITE's deployability and ease reproducing its development environment, we utilise Docker[5], which virtualises operating systems, to encapsulate TYPERITE. We provide source code, binaries, and dockerised releases of TYPERITE as open source at `http://crest.cs.ucl.ac.uk/retype/`.

4.4 Evaluation

Our evaluation first establishes that TYPERITE works, that it changes a program's type annotations without otherwise disrupting that program. Then we demonstrate TYPERITE's utility, by using it to fix field bugs and automate the retyping tasks inherent to taint analysis and refinement types. We validate that TYPERITE preserves behaviour when LSP holds, like replacing `int` with `TaintedInt`, and changes behaviour modulo some predicate specified by the user when LSP does not hold, like replacing `double` with `fixed`. Finally, we show TYPERITE scales to `Pidgin`, a program with 363K lines of code. Note that external, immutable definitions are the fundamental challenge of TYPERITE, and the number of injected type-conversion operators measures the flow across the retypeable boundary.

4.4.1 Sanity Check

If TYPERITE is correct, TYPERITE(TYPERITE(P, τ_o, τ_n), τ_n, τ_o) (*i.e.* $f \circ f^-1$, for $f =$ TYPERITE), a roundtrip application of TYPERITE, should return a program that is semantically equivalent to the input program, when τ_n is a subtype of τ_o under the Liskov substitution principle.

A problem naturally arises: if some terms already use τ_n in a retypeable region of the original program, the second retyping process might incorrectly change them. Similar to α-renaming [60], TYPERITE solves this problem by always using fresh names for τ_n. That is, if τ_n is ever used in the input program, TYPERITE will generate another unique name for τ_n using the renaming construct available in ROSE.

In the sanity check, we use testing to under-approximate the programs' seman-

[5]`https://www.docker.com`

Project	**Statement Coverage**	**Retyping Successful?**	**Sanity Check?**
Commons Email 1.4	69%	Yes	Yes
Commons Validator 1.4.4	83%	Yes	Yes
fwknop 2.6.6	90%	Yes	Yes
Joda Time 2.1	90%	Yes	Yes
OpenSSH 6.6	55%	Yes	Yes

Table 4.1: The sanity check results. *Retyping Successful?* shows if TypeRite successfully changed the types twice; *Sanity Check?* reports if the retyped programs passed the tests.

tics. We select five open source programs that have test suites with relatively high statement coverage, 77.4% on average, two written in C and three in Java. For each program, we build two sets of types and randomly select a pair (τ_o, τ_n). We apply TypeRite without defining r that specifies retypeable terms, first changing τ_o to τ_n and then τ_n to τ_o. We observe that all five transformed programs pass all the tests. Table 4.1 reports the results of this sanity check.

4.4.2 The Utility of TypeRite

As a general tool, TypeRite helps semi-automate many analysis techniques, such as taint analysis and refinement types, and has huge potential in fixing bugs.

Taint analysis Taint analysis [165] marks (taints) values from a source, tracks the propagation of these marks, and checks whether the values reach a sink. It shines in software security, but also shows its strength in testing and debugging. A user wishing to use TypeRite to inject taint tracking types would have to supply a `struct` type that includes an additional taint field and corresponding operators. They need to identify a retypeable region contains a source and sink of interest. TypeRite would automatically instrument the program and handle explicit information flow (data dependence). TypeRite could also automate taint analysis on implicit information flow (control dependence) in a language whose control constructs, like `if`, were functions, amenable to redefinition. Note that the tainted type only introduces new behaviour, and is a structural subtype of the old type under LSP. Therefore, the original behaviour, after retyping, should remain intact.

```
int main(){
    int a,b,w,x,y,z;
    a = readInt(); // readInt reads an integer from stdin
    b = readInt();
    w = a * 2;
    x = b * 1;
    y = w + 1;
    z = x + y;
}
```

(a) *P*, the code to retype.

```
typedef struct {
    int value; bool tainted;
} t_int;
```

(b) The new, target type.

```
t_int op+(t_int op1, t_int op2){
    return {op1.value + op2.value, op1.tainted || op2.tainted};
}
t_int constructTaintedInt(int val){
    return {val, false};
}
```

(c) Illustrative example of the user provided operators.

```
int⁶ main(){
    t_int a, b, w, x, y, z;
    a = constructTaintedInt(readInt());
    b = constructTaintedInt(readInt());
    w = a * constructTaintedInt(2);
    x = b * constructTaintedInt(1);
    y = w + constructTaintedInt(1);
    z = x + y;
}
```

(d) The retyped program.

Figure 4.10: An example of taint analysis on explicit information flow.

Figure 4.10a adapts a taint analysis example from a highly-cited paper on dynamic taint analysis which introduces a general framework for previous ad-hoc techniques and considers both control and data flow [219, Section 2, Figure 1]. To semi-automate taint analysis on this program, we replace `int` with `t_int` defined in Figure 4.10b. Figure 4.10c presents a complete definition of the operator `+` on `t_int` and a constructor. TYPERITE successfully transforms the input program, shown in Figure 4.10d. Variables `a` and `b` store values provided by the user who may be malicious, so these variables are taint sources. We add two lines to the retyped program to taint them: `a.tainted = true;`, and `b.tainted = true;`. Assuming the user enters `11` and `5`, we run the retyped program and output variables `w`, `x`, `y` and `z`: `w = {22,`

```java
public class SecuredString {
    String string_;
    public SecuredString(String s) {
        if(s.contains('\'')) this.string_ = "";
        else this.string_ = s;
    }
    public String opPlus(String leftOperand) {
        return leftOperand + this.string_;
    }
    public String toString() {
        return this.string_;
    }
}
```

Figure 4.11: Definition of `SecuredString`, to replace `String`.

`true}; x = {22, true}; y = {5, true}; z = {28, true}`. As expected, all four variables are tainted, while their actual values remain the same as those in the original program.

Refinement Types Refinement types enrich a coarse-grained type lattice and can be used to prevent real world bugs. For example, an ordinary string may flow into a variable intended to store passwords which have certain patterns; an arbitrary integer may flow into a variable used as the index of an array, which causes an error out-of-bound array access. TYPERITE facilitates the implementation of refinement types.

`Hibernate`, a Java framework used in many open source projects [7], enables developers to easily write applications whose data outlives the application process. `Hibernate 2.1.4` contains a vulnerability in `SessionImpl.find()`, which generates an SQL query directly from its argument of type `String` without validation. This vulnerability permits an adversary to bypass authentication by concatenating `'OR 1 = 1'` to an account selection query:

```
SELECT * FROM accounts WHERE name = 'legit_user' AND password = '';
```

To fix this kind of security vulnerability, one might use TYPERITE to refine a normal `String` into a safer version, which does not allow SQL separators. Figure 4.11 shows an example of such a refinement, `SecuredString`. The `SecuredString`'s constructor does not allow the special character for SQL command `'`. Whenever this is received, the `SecuredString` holds an empty `String`, and thus, the risky API in `hibernate` cannot receive an SQL command.

[7] https://goo.gl/mBhWg1 and http://snipsnap.org

Type of Changes	Occurrences
Type Annotations	67
Type-Constants	37
Function Call Replacement	39
Initializations	1

Table 4.2: Changes that TYPERITE does for fixing security vulnerabilities.

We apply TYPERITE to `personalblog 1.2.6`, an application which uses hibernate and inherits this vulnerability. Retyping the whole program is unnecessary, because only the strings provided by the user that flow into a SQL query are of interest. We used context filtering for discretionary deciding which terms to retype, and which not. The call places of the vulnerable `hibernate Session.find()` method are only in the class `PersonalBlogService`. We provided the name of the file containing this class to TYPERITE, together with the complete definition of `SecuredString`, part of which depicted in Figure 4.11. TYPERITE successfully retyped `PersonalBlogService`, and made a total of 144 modifications, shown in Table 4.2, 37 of which are type-conversions that measures the flow across the retypeable boundary.

Bug fixing Table 4.3 reports three field bugs that can be fixed by using TYPERITE. Time stamps in Unix, whose type, `time_r`, is a signed 32-bit integer, are represented by the number of seconds since 1st January 1970. From the year of 2038, the number of seconds will exceed the largest value that a signed 32-bit integer can represent: $2^{31} - 1$, and therefore causes an integer overflow. Manually modifying all the `time_r` structures in Unix source code is a tedious and error-prone task. One can use TYPERITE for doing this.

`Limbus`[8] is a popular open source library for the Meter-Bus protocol, which standardise the data from electricity meters, heat meters, gas meter, *etc.*. The users of `limbus` reported the Y2K bug of issue number 75[9]. `Limbus` encodes a year in only two digits and calculates the current year by simply adding 2000 to it. Therefore, if one uses `limbus` on historical data, before the year of 2000, the bug will manifest. By using TYPERITE, we replaced the current data type that holds a two-digit year

[8]https://github.com/rscada/libmbus
[9]https://github.com/rscada/libmbus/issues/75

Program	Bug Description	Bug Fixed
`limbus`	Year 2000[10]– 2 decimals	YES
`Unix`	Year 2038 – integer overflow	YES
`finger`	Morris Worm	YES

Table 4.3: Field bugs that can be fixed by using TYPERITE.

`struct tm` into a new one, to allow four digit years.

Morris worm [220] is the first computer worm distributed through the internet, and thus the first internet attack. It exploited a set of vulnerabilities in different Unix utilities. Here we concentrate on `Finger`, a tool that is used for showing the users logged on the network. The version of `Finger` contained at line `50` the code: `gets(line)`. Here `finger` reads the request with `gets()`, an insecure standard C library routine. This method does not check for the overflow of the string variable `line`. Thus, Morris worm, sends to finger a request bigger than the allocated space for `line`. The code that overflows gets executed, and causes the spreading and execution of the Morris worm. By using TYPERITE, we retype `char *` into `SecuredString` in file `fingerd.c`:

```
struct SecuredString(){
    char * string;
    int length;
}
```

Note that we do not provide the type-conversion operator, for identifying all the insecure usages of the strings in `fingerd.c`. TYPERITE reports an error:

```
ERROR: "gets", application "1" not handled!
APPLICATION: gets(line)
LOCATION: fingerd.c, line 50
```

Thus, the definition of the operator `gets()` does not exist for `SecuredString`. The developer figures out that this operator is a big security vulnerability, and fix the bug.

4.4.3 Changing Behaviour

TYPERITE can change the program behaviour, when τ_n is not a behavioural subtype of τ_o under LSP. In this case, TYPERITE allows the user to specify a predicate over semantics of the original program that τ_n preserves, and guarantees that the

[10]`https://goo.gl/U37IB7`

difference is modulo the predicate. GNU Scientific Library (GSL) extensively uses floating-point arithmetic. We apply TYPERITE to programs from the `specfunction` component of GSL, changing `double` to `fixed` discussed in Section 4.1. We show that, on GSL's test suite, the difference between outputs before and after retyping is within some error threshold.

By retyping `specfunction`, TYPERITE successfully modified 9,489 type annotations, and 120,096 operator applications, indicating that the flow across the boundary of the retypeable region was significant. Of the 120,096 modifications on operator applications, 45,066 used type-conversions, and 75,030 used operator overloadings, of which 2,272 were applied to retypeable operators and 72,758 were applied to unretypeable operators. In terms of lines of code, TYPERITE modifies 23,379 lines, out of a total of 61,378 lines of code as reported by `cloc`[11]; that is, 38% of lines of code are changed. Manually retyping these many lines requires a considerable amount of effort.

We then ran 2,426 accompanying test cases on programs from `specfunc` and the retyped versions in parallel. Instead of generating random inputs, we rested on test cases, which specify representative, valid inputs that the users may provide or typical invalid inputs that the programs should fail. Of the 2,426 test cases executed on the retyped programs, 274 (11%) failed due to overflows or underflows, another 1,000 (41%) failed because the outputs of the retyped programs fell outside the error tolerance defined in the test cases, and the remaining 1,152 (47%) passed. For the 1,000 failed test cases, whose behaviour TYPERITE modified, the output difference between the original and retyped programs has the mean value of $2.502 \times e^{-6}$ and the maximum value of $1.131 \times e^{-3}$. In summary, the differences are relatively small, which demonstrates that TYPERITE preserves the original behaviour modulo a user-defined predicate, here an error threshold.

4.4.4 TYPERITE's Scalability

To demonstrate the scalability of TYPERITE, we apply it to `Pidgin`, an open-source online chatting (instant messaging) client which contains 363K lines of code. We

[11] `http://cloc.sourceforge.net/`

Category	**Whole Program**		**One Variable**	
	Tests Passed	# Modifications	Test Passed	# Modifications
Prim → Prim	100%	225	100%	7
Prim → ADT	100%	439	100%	16
ADT → Prim	100%	439	100%	16
ADT → ADT	100%	173	100%	4

Table 4.4: Scalability and effectiveness example for TYPERITE, applied to globally retype all 363K line of code of Pidgin, a real world instant messaging system that has several million users worldwide. *Category* shows the categories of τ_o, and τ_n. For example retyping from primitive to Abstract Data Types (ADT) is written as: 'Prim → ADT'.

perform four different sets of experiments on this real world system, converting between primitive types and abstract data types.

Table 4.4 presents the results for these four categories of retypes. The first row's category is *Prim → Prim*. The experiments in this category use TYPERITE to change an initial primitive data type, to a new primitive data type, that is guaranteed to respect LSP. Here we had five pairs of initial (τ_o), and new (τ_n) types, for example `int` → `long`. The second category is *Prim → ADT*, which consists of five specific retypings, involving insertion of a taint analysis type, for example, `double` → `TaintedDouble`. The third category is *ADT → Prim*, which implements the inverse retyping operations to those applied in the second category. For the final set of scalability experiments, *ADT → ADT*, we randomly selected five *ADT*s defined in `Pidgin`, for which we provided five new semantically equivalent *ADT*s definitions, into which each is retyped.

For each of these four categories of retyping operation, we report the percentage of passed test cases, and the number of modifications applied. Of course, we expect all test cases the pass, because we have proved that our retyping operation is safe (4.2.1 guarantees semantics-preservation under LSP). However, this set of large-scale real-world experiments provide an additional empirical validation that TYPERITE faithfully implements the retyping approach for which this proof obligation was discharged; as Knuth famously remarked "Beware of bugs in the above code; I have only proved it correct, not tried it." [221].

	Retypeable Specification	**IJ**	**R**
1	Granularity of Variable	Yes	Yes
2	Granularity of Type	No	Yes
3	Granularity of Project	Yes	Yes
4	Granularity of File(s)	Yes	Yes
5	Granularity of Function(s)	No	Yes
6	Granularity of Line(s)	No	Yes

	Handled Program Constructs	**IJ**	**R**
1	Literal Constants	No	Yes
2	Constructors	No	Yes

	Error Handling	**IJ**	**R**
1	Error Location	No	Yes
2	Error Reason	No	Yes
3	Call Site Sensitive	No	Yes

	Supported PLs	**IJ**	**R**
1	OO: Java	Yes	Yes
2	Procedural: C	No	Yes
3	Multi-Paradigm: C++	No	Yes

	$\tau_o \rightarrow \tau_n$**relation**	**IJ**	**R**
1	Explicit Subtype with Inheritance	Yes	Yes
2	Subtype without Inheritance	No	Yes
3	$name(\tau_o) \neq name(\tau_n)$	Yes	Yes
4	Different Method Names	No	Yes
5	Different Field Names	No	Yes

	Unretypeable Calls	**IJ**	**R**
1	Convert Incoming Values	No	Yes
2	Convert Outcoming Values	No	Yes

Table 4.5: The results of the comparison between TYPERITE (R) and the type migration feature in a popular IDE: IntelliJ IDEA (IJ).

4.4.5 Qualitative Comparison

Programmers' desire to change a type has captured industry's attention, reflected by the fact that two popular IDEs have built-in, partial support for this feature. Eclipse[12] implements *Change Method Signature*, allowing one to change the types of a method's parameters or return value. For this retyping to succeed, Eclipse must check whether the method's new type signature matches its call sites and whether the parameters' new types match their uses in the method body. If the new types are explicit subtypes of the old ones, these checks pass and Eclipse replaces only the type annotation in the method's signature. Otherwise Eclipse reports an error. Similarly, Eclipse supports *Use Supertype Where Possible*.

IntelliJ IDEA[13] implements type migration, capable of changing the type of a variable or a method by performing inter-procedural data flow analysis[14]. To qualitatively compare TYPERITE against IntelliJ IDEA, both tools were given the same retyping task (*e.g.* `int` to `TaintedInt`), using the same Java program, shown in Table 4.5. For each of the features in the second column, we specified the same type change operation for both TYPERITE and IntelliJ. We used IntelliJ's GUI for calling Type Migration, and report as unhandled by IntelliJ each case when the Type Migration window misses our desired Type Migration operation (as handled by TYPERITE), or when the resulting post-migration code contains an error. The

[12]`https://goo.gl/77KkCt`

[13]`https://www.jetbrains.com/idea/`

[14]`https://goo.gl/zvGivR` and `https://goo.gl/aO4qBk`

code was simple enough that we were able to manually check its correctness. For TYPERITE we used the command line interface for specifying the type change operations, as presented in Section 4.1. The comparison is over six dimensions, each of which appears in bold below.

IntelliJ allows the user to **specify retypeable terms** at the granularities of variable, file, and project; TYPERITE provides two more, line and function. Changing a certain type, for all the variables in the specified regions, is possible only in TYPERITE. IntelliJ reports an error when the region to retype contains some **program constructs**, such as constants and constructors; TYPERITE uses type-conversion to handle constants, and treats constructors as normal operators. IntelliJ is an IDE for Java and supports only the OO paradigm; TYPERITE transcends the OO paradigm and the **language support** of its current implementation matches that of ROSE. IntelliJ requires τ_n to be inherited from τ_o using the keyword `extends`; TYPERITE, with an optional mapping m, does not constrain as strictly **the relation of τ_o and τ_n** and is able to retype cases like subtyping without `extends` or a pair of types with different method or filed names[15]. IntelliJ fails to provide any useful **error information** when a retype operation is faulty, and it cannot identify errors raised when one retypes the arguments or returns of functions at a non-retypeable call site; TYPERITE, whose type inference uses constraint solving, explains the causes of an error and its exact location in the original program, and TYPERITE is call site sensitive. IntelliJ does not permit retyping for terms that **flow into or out of retypeable regions**; TYPERITE uses type-conversions to handle these terms.

ReplaceClass In 1998, JDK 1.2 introduced `ArrayList`, an unsynchronised list, to complement `Vector`, a synchronised list. Developers who did not need synchronisation and preferred the performance of `ArrayList` were faced with the task of manually retyping their programs to replace uses of `Vector` with `ArrayList`. To automate this tedious task, Balaban *et al.* proposed a refactoring technique, Class Migration [206], later ReplaceClass [208]. ReplaceClass takes, as input, the program to refactor and a migration specification defined by the user, a set of rewriting rules from field accesses

[15]For cases handled by IntelliJ or nominal subtyping, TYPERITE does not require m.

and method calls on the original class to those of its replacement. ReplaceClass assumes that the original and its replacement are semantically equivalent, at least over the operations in the migration specification. To ensure type correctness and behaviour-preservation, ReplaceClass uses an extension of type constraints [209]. ReplaceClass is conservative, so it does not replace the type of variables that can be bound to values of the old type, as when the variable flows into or out of a library call. ReplaceClass preserves synchronisation using escape analysis [222]. Following in Balaban *et al.*'s footsteps, TYPERITE generalises ReplaceClass along three dimensions: TYPERITE 1) supports retyping non-Object-Oriented languages and primitive types; 2) converts values that flow between the old and new type; and 3) allows introducing bounded changes to the original program's behaviour, as shown in Section 4.4.3 where TYPERITE retypes the GSL's special functions from float to fixed point.

4.5 Chapter Summary

We have defined retyping, a process for changing program behaviour under the control of a type system. To retype a program is, in a user-specified region, to replace occurrences of one type annotation with another, replace operator applications that involve the new type, and convert the type and if necessary, the values of affected variables that flow across the retypeable boundary. We have presented a type inference engine and an accompanying term rewriting system that formalises retyping. We have implemented TYPERITE, a tool for automatically retyping programs and validated it on real-world programs. It is our hope that TYPERITE will facilitate research by eliminating the need to reinvent the wheel by implementing bespoke, error-prone retypings.

Chapter 5

ValueScope: Tracking Values to Their Origin

Program transformation can facilitate program analysis in various ways. While Chapter 4 explores how type-based program transformation extends the reach of existing program analysis techniques, this chapter uses program transformation to implement a specific debugging technique.

Debugging reprises the scientific method: developers observe a failure, posit a hypothesis for the failure, add or modify code to test that hypothesis, then execute the modified code. Developers explore many hypotheses while looking for the cause of a failure, which, according to early study [223], accounts for 90% of the debugging effort. Some of these hypotheses concern the origin of unexpected values, like a division-by-zero when a variable `y` is bound to `0` in `x/y`. Was `y` bound to `0` here because of the evaluation of an erroneous expression or did the program read this binding from a database? Indeed, tracking down a value's origin is common in debugging. At the time of writing, a search on Stack Overflow with the keywords "track down `null`" returns 377 results that have been viewed more than 1,000 times.

In some cases, the origin of an observed, erroneous value v is buggy and causes a failure, like `null`'s causing a `null` dereference. In other cases, however, the bug may be executed well before when v is produced, and is actually related to other *unobserved* values that contribute to v's production. Consider an out-of-bound array access in `x = arr[i]` in an unguarded language, like C, where `arr`'s length is `4` but `i` is bound to `5`. Often, a developer can observe only the unexpected resulting value stored in `x`, like a freed pointer, manifesting itself later in the execution. In fact, it is the unobserved, out-of-bound index value `5` that causes the failure. Therefore, tracking down `5`'s origin is essential to localising the dangling pointer.

To capture both cases, we generalise the origin of a value from the expression that produces it to a set of expressions that "contribute" to its production. Section 5.1 defines a value's origin and Section 5.3 explains how to identify these expressions. Then, the *value origin problem* is to find the origin of a value from a program location where the value manifests (Section 5.1). In generalising the atomic debugging tasking of tracking the life cycle of a value, this problem concerns the general problem of program understanding. The most naïve, yet common, approach to tackle the value origin problem involves repeatedly inserting a `print` statement, which is tedious.

We present VALUESCOPE, a novel value-sensitive, backward program analysis to automatically solve the value origin problem. Given a value of interest that is observed to manifest at an point in a program execution, VALUESCOPE realises a conceptual reverse execution to track the value from its manifestation through its propagation to its origin (Section 5.2). While researchers have worked on dynamically finding the origin of a value of interest [158, 159], VALUESCOPE is the first to track arbitrary, user-specified primitive values, including `0`, `null`, and `Inf`. Therefore, it transcends debugging and aids general program understanding.

To maximise VALUESCOPE's utility, we have designed it to track a value of interest through computations that produce it from other values. These new values become values of interest on which VALUESCOPE restarts (Algorithm 3). Consider the out-of-bound array access `x = arr[i]` again. VALUESCOPE stopping its backtracking for the value in `x` at this assignment may only trigger the developer to ask "Where does the erroneous value in `i` come from?". Automatically connecting the propagation chains of the value in `x` and the value in `i` helps developer more quickly determine the true cause of the out-of-bound array access.

Blindly tracking all values used in the computation that produces the value of interest is infeasible, because of the exponential time and space complexity.

Instead, VALUESCOPE ramifies a propagation chain by analysing which value to variable bindings are "relevant" to the computation of a value of interest at its birthplace[1]. For example, assume evaluating the expression `x + 0 * y` returns 3; only `x` is *relevant* to producing 3. VALUESCOPE introduces *relevance analysis* (Section 5.3) to capture only variables that make "sufficient contribution" to the computation of an expression. Given an expression e in a program execution, relevance analysis quantifies the impact of the free variables that e involves and selects those variables whose relevance is above a threshold. Then, the values bound to these relevant variables become the new values on which VALUESCOPE recurs. Relevance analysis can dynamically and automatically generate new values of interest, thereby granting VALUESCOPE the support for multi-value tracking and an open vocabulary of values.

We implement VALUESCOPE via program transformation (Section 5.4) and target it for JavaScript, a prevalent language that encodes all numeric values in floating point and therefore is particularly vulnerable to special values like `Inf`. To demonstrate its utility in bug localisation, we compare VALUESCOPE with two baselines: an effective spectrum-based fault localiser that uses the DStar ranking measure [11], and ORBS [12], a program slicer which keeps deleting statements until the observable behaviour for a given slicing criterion is no longer preserved. We apply these three techniques to 44 real-world JavaScript programs with value-related bugs and their fixes. To report meaningful results, we introduce the "on screen" measure, which determines whether l_c, any line a technique identifies when attempting to localise a bug, and l_f, any line touched by the bug's fix, can appear on the same screen, when l_c is at the middle of the screen. We demonstrate that a line VALUESCOPE reports is, on average, 12 lines away from the fix, 42 lines closer than that of DStar, while the total number of lines VALUESCOPE reports is on average 12.5% of that of ORBS. (Section 5.5).

In summary, we have made the following contributions:

- We identify the value origin problem, which generalises the heretofore un-

[1]Though all parameters are created equal, some are more equal than others (according to some sources, George Orwell said something similar).

named debugging task of tracking values, like `NaN`, to their origin.

- We define parameter relevance to quantify the impact of a function's parameters (or an expression's free variable) on that function's output.
- We present VALUESCOPE, a tool which automatically solves the value origin problem for arbitrary primitive values in JavaScript.
- We evaluate VALUESCOPE's effectiveness in localising real-world bugs against two baselines: DStar, a state-of-the-art spectrum-based fault localiser, and ORBS, a program slicer, and show that a line VALUESCOPE reports is, on average, 12 lines away from the fix, 42 lines closer than that of DStar, while the total number of lines VALUESCOPE reports is on average 12.5% of that of ORBS.

5.1 Problem Definition

When debugging an erroneous value v in a program execution, a developer often first tracks v to where it was produced and then ponders what went wrong. Sometimes, the developer blocks the value's propagation to fix the bug. Other times, the developer fixes the bug at v's expression of birth. Still other times, the bug lies "deeper": at the origin of or propagation from the values from which the program computes v. When this happens, the developer restarts the origin search on those values, as when tracking a zero to the expression `a + b` in an execution that binds `a` to 5 and `b` to −5. In a case like this, the question immediately becomes "Where did these two bindings originate?".

To handle these cases, we generalise the origin of the value v to the set of expressions that originate the values from which a program produces v. Let E denote the set of expressions in the host language that can produce v. For some expressions in E, like the example in the last paragraph, a developer would restart the value origin search, on new values if necessary. We call these restart expressions, denoted by E_R. Nonrestart expressions, $E_N = E \setminus E_R$, are those that the developer considers an origin of v and does not search further. Three notable disjoint subsets of E_N are environmental interactions, like prompting a user, external library calls, and literals (which are externally defined).

Definition 5.1.1 (A Value's Originating Expressions). *The set of* originating expressions of the value *v is*

$$O(v) = \begin{cases} \{e\} & \textit{if } e \in E_N \\ \bigcup_{v' \in R(e)} O(v') & \textit{otherwise } (e \in E_R), \end{cases}$$

where $R(e)$ extracts values from the bindings of the free variables in the restartable expression e on which to restart the origin search. Section 5.3 describes how we realise E_R and R.

The *value origin problem* is to take a trace and a value of interest at a observation point in the trace and find that value's originating expressions and value-flows from those expressions to the observation point. This problem captures the first critical step of debugging an erroneous value. It is not limited to primitive values: it can search for the origin of the values of arbitrary types. Any user-defined datatype can be extended with a primitive field, like an int, and a transformation that ensures that the new primitive field changes whenever any other field of the datatype changes.

For some values of interest, such as `NaN` [2], the value origin problem can be challenging. Even without a restart, `NaN` is problematic, because any floating point operations involving `NaN` produce `NaN`, making the task of tracking down its actual origin time-consuming. The top question on StackOverflow about debugging a `NaN` has been viewed more than 20,000 times and a user responded, "it's much tougher to pinpoint where the nans and infs may occur than to fix the bug" [224]. Popular libraries, like Tensorflow, have even added debugging support for `NaN`, which, however, is mostly limited to checking whether an object contains a `NaN`.

Figure 5.1 presents a real-world JavaScript program that mishandles `NaN`. A user reported that the property `percentageDifference` of an element in the array `toParse`

[2] Short for Not-a-Number, `NaN` is the special value that represents the result of an undefined operation in floating point arithmetic, like `0/0` or `sqrt(-1)`.

```
87  // Calculate the percentage of this one trade
88  toParse[i].percentageDifference = percentageString(toParse[i
       ].currentValueOfAll, toParse[i].invested);

    ...

201 function percentageString(newValue, oldValue) {
202     let percentage = "";
203     if (newValue >= oldValue) {
204         percentage = "+" + roundNumber((newValue - oldValue)
                / oldValue * 100) + "%";
205     } else {
206         percentage = "-" + roundNumber((oldValue - newValue)
                / oldValue * 100) + "%";
207     }
208     return percentage;
209 }
```

Figure 5.1: The code snippet from file `index.js` in a GitHub repository `CryptoSummary` at commit `ef24e35`. A user reports that this program generates a `NaN` observed at line 88.

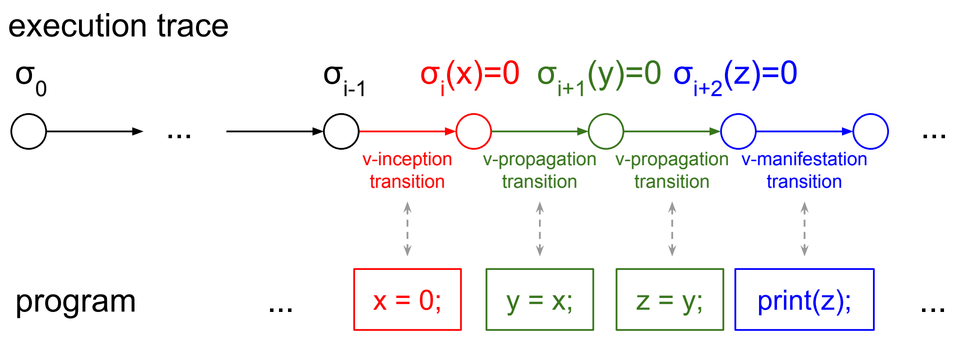
could have the unexpected value `"+NaN%"`. The error was that both parameters of the function `percentageString` were bound to `0` at the right-hand-side of the assignment on line 88. Therefore, the expression `(newValue - oldValue) / oldValue` on line 204 evaluates to `NaN`. The production of `NaN` is 116 lines away from its observed location, which hinders debugging.

5.2 Discovering Value Origins

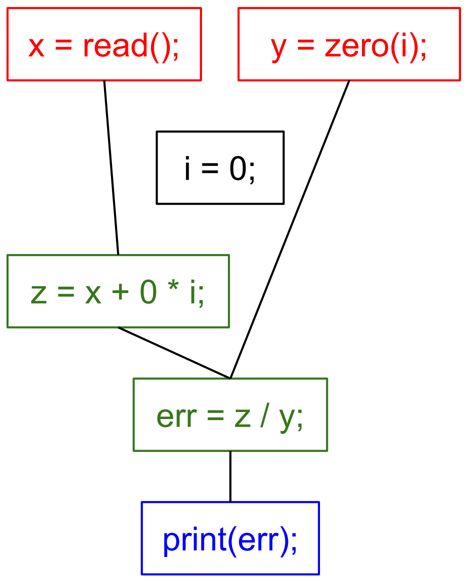
We present VALUESCOPE to automatically tackle the value origin problem. At its core, VALUESCOPE realises a reverse execution that proceeds backwards from a value's observation point to the origin of that value. This section formally describes it by first introducing the models and the notations.

For presentation purposes, we assume we have the complete execution trace of a program; Section 5.4 lifts this constraint. Formally, VALUESCOPE operates on an execution trace[3]; it projects out state transitions that generate, propagate, or observe a value of interest. To explain this projection, we model a program $\mathbb{P}$ as a Labelled Transition System (LTS) [225]. A LTS is a triple $\langle \Sigma, A, \longrightarrow \rangle$, where Σ is

[3]Section 5.4 discusses how we implement VALUESCOPE in a tool that operates on source code.



(a) A c-chain presents a straightforward, linear view of the life cycle of a value of interest.



(b) A c-tree connects multiple c-chains.

Figure 5.2: Data structures central to VALUESCOPE.

the set of $\mathbb{P}$'s *states* that bind memory locations to values, A is the set of $\mathbb{P}'s$ *actions* (*i.e.* statements), and $\longrightarrow \subseteq \Sigma \times A \times \Sigma$ is the *transition relation*. For simplicity, we generalise *variables* to mean memory locations and denote them by X. We denote a state transition with $t_i \in \longrightarrow$, where $t_i = \langle \sigma_{i-1}, \alpha_i, \sigma_i \rangle$. Let $S \subset \Sigma$ be the set of states from which $\mathbb{P}$ can start. An execution of $\mathbb{P}$, then, is a path in the LTS $p = \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \cdots \sigma_{n-1} \xrightarrow{\alpha_n} \sigma_n$, where $\sigma_0 \in S$, or $\langle t_1, t_2, \cdots, t_n \rangle$ for concision.

C-Chain In the simplest case, VALUESCOPE considers transitions that form a sequence. For the value of interest v bound to the variable x_n, its causality chain, or simply *c-chain*, is a subsequence of a path in the LTS: one end is the inception transition t^i whose action creates v, the other end is the observation transition

```
1      let i = 2020;
2      let x = read(); // reads 0 from a user
3      let y = zero(i); // i is irrelevant
4      let z = x + 0 * i; // i is irrelevant
5      print(z);
6      let err = z / y; // 0/0 generates a NaN
7      print(err);
8      function zero(arg) { return 0; }
```

Figure 5.3: A code snippet that illustrates the need for c-trees.

t^o where the user observes that x_n is bound to v. All other internal nodes are propagation transitions t^p whose actions copy v from one variable to another. With these notations, a c-chain is the sequence $\langle t^i, (t^p)^*, t^o \rangle$, where $*$ denotes 0 or more. A c-chain may have only a single transition when $t^i = t^o$, such as `print(NaN)` when $v =$ `NaN`. Programmatically, the action of t^i writes v into a variable but does not read v from a variable, such as an external interaction; the action of t^p both reads and writes v. Figure 5.2a depicts the c-chain of 0 in a simple program.

A c-chain presents a straightforward, linear view of the life cycle of a value and is useful. Casper [159] finds the c-chain of a `null` in a Java program and demonstrated its utility for debugging. A c-chain is conceptually related with a def-use chain: Given a variable v, a def-use chain is a sequence $\langle D, U_1, \cdots, U_n \rangle$, where D is a definition of v and $U_1, \cdots, U_n$ are all uses of v that only D and no other definitions of v can reach. A c-chain differs from a def-use chain in two ways. First, like a def-use chain, a c-chain also starts with a definition, but it is centred around a value, not a variable, so it may contain definitions of other variables as propagation transitions. Second, among all the uses of the defined variables, a c-chain contains only those uses that propagate, or the final use that manifests the value of interest. Therefore, a c-chain links multiple *partial* def-use chains.

A c-chain is easy to understand, but too simple: it tracks only a single value and overlooks that some values passed to an operator naturally cause others. Take the assignment `err = z / y` at line 6 in Figure 5.3 as an example. JavaScript encodes all numeric values in the double precision floating-point format; the fact that both `z` and `y` are `0` causes the RHS expression `z / y` to generate a `NaN`. When `NaN` is the value of interest and it is observed at line 7, VALUESCOPE backtracks to line 6 and deems it the inception transition of the `NaN`. It is natural for developers to then wonder why, in this execution, `z` and `y` have the value `0`. It may well be the case that the bug lies at the statement that binds `y` to `0`, which a c-chain misses.

C-Tree The limitation of a c-chain necessitates a more powerful data structure, which we call *c-tree*. Intuitively, a c-tree comprises c-chains of different values. For a state transition that reads v_i from a variable and writes v_j into another, a c-tree conceptually fuses the inception transition of v_j to the observation transition of v_i, thereby creating a more informative structure and hopefully speeding bug localisation.

How should one construct a c-tree? Given a state transition whose action α reads a set of variables X, one can trivially follow all the bindings for X. This exhaustive approach is expensive, as the number of bindings to track grows exponentially. More importantly, it ignores that some variables in X may be *irrelevant* to the computation of α. For example, consider the assignment `z = x + 0 * i` at line 4 in Figure 5.3 and assume `0` is the value of interest. This assignment reads from both `x` and `i`, but `i` is, in fact, irrelevant, because the RHS is semantically equivalent with just `x`. Thus, the result of an expression may depend on some of its free variables more than others and a c-tree should include only the more relevant bindings.

Relevance Analysis To build a c-tree efficiently, we introduce the *relevant variable* function $\triangledown$ that determines the set of relevant free variables in an expression. It permits an efficient way to explore a useful subset of all bindings in a value's inception transition. We define $\triangledown$ in Section 5.3 and discuss it at a high-level here. $\triangledown$ takes three parameters: a transition t_i, a variable x to which t_i's action writes the value of interest, and w, the maximum number of relevant variables an action permitted to have. w determines whether VALUESCOPE stops at a transition, labelling it as an inception transition, or deems a propagation transition and restarts on new values of interest $\triangledown$ computes. When interpreted as causes of a value of interest, too many inception transitions are unwieldy and are likely to violate Occam's razor [226]. For this reason and for scalability, we keep w small. Section 5.5.2 investigates the effects of varying w. $\triangledown(t_i, x, w)$ returns the variables that have a greater impact on the action and therefore on the production of the value of interest. We call these parameters *relevant*. It returns the empty set when α_i has no relevant parameters or has too many.

$\triangledown$ enables VALUESCOPE to dynamically generate and restart on new values of interest. It grants VALUESCOPE a natural support for multi-value tracking and obviates the need to predefine a set of values of interest.

Using $\triangledown$, we can now formally define a c-tree. For the value of interest v at the state σ where $\sigma(x_n) = v$, its *c-tree* is a tree of state transitions: the root is the observation transition, the leaves are inception transitions, and all internal nodes are propagation transitions, where

- the *observation transition* is a state transition $t^o \in \Sigma$, where the user observes $\sigma(x_n) = v$;
- a *inception transition* is a state transition $t^i \in \Sigma$, where $\triangledown(t^i, x_n) = \emptyset$;
- a *propagation transition* is a state transition $t^p \in \Sigma$, where $\triangledown(t^p, x_n) \neq \emptyset$.

Figure 5.2b exemplifies a c-tree.

Approach VALUESCOPE constructs a c-tree in a typical worklist algorithm, as Algorithm 3 outlines. Given a program $\mathbb{P}$, the algorithm takes as input the execution trace $p_o = \langle t_1, t_2, \cdots, t_o \rangle$ (*i.e.* a path in $\mathbb{P}$'s LTS that ends in the observation transition t_o), the variable x witnessed to hold the value of interest at the observation transition t_o, and the width parameter w for $\triangledown$. Key to understanding the algorithm is to understand the identifiers it uses. At line 5, VALUESCOPE uses the helper function *findDef* to find the definition of a particular use:

$$\begin{aligned}
\mathit{findDef}(p, \langle\langle\sigma_{i-1}, \alpha_i, \sigma_i\rangle, x\rangle) = \langle\sigma_{j-1}, \alpha_j, \sigma_j\rangle, \text{ where} \\
x \in \mathit{writes}(\alpha_j) \wedge j \leq i \wedge \\
(\nexists \langle\sigma_{k-1}, \alpha_k, \sigma_k\rangle \in p, x \in \mathit{writes}(\alpha_k) \wedge j < k \leq i)
\end{aligned}$$

The central data structure *WL* is a queue of pairs $\langle t_u, x\rangle$, where $t_u = \langle\sigma_{u-1}, \alpha_u, \sigma_u\rangle$. At any point in the execution of Algorithm 3, x holds the current value of interest

v, x_r holds the new one, t_u's α_u is a use of x, and t_d's α_d is the definition that writes v into x. The variable *cur*, defined on line 4 and used on line 8 and 11, points to the current parent node in the c-tree that the algorithm intends to build. On line 8, $\_$ denotes the absence of a relevant variable for t_d.

Algorithm 3 The VALUESCOPE algorithm: If no bindings α_i reads are relevant to its computation, VALUESCOPE deems the transition an inception and aborts. Otherwise, VALUESCOPE views the transition as a propagation and recurs on every binding in the relevant set $\triangledown$ returns. *cur*, the current parent node in the c-tree. $\sigma_u(x) = v$, the current the value of interest.

```
1:  procedure $VS(p_o, x, w)$                          ▷ $p_o = \langle t_1, t_2, \cdots, t_o \rangle$
2:      $WL \leftarrow \langle\langle t_o, x\rangle\rangle, root \leftarrow \langle t_o, x\rangle$      ▷ $t_o = \langle \sigma_{o-1}, \alpha_o, \sigma_o \rangle$
3:      while $WL \neq \langle\rangle$ do
4:          $cur = \langle t_u, x\rangle \leftarrow WL.dequeue()$             ▷ $t_u$, a use of $x$
5:          $t_d \leftarrow findDef(p_o, cur)$                  ▷ $t_d$, a definition of $x$
6:          $X_r \leftarrow \triangledown(t_d, x, w)$                     ▷ Section 5.3 defines $\triangledown$
7:          if $X_r = \emptyset$ then                       ▷ $t_d$, an inception trans.
8:              $cur.addChild(\langle t_d, \_\rangle)$                         ▷ add a leaf
9:          else                                   ▷ $t_d$, a propagation trans.
10:             for all $x_r \in X_r$ do
11:                 $cur.addChild(\langle t_d, x_r\rangle)$                 ▷ add a nonleaf
12:                 $WL.enqueue(\langle t_d, x_r\rangle)$
13:             end for
14:         end if
15:     end while
16:     return $root$
17: end procedure
```

5.2.1 Relation to Existing Work

Value Tracking VALUESCOPE generalises Casper [159] in two ways. Casper restricts the value of interest to `null`; VALUESCOPE accepts any primitive values that are not necessarily erroneous. Second, Casper aborts after identifying the inception transition of `null` and produces a c-chain; VALUESCOPE performs relevance analysis, restarts on the relevant bindings, and constructs a c-tree.

Dynamic Slicing VALUESCOPE differs from dynamic slicing [164] in its statement selection criteria and stopping condition. VALUESCOPE implements relevance-guided variable-to-value binding tracking, which does not select control flow statements, whereas dynamic slicing keeps all statements on which the slicing criterion is control or data dependent. A backward dynamic slicer stops at program entry, whereas VALUESCOPE stops at inception transitions. As a result, dynamic slicing returns much larger program subsequences than VALUESCOPE in practice.

Causal Analysis Cast in causal terms [227], an inception transition causes the manifestation of a value of interest. VALUESCOPE, however, is not a causal inference tool. Consider the causal relation $a \rightarrow b$. A causal analysis is given the events a and b and asked to determine whether a causes b, *i.e.* to discover a $\rightarrow$ that links them. For example, LDX [228] uses *dual execution* to determine whether a program event b, like executing an instruction, is causally dependent on a preceding event a. Specifically, from a given execution, LDX spawns a slave execution where it mutates the states at a and observes whether the mutation triggers any changes to b. VALUESCOPE takes the effect b, an observation transition, as input and exploits known causal relations, (*i.e.*, it knows $\rightarrow$, specifically program semantics) to discover a, a set of inception transitions. In short, causal analysis tools, like LDX, are not directly comparable to VALUESCOPE.

Dynamic Taint Analysis When values are viewed as a degenerate case of tags, VALUESCOPE resembles dynamic taint analysis [229], because it audits a value/tag's inception and propagation, but it also has important differences. Dynamic taint analysis tracks data flow from a known source to a known sink, following a forward execution; when an observation transition is deemed a source, VALUESCOPE employs a backward analysis to discover its unknown sinks, *i.e.* inception transitions.

5.3 Branching at Relevant Values

Section 5.2 generally describes VALUESCOPE, but uses the relevant variable function ∇ as a magic operator. This section defines it, by first bridging the gap between state transitions and functions.

Given a state transition, we assume its action α is deterministic. Under this assumption, we model α as a function f over the free variables it reads. We relax this constraint and discuss how VALUESCOPE combats non-determinism at the end of this section. For notational convenience, we assume all function applications use *named parameters*, as in JavaScript or Python. This assumption allows us to ignore the order of the arguments in an application and apply a function to a *set* of formal to actual bindings instead of a sequence of the arguments. It is not a fundamental restriction to the approach and is not required by the current implementation of VALUESCOPE, as Section 5.4 shows.

5.3.1 Sensitivity

Every function application binds its formal parameters to its actual ones. The result of a function application, however, does not necessarily “equally” depend on all the bindings of its parameters. Sensitivity analysis [230] is the general study of how much a function’s output changes in response to changes to its input. For example, the function $f(a,b) = 3a + 2b$ is more sensitive to a change in a, because of the greater coefficient `3`. As a result, one may argue that a is more relevant than b.

It is, however, challenging to directly apply sensitivity analysis to programs. Program behaviour is discrete over different domains, making it difficult to measure the changes between two outputs. One need only consider numeric types to illustrate the point. What is the numeric difference between `0` and `NaN`? How sensitive is a program to an input change that changes the output from `0` to `NaN`? And is that program more sensitive to another input change that changes the output to `1` to `NaN`?

In this work, we simplify the established notion of sensitivity to obviate the need to define “distance” for every type and across types. Instead of measuring the magnitude of the change in function’s output to changes in its inputs, we simply observe whether the output has changed. Thus, we change the question from ”How different are two outputs?” to ”Are two outputs different?” and reduce the range of sensitivity from $\mathbb{R}$ to $\{0,1\} = \mathbb{B}$.

Let $\mathfrak{B} = \{x_1 = a_1, x_2 = a_2, \cdots, x_n = a_n\}$ denote the parameter bindings for an application to the n-ary function f. For the function application generating a value of interest, $f(\mathfrak{B}) = v$, let $B \subset \mathfrak{B}$ be a subset of f’s parameter bindings to which we want to establish f’s sensitivity. Then $\overline{B} = \mathfrak{B} \setminus B$ contains the remaining bindings, which we keep fixed while viewing as variables the parameters that have bindings in B. Let $f_{\overline{B}}$ be the function resulting from *partially applying* f to $\overline{B}$ and $\Pi_X(B)$

be the set of parameters that B binds. Then, $f_{\overline{B}}$ is over $\Pi_X(\overline{B})$, *viz.* the parameters $\overline{B}$ does not bind and its arity is $|B| = |\mathfrak{B} \setminus \overline{B}|$. Consider, for example, applying the function $f(a,b,c) = a+b+c$ to $\{a=1, b=2, c=3\}$. If $B = \{a=1\}$, then $\overline{B} = \{b=2, c=3\}$ and $f_{\overline{B}}$ is the unary function $g(a) = a+5$.

Definition 5.3.1 (Sensitivity). *The application* $f(\mathfrak{B})$ *is* sensitive *to* $B \subset \mathfrak{B} \iff \forall B', \Pi_X(B) = \Pi_X(B') \wedge B \neq B' \wedge f_{\overline{B}}(B') \neq f_{\overline{B}}(B) = v$.

Intuitively, this definition states that, given a function application, if any changes to the bindings of some parameters cause the result to also change, the application is sensitive to these bindings. Given B that binds a subset of the function's parameters, the definition uses B' to denote some different bindings ($B \neq B'$) for the same variables ($\Pi_X(B) = \Pi_X(B')$), which cause the result to change ($f_{\overline{B}}(B') \neq f_{\overline{B}}(B)$).

5.3.2 Sensitivity under Testing

The goal of $\triangledown$ is to compute B, the new values of interest to which $f(\mathfrak{B})$ is sensitive, for VALUESCOPE to restart. $\triangledown$ can perform this computation symbolically or treat f as a black box and use fuzz testing. A symbolic approach could leverage some operators' special values, like `0` for multiplication, which, when used as an operand, determines the result regardless of the values of other operands, to speed the sensitivity computation. It would also need to contend with intractability, as f models a program expression that can be arbitrarily complex. In this work, we focus on black box testing, so our technique does not require source code. Black box testing f's sensitivity to X requires us to sample the inputs. Applying the principle of indifference [231], we uniformly sample the function's input domain.

Quantifying Relevance When a function is injective, it is sensitive to all of its parameters, because an injective function, given uniformly sampled inputs, generates uniformly distributed outputs. When we take a probabilistic view of the behaviour of f and model its output as a random variable O equipped with a probability distribution [232], the *Shannon entropy* [233] of O maximises, when it exhibits a uniform distribution. We use this insight to measure a function's sensitivity in the following definition.

Let H be the Shannon entropy of a discrete random variable. For the function application $f(\mathfrak{B})$ and $B \subset \mathfrak{B}$, let I be the random variable modelling the parameters that B binds and $f_{\overline{B}}(I)$ the dependent, output random variable.

Definition 5.3.2 (Relevance)**.** *For the function application $f(\mathfrak{B})$, the* relevance *of the set of parameters B binds to f is $H(f_{\overline{B}}(I))$.*

By modelling B as a random variable I and fixing the bindings $\overline{B}$ of the others, we make the resulting function, even a deterministic one, stochastic, denoted $f_{\overline{B}}(I)$. The relevance of B is then the Shannon entropy of this induced random variable.

Informally, Shannon entropy quantifies the uncertainty of a random variable. In particular, when its entropy is 0, a random variable has one certain outcome. When we view the parameters and the result of a function as random variables, if changes in the bindings of some parameters cause the result's entropy to be non-zero, then the result is uncertain and these parameters contribute to the uncertainty; in other words, the result depends on these parameters. More generally, high uncertainty induces large entropy. Thus, the more relevant some parameters are to the result, the more uncertain the random variable resulting from modifying these parameters' bindings will be, and consequently the larger the entropy will be.

Given a discrete random variable, computing its Shannon entropy takes as input its probability distribution. In Definition 5.3.2, the probability distribution of $f_{\overline{B}}(I)$ directly depends on that of I, when f is assumed to be deterministic. However, for functions converted from program expressions, inferring the probability distributions of the parameters' bindings is undecidable, even if the probability distribution of the inputs is already known. When one knows little about a random variable's probability distribution, uniform distribution that implies maximum uncertainty, is often used. Section 5.4.2 details how VALUESCOPE employs uniform distribution to approximate the entropy of $f_{\overline{B}}(I)$.

Thresholding Relevance In addition to quantifying the relevance of some parameters, we also would like to decide when they are *relevant*. To this end, we need a threshold: if the relevance of some parameters is above this threshold, they are relevant; otherwise, they are not. However, relevance is measured using entropy which varies with the underlying probability distribution. Therefore, setting a universal threshold of relevance, an absolute value of entropy appropriate for all probability distributions over a sample space, is impossible. Instead, we leverage the fact that given the sample space Ω, the entropy of a discrete random variable I is maximised when I has a discrete random distribution over its sample space Ω, or formally, $I \sim U_\Omega$. We introduce a parameter η that represents a fraction of the maximal entropy. By construction, this threshold is universal for all probability distributions over a sample space. Given $\mathfrak{B}$, let X be the set of parameters $B \subset \mathfrak{B}$ binds, *i.e.* $X = \Pi_X(B)$.

Definition 5.3.3 (Parameter Relevance)**.** *For the function application $f(\mathfrak{B})$, X is* relevant *if $H(f_{\overline{B}}(I)) > \eta \cdot H(U_\Omega)$, where $\eta \in [0,1)$ and Ω is the sample space for I.*

Functions cannot increase the Shannon entropy of their parameters. The inequality in the above definition puts a lower bound on how much entropy $f_{\overline{B}}$ loses on $\overline{B}$ and still are sensitive to X. Under this definition, an empty set of parameters is irrelevant. Formally, $X = \emptyset \implies B = \emptyset \wedge \overline{B} = \mathfrak{B}$; the partial application $f_{\overline{B}}$ now becomes a complete application that returns a constant and its entropy is always 0 regardless of the bindings in $\mathfrak{B}$. Additionally, unless a function uses none of its parameters, all of its parameters, taken together, must be relevant. Fundamentally, surjection causes partial relevance. In other words, if a function is injective, every parameter must be relevant.

Minimal Relevance The existence of some relevant parameters modulo a threshold, however, may hide the irrelevance of others. For example, when $\eta = 0$, $\{x, y\}$ is relevant to the function $x + 0 * y$, only because $\{x\}$ is relevant. We seek to capture these subsets of truly relevant parameters via *minimal relevance*. Given f, let $\mathfrak{X}$ be the set of f's parameters and let X and X' be two subsets of $\mathfrak{X}$.

Definition 5.3.4 (Minimally Relevant)**.** *For the function application $f(\mathfrak{B})$, a relevant subset $X \subseteq \mathfrak{X}$ is* minimal, *if $\nexists X' \subset \mathfrak{X}$, X' is relevant $\wedge$ $|X'| < |X|$.*

A set of relevant parameters of a function is *minimal* if no smaller relevant set exists. The constraint of minimality refines the set of all parameters, which is

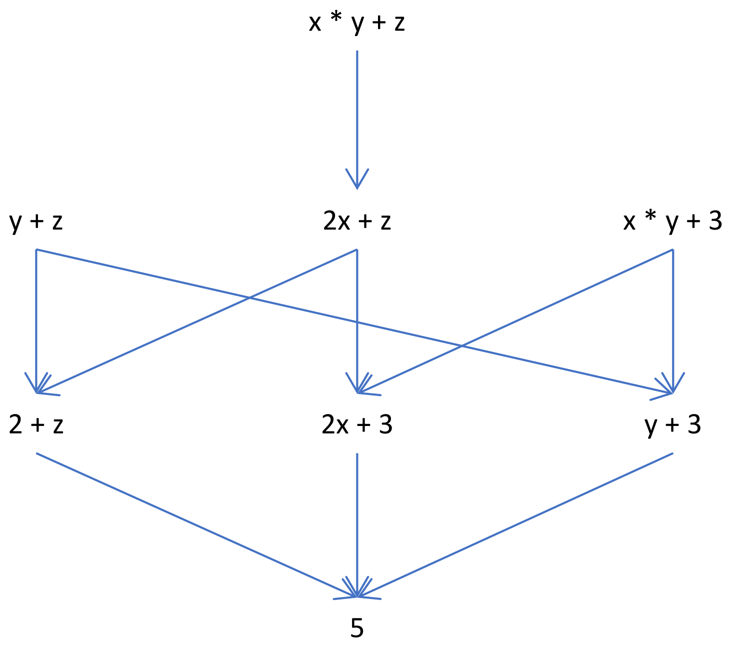


Figure 5.4: The partial function hierarchy for function $f(x,y,z) = x*y+z$ and $\mathfrak{B} = \{x = 1, y = 2, z = 3\}$. Arrows indicate partial applications.

relevant in general, to its very core of actually relevant parameters.

∇ aims to find the minimal relevant set of parameters of a transition, preventing VALUESCOPE from chasing irrelevant parameters. Given a function f and a set of parameter bindings B, different partial applications produce different functions, which we call *partial functions*. These partial functions, along with f, form a hierarchy. At the top sits f and at the bottom a constant that is the result of applying f to $\mathfrak{B}$. The functions in the middle levels are the results of partially applying the functions one level above to a binding of one of their parameters, thereby reducing the arity by one; functions at the same level have the same arity. As an example, Figure 5.4 displays the partial function hierarchy for $f(x,y,z) = x*y+z$ and $\mathfrak{B} = x = 1, y = 2, z = 3$. With this hierarchical concept, finding f's minimally relevant set of parameters is essentially a search problem, in which we aim to find a partial function in the hierarchy whose parameters' relevance is above the threshold and whose arity is minimised.

∇ tackles this problem in a bottom-up exhaustive search: it starts with testing the relevance of singleton sets that contain only a single parameter and gradually increases the arity of the partial function under investigation. The width parameter w that captures our philosophical view of causes limits the number of partial functions ∇ can test, which is $\sum_{i=1}^{w} \binom{w}{i}$. The exhaustive search stops when it finds a relevant set of parameters or finishes testing all the parameter sets whose sizes are smaller than $w+1$.

Example of Relevance Computation To be concrete on Definition 5.3.2 and 5.3.3, consider the expression `x * (y + z)`, in which variables `x`, `y`, and `z` are 2-bit unsigned integers. We first convert the expression to its function form $f(x,y,z) = x*(y+z)$. Assuming a program execution yields the binding $\{x = 0, y = 1, z = 2\}$. To test for relevance, VALUESCOPE explores subsets of this binding. When we model a non-empty subset of the parameters as a random variable, f has $2^3 - 1 = 7$ possible different partial applications. Under the assumption that all the parameters independently have a uniform distribution over the value domain of 2-bit unsigned integers, Table 5.1 presents the result's entropy in all 7 cases of partial applications. For example, when we model $\{x\}$ as a random variable and fix the binding $\{y = 1, z = 2\}$, the result of this partial application is $3x$ and its entropy is 1.4, the maximal value for random variables that have $2^2 = 4$ different outcomes. However, when we model either $\{y\}$ or $\{z\}$ as a random variable, the result's entropy is 0.0. If $\eta = 0$ in Definition 5.3.3, $\{x\}$, $\{x,y\}$, $\{x,z\}$, and $\{x,y,z\}$ are all relevant, but $\{x\}$, the truly relevant parameter set, is what ∇ aims to compute.

B	$\overline{B}$	X	$f_{\overline{B}}(X)$	$H(f_{\overline{B}}(X))$	$H(U_\Omega)$
{ x = 0 }	{ y = 1, z = 2 }	x	3x	1.4	1.4
{ y = 1 }	{ x = 0, z = 2 }	y	0	0.0	1.4
{ z = 2 }	{ x = 0, y = 1 }	z	0	0.0	1.4
{ x = 0, y = 1 }	{ z = 2 }	(x, y)	x(y+2)	2.3	2.8
{ y = 1, z = 2 }	{ x = 0 }	(y, z)	0	0.0	2.8
{ x = 0, z = 2 }	{ y = 1 }	(x, z)	x(z+1)	2.3	2.8
{ x = 0, y = 1, z = 2 }	$\emptyset$	(x, y, z)	x(y+z)	2.3	4.2

Table 5.1: The result's entropy for the function $f(x,y,z) = x*(y+z)$ with the binding $\mathfrak{B} = \{x = 0, y = 1, z = 2\}$ in 7 different cases of partial application.

Probabilistic Functions Functions in programs may be *probabilistic*; that is, a function may produce different outputs for the same input. Probabilistic functions adversely impact VALUESCOPE in two ways. First, they may introduce false positives to VALUESCOPE's relevance analysis. For example, consider a function that takes two parameters. Instead of using these parameters, the function simply returns the current time. This function is probabilistic and its parameters do not contribute to its computation. VALUESCOPE, however, will falsely deem the parameters relevant, because it will observe changes in the output with changes in the inputs. Second, probabilistic functions may cause the value of interest to not manifest, as in flaky tests. To contend with probabilistic functions, we set a resource budget to VALUESCOPE. We require the value of interest (or the relevant parameters) to manifest themselves sufficiently often, relative to this budget. Given a path to an observation transition that manifests the value of interest often enough to be detected, the budget cannot be exceeded at any intermediate transition during VALUESCOPE's computation. Thus, though falsely relevant parameters may cause VALUESCOPE to track to irrelevant parts of the program, eventually VALUESCOPE will be forced to stop because of the resource budget. Essentially, this budget bounds the probabilistic functions we can handle.

Special Values Given a function, there may exist a special value, like `0` for multiplication, such that the function applied to this special value will always produce the same result, regardless of the bindings of other parameters. When some parameters of a function are bound to these special values, the relevance of other parameters will be 0 (Definition 5.3.2). We define them as follows.

Definition 5.3.5 (Special Value). *For an n-ary function f, s is a* special value $\iff \forall B = \{x_1 = a_1, \cdots, x_n = a_n\}, \exists i \in \{1, \cdots, n\}, f(B[x_i = a_i/x_i = s]) = c$.

When $s = c$, we have a subset of special values that we call *fix values*. Under these definitions, in JavaScript , `undefined` is a special value for arithmetic operators when the other operand is numeric, because arithmetic operations involving `undefined` will evaluate to `NaN`. In floating point arithmetic, `NaN`, short for Not a Number, represents the result of undefined operations, such as $0 \div 0$ and $\infty - \infty$.

$$True \Rightarrow \quad \text{“}x = e\text{”} \to \text{“}x = ca(e)\text{”} \tag{5.1}$$

$$f \neq cu \Rightarrow \quad \text{“}f(a_1, \cdots, a_n)\text{”} \to \text{“}f(cp(a_1), \cdots, cp(a_n))\text{”} \tag{5.2}$$

$$l = observation \wedge pos(x) \neq LHS \Rightarrow \quad \text{“}x\text{”} \to \text{“}cu(x)\text{”} \tag{5.3}$$

Figure 5.5: The rewriting schema for VALUESCOPE's program transformation.

Floating point expressions involving `NaN` always evaluate to `NaN`, so `NaN` is a fix value for all floating point operators. In real arithmetic, 0 is a fix value for $\times$ and $\div$ when it is the nominator. In strictness analysis, a function f is *strict*, if and only if $f(\perp) = \perp$. Hence, $\perp$ is a fix value for any strict functions.

Many special values are actually silent or deferred errors. `NaN` is a notable example. By default, when a `NaN` is generated, the invalid flag, one of the five status bits designed to handle exceptional cases, is set, but without raising an exception that terminates the execution. Such a "quiet" `NaN` may be normalised by subsequent operations or even checked for to recompute the value, without incurring a costly termination. When unintended, these special values can propagate far from their origins and create errors that are hard to debug.

5.4 Implementation

Directly implementing VALUESCOPE (Algorithm 3) requires collecting complete execution traces of a JavaScript program, a monumental space cost that few would be willing to take. Instead, we realised VALUESCOPE by first transforming the program and then executing the transformed program to gradually construct the c-tree of a value of interest. This program transformation operates at the source code level, so dynamic instrumentation frameworks that target binaries, such as DynamoRIO [234] and Pin [235], are not suitable. VALUESCOPE applies the "replay" technique [236, 237, 238] to realise a reverse execution.

5.4.1 Transformation

At a high level, VALUESCOPE's program transformation turns the value of interest to a record by adding a field to it, builds a c-tree as the program executes, stores the c-tree in the additional field, and projects the c-tree to program statements when the execution reaches the observation transition. JavaScript is a large, complex, industrial language, notorious for its permissive syntax and semantics. Writing a program transformation that captures all value flows in raw JavaScript is impossible. So, our program transformation is *best effort*: we wrote it iteratively for our development corpus with the aim for, but not guarantee of, general applicability to real-world JavaScript programs.

Figure 5.5 illustrates the transformation's rewriting schema. For every assignment, VALUESCOPE wraps the right-hand side expression `e` with a function `ca` that checks its value (Equation 5.1). If `e` evaluates to the value of interest v, `ca` returns a special record that contains `e`'s original value and an additional field, a c-tree object noting an inception of the value of interest and recording the location of this expression. If `e` is itself a special record, `ca` regards it as a propagation transition and augments the existing c-tree object with the description and the location of the expression. Otherwise, `ca` is an identity function. For every function call, VALUESCOPE wraps every argument with a function `cp` (Equation 5.2) that behaves exactly the same as `ca`. For every use of a variable `x` at the observation transition, VALUESCOPE wraps it with a function `cu` that checks the value of `x` (Equation 5.3). If `x` is the special record, `cu` projects nodes in the c-tree to their corresponding statements in the program and outputs these statements. This projection converts a tree of transitions into a graph of statements. Otherwise, `cu` is an identity function. Finally, VALUESCOPE prepends the definitions of `ca`, `cp`, and `cu` to the transformed file.

Some operators in JavaScript, like `+=` and `++`, both read and write their operands. VALUESCOPE's transformation, however, treats a read and a write differently. A preprocessing that rewrites these operations such that the reads and the writes are separated at the source code level and maintains the program semantics is therefore required, shown in Figure 5.6. Specifically, this preprocessing converts short-hand assignments into their corresponding complete forms (Equation 5.6), *e.g.*, `a += b` to `a = a + b`. It also rewrites prefix and postfix increment and decrement operations. While the prefix ones are simply semantically equivalent to first adding/subtracting

$$\otimes \in \{++, --\} \quad \Rightarrow \quad \text{“}\otimes x\text{”} \rightarrow \text{“}x = x\ m(\otimes)\ 1\text{”} \tag{5.4}$$
$$\otimes \in \{++, --\} \quad \Rightarrow \quad \text{“}x \otimes\text{”} \rightarrow \text{“}(x = x\ m(\otimes)\ 1, x\ o(\otimes)\ 1)\text{”} \tag{5.5}$$
$$\otimes \in \{+=, -=, *=, /=, \%=\} \quad \Rightarrow \quad \text{“}x \otimes e\text{”} \rightarrow \text{“}x = x\ m(\otimes)\ e\text{”} \tag{5.6}$$

Figure 5.6: The rewriting schema of VALUESCOPE's preprocessing.

one to/from the operands and then assigning the new values to them (Equation 5.4), rewriting the postfix ones is slightly more complicated. Here VALUESCOPE utilise the `,` operator in JavaScript, which evaluates each of its operands (from left to right) and returns the value of the last operand. Therefore, Equation 5.5 first increments/decrements the operands by one and then returns the original values for further computation.

We implement VALUESCOPE's program transformation using esprima[4] which parses JavaScript code into an AST and escodegen[5] which generates code from an AST. Despite our best effort, this transformation does not support full JavaScript. For example, it does not handle `eval` and overlooks implicit, conceptual assignments, like `Object.assign(target, source)`.

5.4.2 Relevance Analysis

As Section 5.2 mentioned, the probabilistic distributions of the arguments are generally unknown. We apply the principle of indifference [231] and assume all the arguments have uniform distributions over the value domains of their types. The users can supply us with more realistic distributions. Further, the value domains of some types, like `double` or, worse, `string`, are too large for computation. We resort to uniform sampling to shrink the value domains of these types and approximate the arguments' uniform probability distributions.

The sampling approach has several limitations. First, it may miss values that have significant impact on the result but account for only a tiny proportion of the whole domain. Take the expression `a / b` as an example, where all variables are *half-precision* floating-point numbers. Assume in a particular execution, we have

[4]`http://esprima.org/`
[5]`https://github.com/estools/escodegen`

the following value binding: `{a = 0.0, b = 0.0}`, this division therefore evaluates to `NaN`. When we model `b` as a random variable, exhibiting a uniform distribution over its value domain, and bind `a` to `0.0`, this partial application returns the function `0.0/b`. Because for half-precision floating-point formats, there are $63,490$ distinct values, including infinities and `NaN`, the probability of sampling 0.0 for `b` is $\frac{1}{63490}$. Therefore, the result of `0.0/b` via sampling the value of `b` is highly likely to be 0.0 and the entropy 0.0. Second, the independent, isolated view of each argument loses the relations among them before the expression. Some values resulted from sampling may be impossible to occur in reality because of these relations.

Introduced in Section 5.2, $\triangledown$ has a parameter w that limits the number of relevant bindings VALUESCOPE tracks at a particular transition. This is an attempt to limit the number of inception transitions, so that VALUESCOPE does not violate Occam's razor [226], when viewing inception transitions as causes of the interesting value's manifestation. In implementation, we equip $\triangledown$ with another parameter d to limit the number of restart VALUESCOPE performs in a single c-tree branch. If we imagine $\triangledown$ as a tree traversal from the root, then w controls the number of children to visit from a node and d controls the depth of the whole traversal. When $d = 0$, VALUESCOPE turns off relevance analysis and degenerates into Casper. The current implementation defaults w and d to 3 and 1 respectively.

5.5 Evaluation

This section evaluates VALUESCOPE's effectiveness in localising the causes of faulty values on real-world programs. We first describe the experimental setup, including the baseline techniques, the corpus, and the measures. Then, we measure VALUESCOPE's performance on the corpus and compare it with the baselines. Next, we tune VALUESCOPE's two parameters, width and depth, and investigate their impacts on VALUESCOPE's performance.

5.5.1 Experimental Setup

Baselines Spectrum-based fault localisation (SBFL) [239] has attracted much attention for its efficiency and deployability. Simply put, it correlates the "faultiness" of a program entity, such as a statement, with its coverage in executing the failed test cases. Pearson *et al.* [240] have empirically compared various SBFL techniques, demonstrating that the DStar ranking measure [11], or DStar for brevity, outperforms the others. Therefore, we consider DStar as a baseline. Program slicing [241] has also been extensively studied and one of its applications is debugging. ORBS [12] is an observation-based program slicer capable of slicing a multi-language system. It produces a slice by repeatedly deleting statements until the program's observable behaviour changes. ORBS is publicly available and able to slice JavaScript programs due to its language independence. We also include it as a baseline.

Like VALUESCOPE, DStar and ORBS are line granular. They, however, are not designed for the problem of value localisation. DStar seeks to localise a larger class of potential bugs, and ORBS is more general than DStar, since its slices have many other potential uses beyond localisation, and therefore, like SBFL, it can handle a larger class of bugs than VALUESCOPE. This experiment does not evaluate their effectiveness in localising non-value bugs.

Corpus To collect a corpus of real-world bugs, we first built a set of keywords, including ”NaN”, ”null”, and ”undefined”. Then we searched on GitHub for issues whose titles and discussions contained the keywords and which had been closed so that we could identify the fixes and therefore the bugs. Some of these issues are not bug-related. To filter them, we picked an issue uniformly at random and manually inspected it to determine whether it conformed with three requirements: 1) it must track a value-related bug; 2) it must specify the bug-triggering input; 3) it must specify the observation transition. In this manual process, we gathered 25 bugs. We also considered BugsJS [242], a benchmark of 453 JavaScript bugs from 10 popular server-side projects on GitHub. Each bug is accompanied by its bug report, the test cases that detect it, as well as the patch that fixes it, so BugsJS is well-suited for this experiment. We manually combed through the benchmark and kept 19 value-related bugs. In summary, our corpus contains 44 bugs from diverse projects, whose number of stars ranges from 0 to 40,000. However, only 20 bugs come with a test suite, which DStar requires, so DStar is applicable to only these 20 bugs.

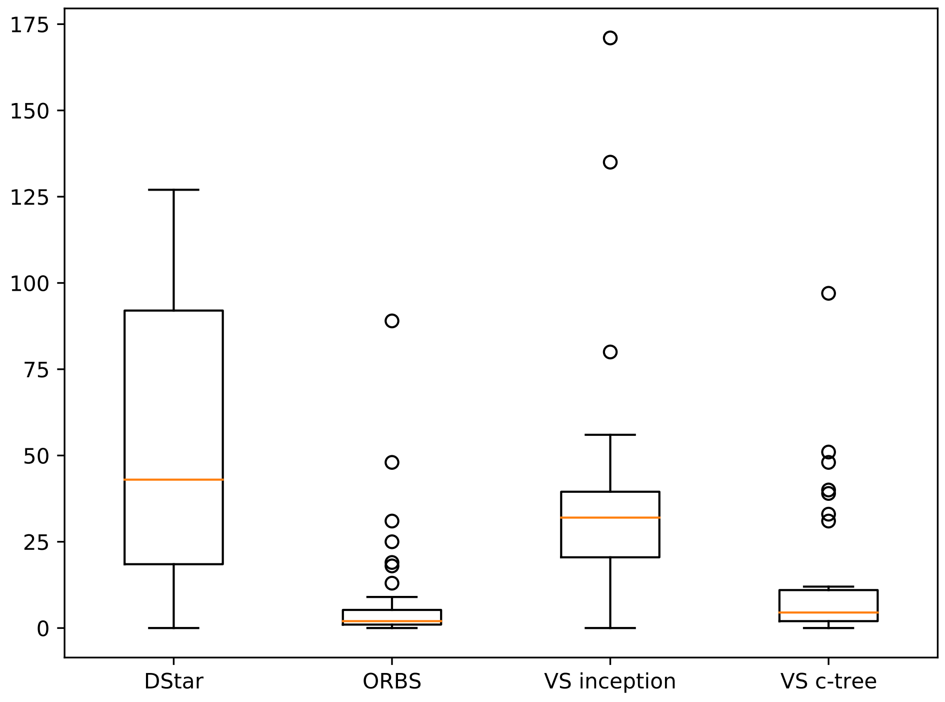


Figure 5.7: Comparing fault localisation effectiveness.

Measures Over our corpus, we assume that every fix correctly eliminates its bug. VALUESCOPE and ORBS return sets of lines and DStar returns a ranked list of lines. Therefore, we measure the effectiveness of fault localisation in the minimal distance in lines of code (LOC) between any line a technique reports when localising a bug and any line in the region patched by the fix.

5.5.2 Results

Fault Localisation Figure 5.7 compares DStar, ORBS, and VALUESCOPE with default configuration (*i.e.* $w = 3$ and $d = 1$) on the smallest distance between any line a technique reports and any line in the patched region. The median of the smallest distance for VALUESCOPE is 32 lines and the average is 36 lines, 11 and 18 lines closer than that for DStar, respectively. ORBS, on the other hand, performs extremely well on the smallest distance measure, achieving a median of 2 lines and a mean of 8 lines. The reason for ORBS' impressive performance is that it generally produces large slices, so the chances of its slices overlap with or near the patched regions are very high. Figure 5.8 compares the size in lines of c-trees VALUESCOPE produces

Target	**Frequency**
Same line?	2%
On screen?	20%
Same function?	53%

Table 5.2: The chances of the identified inception transitions hitting different targets.

and slices ORBS produces. On average, the size of a c-tree is 12 lines, 12.5% of the size of a slice ORBS computes.

To make these distances more meaningful, we introduce the On Screen measure, which determines whether the tool-reported line and the patched line that are closest can appear on the same screen in an editor when the former is at the middle of the screen. This measure reflects the ideal case where a developer can focus on the same screen without scrolling, when debugging from the tool-reported line. We performed a convenience sampling to estimate the lines of code a typical editor can fit on a screen. On average, our screen size is 37 lines, which we use for this experiment. Table 5.2 reports how often an inception transition VALUESCOPE identifies is 1) actually changed by the fix (*i.e.* a perfect localisation), 2) on screen, and 3) in the same innermost function with the bug. VALUESCOPE successfully localises a bug. For 20% of the buggy programs in our corpus, it is able to identify a line that is half screen away from the bug, so that if the identified line is at the middle of the screen, a developer can generally avoid scrolling up and down, hunting for the bug.

So far, we have used only the inception transitions VALUESCOPE identifies. Next we study how much more effective VALUESCOPE can be when also including the propagation transitions for the minimal distance measure. As the last two boxes in Figure 5.7 illustrate, when considering the whole c-tree, a line VALUESCOPE reports is on average 12 lines away from the fix and the on-screen rate is 84%.

Ablation and Tuning VALUESCOPE has two parameters: w, which limits the number of relevant bindings to track at a particular transition, and d, which limits the number of restarts in a single branch of a c-tree. Here we investigate their impacts on VALUESCOPE's on-screen rate in fault localisation by allowing w and d to range over $[1,5]$ and $[0,2]$. Figure 5.9 visualises the results of this ablation study. While

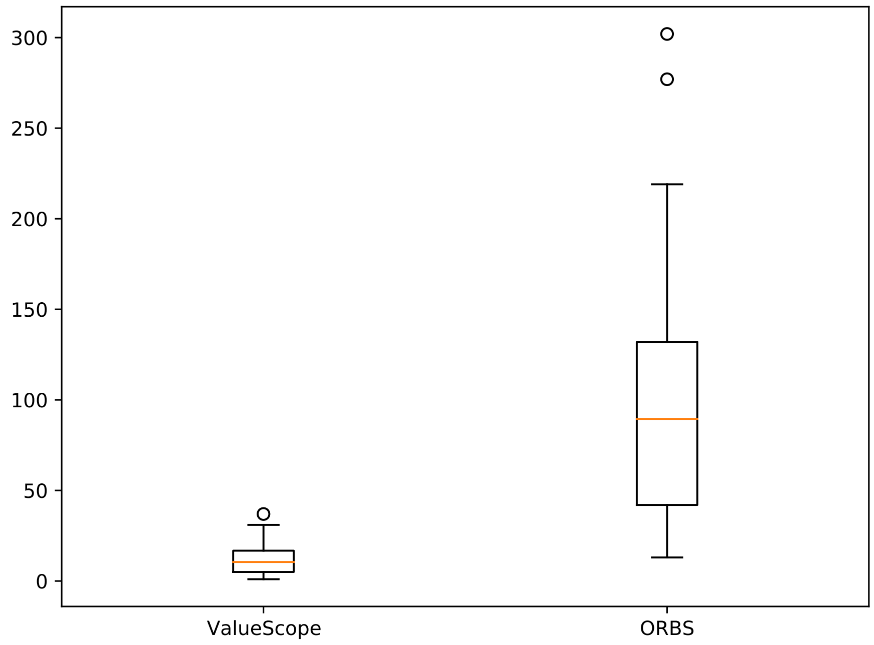


Figure 5.8: A size comparison of the c-trees VALUESCOPE constructs and the slices ORBS computes.

increasing w is beneficial for the minimal distance measure, blindly increasing d may adverse impact VALUESCOPE. When VALUESCOPE degenerates into Casper ($d = 0$), the on-screen rate is the same as when VALUESCOPE runs with the default configuration ($w = 3$ and $d = 1$). In fact, the bugs for which VALUESCOPE is able to identify an on-screen inception transition are not exactly identical, which we explain as follows.

A developer can decide where to fix a bug: "shallow" vs. "deep". A shallow fix is one closer to the bug's symptom, *e.g.* the observation transition of an erroneous value. In a shallow fix, the developer blocks the flow of the erroneous value at one of its propagation transitions. In a case like this, a c-chain produced by degenerate VALUESCOPE is sufficient to localise the bug.

In a deep fix, the developer directly patches the erroneous value's originating expressions (Definition 5.1.1). In this case, VALUESCOPE with relevance analysis outperforms Casper. When both a shallow and a deep fix are viable, we contend that deep is better, because, to borrow terms from taint analysis, it eliminates the chance of erroneous data flowing from its source to other sinks.

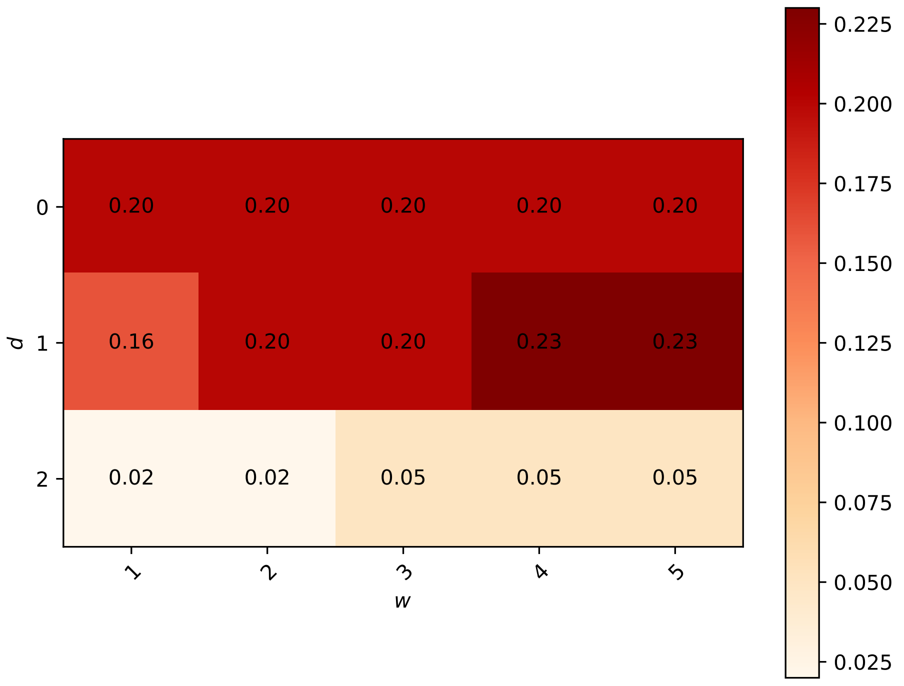


Figure 5.9: VALUESCOPE's effectiveness in fault localisation for different combinations of w and d.

5.6 Chapter Summary

Debugging involves backward reasoning. Given a bug that is triggered by an unexpected value, developers often wonder where this value is from and how it propagates from the origin to here. This chapter presents VALUESCOPE, a tool that aids program understanding by automatically solving the value origin problem. When used in bug localisation, a line VALUESCOPE reports is, on average, 12 lines away from the fix, 42 lines closer than that of DStar, while the total number of lines VALUESCOPE reports is on average 12.5% of that of ORBS.

Chapter 6

Conclusion and Future Work

Program analysis becomes increasingly critical to ensuring software quality. Researchers, however, often seek to devise new analysis techniques, neglecting that making programs more amenable to the existing techniques is also of great value. This thesis aims to fill this gap through program transformation and eliminates the need to reinvent the wheel of bespoke program transformations.

Chapter 2 of this thesis reviews literature on specific program analysis techniques, during which three research questions arise. Chapter 3 to 5 present three interconnected projects, each of which attempts to advance the state of the art of facilitating program analysis through transformation. Specifically, Chapter 3 empirically quantifies the benefit of Flow and TypeScript in terms of early bug detection. The results show that both of them are able to detect 15% of the collected bugs, while the transformation costs in terms of time and number of tokens added are low. Chapter 4 presents TYPERITE, a practical, scalable tool which automates retyping an arbitrary part of a program, such as a function or a single variable. TYPERITE scales to large codebases, retyping Pidgin (363K lines of code) and GSL's specfunction (61,378 lines of code) with 129,585 modifications. TYPERITE is useful: had it been available, developers could have used it to detect 4 real-world bugs, including a SQL injection. Chapter 5 discusses VALUESCOPE, a tool that transforms JavaScript programs to enable a value-granular backward program analysis. Experimental results show that the c-chains VALUESCOPE generate are compact, compared against ORBS, a slicing technique. When used in bug localisation, a line VALUESCOPE reports is, on average, 12 lines away from the fix, 42 lines closer than that of DStar,a state of the art spectrum based fault localiser.

Future work includes 1) extending the empirical study on static typing to investigate how project quality and bug severity affect the results; 2) supplementing TYPERITE with a library of types and operators, like tainted versions of common types, that would be useful retyping targets; 3) exploring the traditional sensitivity view of relevance discussed in Section 5.3, which considers the impact of unit inputs on outputs.

Bibliography

[1] Strategic Planning. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, page 1, 2002.

[2] Tobias Buck and Peggy Hollinger. BA faces £80m cost for it failure that stranded 75,000 passengers. *Financial Times*, June 2017.

[3] The U.S. General Accounting Office. Patriot missile defense: Software problem led to system failure at dhahran, saudi arabia. 1992.

[4] Wikipedia contributors. Program analysis, 2021. [Online; accessed 22-July-2021].

[5] Mukesh Soni. Defect prevention: Reducing costs and enhancing quality. *IBM: iSixSigma.com*, 19, 2006.

[6] Bertrand Russell. *Principles of mathematics*. Routledge, 2009.

[7] Eric Goubault. Static analyses of the precision of floating-point operations. In *Static Analysis*, pages 234–259. Springer Berlin Heidelberg, 2001.

[8] Matthieu Martel. Propagation of roundoff errors in finite precision computations: A semantics approach. In *Programming Languages and Systems*, pages 194–208. Springer, 2002.

[9] Eric Goubault, Matthieu Martel, and Sylvie Putot. Asserting the precision of floating-point computations: A simple abstract interpreter. In *Programming Languages and Systems*, pages 209–212. Springer Berlin Heidelberg, 2002.

[10] Tevfik Bultan, Fang Yu, Muath Alkhalaf, and Abdulbaki Aydin. String manipulating programs and difficulty of their analysis. In *String Analysis for Software Verification and Security*, pages 15–22. Springer, 2017.

[11] W Eric Wong, Vidroha Debroy, Yihao Li, and Ruizhi Gao. Software fault localization using dstar (d*). In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 21–30. IEEE, 2012.

[12] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. ORBS: Language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120. ACM, 2014.

[13] Jan Willem Klop, Marc Bezem, and RC De Vrijer. *Term rewriting systems*. Cambridge University Press, 2001.

[14] Martin Ward. *Proving program refinements and transformations*. PhD thesis, University of Oxford PhD Thesis, 1989.

[15] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[16] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[17] Edsger Wybe Dijkstra et al. Notes on structured programming, 1970.

[18] Paul R Halmos. *Naive set theory*. Courier Dover Publications, 2017.

[19] Bertrand Russell. *The principles of mathematics*. WW Norton & Company, 1996.

[20] Bertrand Russell. Mathematical logic as based on the theory of types. *American journal of mathematics*, 30(3):222–262, 1908.

[21] Frank P Ramsey. The foundations of mathematics. *Proceedings of the London Mathematical Society*, 2(1):338–384, 1926.

[22] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

[23] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Studies in Logic and the Foundations of Mathematics*, volume 80, pages 73–118. Elsevier, 1975.

[24] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[25] Stefano Berardi. Towards a mathematical analysis of the coquand-huet calculus of constructions and the other systems in barendregt's cube. *Technica1 report, Carnegie-Me11on University (USA) and Universita di Torino (Ita1y)*, 1988.

[26] Jan Terlouw. Een nadere bewijstheoretische analyse van gstt's. *Manuscript (in Dutch)*, 1989.

[27] Henk P Barendregt. Lambda calculi with types. 1992.

[28] Andrzej Mostowski. From frege to gödel. a source book in mathematical logic 1879-1931, 1968.

[29] Benjamin C Pierce. *Types and programming languages*. The MIT Press, 2002.

[30] Peter Thiemann. Towards a type system for analyzing JavaScript programs. In *European Symposium On Programming*, pages 408–422. Springer, 2005.

[31] Mark T Daly, Vibha Sazawal, and Jeffrey S Foster. Work in progress: An empirical study of static typing in Ruby. 2009.

[32] Andreas Stuchlik and Stefan Hanenberg. Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time. In *Proceedings of the 7th symposium on Dynamic languages*, pages 97–106, 2011.

[33] Sebastian Kleinschmager, Stefan Hanenberg, Romain Robbes, É Tanter, and Andreas Stefik. Do static type systems improve the maintainability of software systems? an empirical study. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 153–162. IEEE, 2012.

[34] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *ACM SIGPLAN Notices*, volume 47, pages 683–702. ACM, 2012.

[35] Leo A Meyerovich and Ariel S Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 1–18, 2013.

[36] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM, 2014.

[37] Sebastian Nanz and Carlo A Furia. A comparative study of programming languages in rosetta code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 778–788. IEEE, 2015.

[38] Lutz Prechelt and Walter F Tichy. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Transactions on Software Engineering*, 24(4):302–312, 1998.

[39] Stefan Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *ACM Sigplan Notices*, volume 45, pages 22–35. ACM, 2010.

[40] Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. *Combinatory logic*, volume 1. North-Holland Amsterdam, 1958.

[41] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[42] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[43] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[44] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):253–289, 1993.

[45] Barbara Liskov and Stephen Zilles. Programming with abstract data types. *ACM Sigplan Notices*, 9(4):50–59, 1974.

[46] K Jackson. Parallel processing and modular software construction. In *Design and Implementation of Programming Languages*, pages 436–443. Springer, 1977.

[47] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Static Analysis*, pages 238–255. Springer, 2009.

[48] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for JavaScript. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 587–606, 2012.

[49] Michael Furr, Jong-hoon An, and Jeffrey S Foster. Profile-guided static typing for dynamic scripting languages. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 283–300, 2009.

[50] Michael Pradel and Koushik Sen. The good, the bad, and the ugly: An empirical study of implicit type conversions in JavaScript. In *29th Euro-*

pean Conference on Object-Oriented Programming (ECOOP 2015). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[51] Jan Willem Klop and JW Klop. *Term rewriting systems*. Centrum voor Wiskunde en Informatica, 1990.

[52] Hendrik Pieter Barendregt, Marko CJD van Eekelen, John RW Glauert, J Richard Kennaway, Marinus J Plasmeijer, and M Ronan Sleep. Term graph rewriting. In *PARLE Parallel Architectures and Languages Europe*, pages 141–158. Springer, 1987.

[53] Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, pages 346–366, 1932.

[54] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[55] Alan M Turing. Computability and λ-definability. *The Journal of Symbolic Logic*, 2(4):153–163, 1937.

[56] Stephen C Kleene and J Barkley Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, pages 630–636, 1935.

[57] Simon Thompson. *Type theory and functional programming*. Addison Wesley, 1991.

[58] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[59] Wikipedia. Lambda calculus — Wikipedia, the free encyclopedia, 2017. [Online; accessed 24-November-2017].

[60] Franklyn Turbak, David Gifford, and Mark A Sheldon. *Design concepts in programming languages*. MIT press, 2008.

[61] Colin Kemp. *Theoretical Foundations for Practical 'Totally-Functional Programming'*. PhD thesis, PhD Thesis, The University of Queensland, St Lucia, 2009.

[62] William W Tait. Intensional interpretations of functionals of finite type i. *The journal of symbolic logic*, 32(2):198–212, 1967.

[63] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. pages 203–211, 1991.

[64] Luca Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, 1996.

[65] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.

[66] Harry G Mairson. Deciding ml typability is complete for deterministic exponential time. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 382–401, 1989.

[67] George Kuan and David MacQueen. Efficient type inference using ranked type variables. In *Proceedings of the 2007 workshop on Workshop on ML*, pages 3–14, 2007.

[68] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, PhD thesis, Université Paris VII, 1972.

[69] John C Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.

[70] John C Reynolds. Using category theory to design implicit conversions and generic operators. In *International Workshop on Semantics-Directed Compiler Generation*, pages 211–258. Springer, 1980.

[71] Luca Cardelli. A semantics of multiple inheritance. *Semantics of data types*, pages 51–67, 1984.

[72] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM transactions on programming languages and systems (TOPLAS)*, 13(2):237–268, 1991.

[73] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *ACM Sigplan Notices*, 27(8):15–42, 1992.

[74] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, 1991.

[75] Olaf Chitil. Practical typed lazy contracts. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 67–76, 2012.

[76] Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2007.

[77] Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[78] Gilad Bracha. Pluggable type systems. In *OOPSLA workshop on revival of dynamic languages*, volume 4, 2004.

[79] Mario Coppo and Mariangiola Dezani-Ciancaglini. A new type assignment for λ-terms. *Archiv für mathematische Logik und Grundlagenforschung*, 19(1):139–156, 1978.

[80] Steffen Van Bakel. Complete restrictions of the intersection type discipline. *Theoretical Computer Science*, 102(1):135–163, 1992.

[81] Steffen Van Bakel. Strict intersection types for the lambda calculus. *ACM Computing Surveys (CSUR)*, 43(3):20, 2011.

[82] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 27(2-6):45–58, 1981.

[83] Jean-Yves Girard. Une extension de l'interpretation de gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. *Studies in Logic and the Foundations of Mathematics*, 63:63–92, 1971.

[84] Samson Abramsky et al. The lazy lambda calculus. *Research topics in functional programming*, 65, 1990.

[85] Antonio Bucciarelli, Silvia De Lorenzis, Adolfo Piperno, and Ivano Salvo. Some computational properties of intersection types. In *Logic in Computer Science, 1999. Proceedings. 14th Symposium on*, pages 109–118. IEEE, 1999.

[86] Daniel Leivant. Discrete polymorphism. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 288–297. ACM, 1990.

[87] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The journal of symbolic logic*, 48(04):931–940, 1983.

[88] S Ronchi Della Rocca and Betti Venneri. Principal type schemes for an extended type theory. *Theoretical Computer Science*, 28(1):151–169, 1983.

[89] Steffen Van Bakel. Intersection type assignment systems. *Theoretical Computer Science*, 151(2):385–435, 1995.

[90] Mario Coppo and Paola Giannini. A complete type inference algorithm for simple intersection types. In *Colloquium on Trees in Algebra and Programming*, pages 102–123. Springer, 1992.

[91] Ferruccio Damiani and Paola Giannini. A decidable intersection type system based on relevance. In *Theoretical Aspects of Computer Software*, pages 707–725. Springer, 1994.

[92] Stephanus Johannes Van Bakel. *Intersection type disciplines in lambda calculus and applicative term rewriting systems.* Citeseer, 1993.

[93] Trevor Jim. Rank 2 type systems and recursive definitions. *Massachusetts Institute of Technology, Cambridge, MA*, 1995.

[94] Mario Coppo and Paola Giannini. Principal types and unification for simple intersection type systems. *Information and Computation*, 122(1):70–96, 1995.

[95] Trevor Jim. What are principal typings and what are they good for? In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–53. ACM, 1996.

[96] Assaf J Kfoury and Joe B Wells. Principality and decidable type inference for finite-rank intersection types. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 161–174, 1999.

[97] Daniel Leivant. Polymorphic type inference. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 88–98, 1983.

[98] Steffen van Bakel. Rank 2 intersection type assignment in term rewriting systems. *Fundamenta Informaticae*, 26(2):141–166, 1996.

[99] Tachio Terauchi and Alex Aiken. On typability for rank-2 intersection types with polymorphic recursion. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 111–122. IEEE, 2006.

[100] Assaf Kfoury, Geoffrey Washburn, and Joe B Wells. Implementing compositional analysis using intersection types with expansion variables. *Electronic Notes in Theoretical Computer Science*, 70(1):124–148, 2003.

[101] Assaf J Kfoury and Joe B Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 196–207, 1994.

[102] Sandra Alves and Mario Florido. On the relation between rank 2 intersection types and simple types. In *Joint Conference on Declarative Programming (AGP'2002)*, pages 259–274. Citeseer, 2002.

[103] Bas Kloet and Marcel Moreaux. Essay aspects of programming languages: *Union type*. `http://ledr.luon.net/documents/2M240/essay-unions/essay.pdf`, 2005.

[104] Franco Barbanera and Mariangiola Dezani-Ciancaglini. Intersection and union types. In *Theoretical Aspects of Computer Software*, pages 651–674. Springer, 1991.

[105] Steffen Van Bakel. Completeness and partial soundness results for intersection and union typing for $\bar{\lambda}\mu\tilde{\mu}$. *Annals of Pure and Applied Logic*, 161(11):1400–1430, 2010.

[106] Steffen Van Bakel. Intersection and union types for x. *Electronic Notes in Theoretical Computer Science*, 136:203–227, 2005.

[107] Benjamin C Pierce. Programming with intersection types, union types, and polymorphism. *CMU*, 1991.

[108] Francisco Ortin and Miguel García. Union and intersection types to support both dynamic and static typing. *Information Processing Letters*, 111(6):278–286, 2011.

[109] Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Betti Venneri, et al. The" relevance" of intersection and union types. *Notre Dame Journal of Formal Logic*, 38(2):246–269, 1997.

[110] Flemming M Damm. Subtyping with union types, intersection types and recursive types. In *Theoretical Aspects of Computer Software*, pages 687–706. Springer, 1994.

[111] Peter Buneman and Benjamin Pierce. Union types for semistructured data. In *Research Issues in Structured and Semistructured Database Programming*, pages 184–207. Springer, 2000.

[112] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *Journal of Functional Programming*, 11(03):263–317, 2001.

[113] Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1435–1441. ACM, 2006.

[114] Franco Barbanera, Mariangiola Dezaniciancaglini, and Ugo Deliguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

[115] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[116] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[117] Cristian Cadar. Targeted program transformations for symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 906–909. ACM, 2015.

[118] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[119] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[120] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005.

[121] Paul Dan Marinescu and Cristian Cadar. Make test-zesti: A symbolic execution solution for improving regression testing. In *Proceedings of the 34th International Conference on Software Engineering*, pages 716–726. IEEE Press, 2012.

[122] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[123] Willard V Quine. Concatenation as a basis for arithmetic. *The Journal of Symbolic Logic*, 11(4):105–114, 1946.

[124] Margus Veanes, Nikolaj Bjørner, and Leonardo De Moura. Symbolic automata constraint solving. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 640–654. Springer, 2010.

[125] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 188–198, 2009.

[126] Guodong Li and Indradeep Ghosh. PASS: String solving with parameterized array and interval automaton. In *Haifa Verification Conference*, pages 15–31. Springer, 2013.

[127] Fang Yu, Tevfik Bultan, and Oscar H Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 322–336. Springer, 2009.

[128] Vijay Ganesh and David L Dill. A decision procedure for bit-vectors and

arrays. In *International Conference on Computer Aided Verification*, pages 519–531. Springer, 2007.

[129] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A dpll (t) theory solver for a theory of strings and regular expressions. In *International Conference on Computer Aided Verification*, pages 646–662. Springer, 2014.

[130] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1232–1243. ACM, 2014.

[131] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A Z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 114–124, 2013.

[132] Angelo Brillout, Daniel Kroening, and Thomas Wahl. Mixed abstractions for floating-point arithmetic. In *2009 Formal Methods in Computer-Aided Design*, pages 69–76. IEEE, 2009.

[133] Eva Darulova and Viktor Kuncak. Sound compilation of reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 235–248, 2014.

[134] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan de Halleux. Flopsy-search-based floating point constraint solving for symbolic execution. In *IFIP International Conference on Testing Software and Systems*, pages 142–157. Springer, 2010.

[135] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, (3):215–222, 1976.

[136] Zhoulai Fu and Zhendong Su. Mathematical execution: A unified approach for testing numerical code. *arXiv preprint arXiv:1610.01133*, 2016.

[137] Matheus Souza, Mateus Borges, Marcelo d?Amorim, and Corina S Păsăreanu. Coral: Solving complex constraints for symbolic pathfinder. In *NASA Formal Methods Symposium*, pages 359–374. Springer, 2011.

[138] Willem Visser, Corina S Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107, 2004.

[139] Peter Collingbourne, Cristian Cadar, and Paul HJ Kelly. Symbolic crosschecking of floating-point and simd code. In *Proceedings of the sixth Conference on Computer Systems*, pages 315–328. ACM, 2011.

[140] Peter Collingbourne, Cristian Cadar, and Paul HJ Kelly. Symbolic cross-checking of data-parallel floating-point code. *IEEE Transactions on Software Engineering*, 40(7):710–737, 2014.

[141] Claude Michel, Michel Rueher, and Yahia Lebbah. Solving constraints over floating-point numbers. In *International Conference on Principles and Practice of Constraint Programming*, pages 524–538. Springer, 2001.

[142] Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, 16(2):97–121, 2006.

[143] Willem Visser, Jaco Geldenhuys, and Matthew B Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[144] Nicolás Rosner, Jaco Geldenhuys, Nazareno M Aguirre, Willem Visser, and Marcelo F Frias. Bliss: Improved symbolic execution by bounded lazy initialization with sat support. *IEEE Transactions on Software Engineering*, 41(7):639–660, 2015.

[145] Guowei Yang, Corina S Păsăreanu, and Sarfraz Khurshid. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 144–154. ACM, 2012.

[146] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *ACM SIGPLAN Notices*, volume 46, pages 504–515. ACM, 2011.

[147] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1083–1094. ACM, 2014.

[148] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. Multise: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 842–853. ACM, 2015.

[149] Corina S Păsăreanu, Neha Rungta, and Willem Visser. Symbolic execution with mixed concrete-symbolic solving. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 34–44. ACM, 2011.

[150] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.

[151] Shiyu Dong, Oswaldo Olivo, Lingming Zhang, and Sarfraz Khurshid. Studying the influence of standard compiler optimizations on symbolic execution. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 205–215. IEEE, 2015.

[152] Jonas Wagner, Volodymyr Kuznetsov, and George Candea. {-OVERIFY}:

Optimizing programs for fast verification. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*, 2013.

[153] Asankhaya Sharma. Exploiting undefined behaviors for efficient symbolic execution. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 727–729. ACM, 2014.

[154] David A Ramos and Dawson Engler. {Under-Constrained} symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, 2015.

[155] Saswat Anand, Corina S Păsăreanu, and Willem Visser. Symbolic execution with abstract subsumption checking. In *International SPIN Workshop on Model Checking of Software*, pages 163–181. Springer, 2006.

[156] Earl T Barr, Thanh Vo, Vu Le, and Zhendong Su. Automatic detection of floating-point exceptions. *ACM Sigplan Notices*, 48(1):549–560, 2013.

[157] Zhongxian Gu, Earl T Barr, David J Hamilton, and Zhendong Su. Has the bug really been fixed? In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 55–64. IEEE, 2010.

[158] Michael D Bond, Nicholas Nethercote, Stephen W Kent, Samuel Z Guyer, and Kathryn S McKinley. Tracking bad apples: Reporting the origin of null and undefined value errors. *ACM SIGPLAN Notices*, 42(10):405–422, 2007.

[159] Benoit Cornu, Earl T Barr, Lionel Seinturier, and Martin Monperrus. Casper: Debugging null dereferences with dynamic causality traces. *arXiv preprint arXiv:1502.02004*, 2015.

[160] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[161] Andreas Zeller. Yesterday, my program worked. today, it does not. why? *ACM SIGSOFT Software engineering notes*, 24(6):253–267, 1999.

[162] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of Software Engineering*, pages 1–10. ACM, 2002.

[163] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, pages 342–351. ACM, 2005.

[164] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information processing letters*, 29(3):155–163, 1988.

[165] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[166] Malay Ganai, Dongyoon Lee, and Aarti Gupta. Dtam: Dynamic taint analysis of multi-threaded programs for relevancy. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[167] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering*, pages 572–583, 2018.

[168] Scott Hanselman. JavaScript is assembly language for the web: Part 2 - madness or just insanity? `http://goo.gl/xH762i`, July 2011. [Online; accessed 14-August-2015].

[169] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2010.

[170] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G Zorn. {JSMeter}: Comparing the behavior of {JavaScript} benchmarks with real web applications. In *USENIX Conference on Web Application Development (WebApps 10)*, 2010.

[171] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do. In *European Conference on Object-Oriented Programming*, pages 52–78. Springer, 2011.

[172] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 736–747, 2012.

[173] Frolin S Ocariza Jr, Karthik Pattabiraman, and Benjamin Zorn. JavaScript errors in the wild: An empirical study. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 100–109. IEEE, 2011.

[174] Frolin Ocariza, Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. An empirical study of client-side JavaScript bugs. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 55–64. IEEE, 2013.

[175] Quinn Hanam, Fernando S de M Brito, and Ali Mesbah. Discovering bug patterns in JavaScript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 144–156, 2016.

[176] Marija Selakovic and Michael Pradel. Performance issues and optimizations in JavaScript: An empirical study. In *Proceedings of the 38th International Conference on Software Engineering*, pages 61–72, 2016.

[177] European Carotid Surgery Trialists' Collaborative Group et al. Randomised trial of endarterectomy for recently symptomatic carotid stenosis: final results of the mrc european carotid surgery trial (ecst). *The Lancet*, 351(9113):1379–1387, 1998.

[178] Asger Feldthaus and Anders Møller. Checking correctness of typescript interfaces for JavaScript libraries. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 1–16, 2014.

[179] Thomas C Chalmers, Harry Smith, Bradley Blackburn, Bernard Silverman, Biruta Schroeder, Dinah Reitman, and Alexander Ambroz. A method for assessing the quality of a randomized control trial. *Controlled clinical trials*, 2(1):31–49, 1981.

[180] Donald T Campbell and Julian C Stanley. *Experimental and quasi-experimental designs for research.* Ravenio Books, 2015.

[181] Mark R Rosenzweig and Kenneth I Wolpin. Natural" natural experiments" in economics. *Journal of Economic Literature*, 38(4):827–874, 2000.

[182] Dejan Baca, Bengt Carlsson, and Lars Lundberg. Evaluating the cost reduction of static code analysis for software security. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 79–88. ACM, 2008.

[183] Ferdian Thung, David Lo, Lingxiao Jiang, Foyzur Rahman, Premkumar T Devanbu, et al. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59. ACM, 2012.

[184] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out

of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

[185] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of Software Engineering*, pages 168–178. ACM, 2011.

[186] Christian Bird and Thomas Zimmermann. Assessing the value of branches with what-if analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 45. ACM, 2012.

[187] JavaScript equality table. `https://dorey.github.io/{J}ava{S}cript-Equality-Table/`, February 2015. [Online; accessed 23-February-2016].

[188] René Just, Darioush Jalali, and Michael D Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[189] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: Bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 97–106. ACM, 2010.

[190] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101. ACM, 2014.

[191] Robert V Krejcie and Daryle W Morgan. Determining sample size for research activities. *Educational and psychological measurement*, 30(3):607–610, 1970.

[192] Sharan Merriam. *Qualitative research: A guide to design and implementation*. John Wiley & Sons, 2009.

[193] David W Martin. *Doing psychology experiments*. Cengage Learning, 2007.

[194] Jacob Cohen. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1):37–46, April 1960.

[195] Kilem Li Gwet. Computing inter-rater reliability and its variance in the presence of high agreement. *British Journal of Mathematical and Statistical Psychology*, 61(1):29–48, 2008.

[196] Barbara Di Eugenio and Michael Glass. The kappa statistic: A second look. *Computational linguistics*, 30(1):95–101, 2004.

[197] G McCray. Assessing inter-rater agreement for nominal judgement variables. In *Language Testing Forum*, 2013.

[198] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[199] Konstantin Weitz, Gene Kim, Siwakorn Srisakaokul, and Michael D Ernst. A type system for format strings. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 127–137. ACM, 2014.

[200] Microsoft. Typescript language specification. `https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md`, January 2016.

[201] Hans F Ebel, Claus Bliefert, and William E Russey. *The art of scientific writing: From student reports to professional publications in chemistry and related fields*. John Wiley & Sons, 2004.

[202] Zhongxian Gu, Drew Schleck, Earl Barr, and Zhendong Su. IDE++ (IDE Plus Plus). `http://marketplace.eclipse.org/content/ide/`, 2013. [Online; accessed 14-August-2015].

[203] Mariano Ceccato, Thomas Roy Dean, Paolo Tonella, and Davide Marchignoli. Data model reverse engineering in migrating a legacy system to Java. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 177–186. IEEE Computer Society, 2008.

[204] Mariano Ceccato, Thomas Roy Dean, Paolo Tonella, and Davide Marchignoli. Migrating legacy data structures based on variable overlay to Java. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(3):211–237, 2010.

[205] Andrea De Lucia, Giuseppe A Di Lucca, Anna Rita Fasolino, Patrizia Guerra, and Silvia Petruzzelli. Migrating legacy systems towards object-oriented platforms. In *1997 Proceedings International Conference on Software Maintenance*, pages 122–129. IEEE, 1997.

[206] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. *ACM SIGPLAN Notices*, 40(10):265–279, 2005.

[207] Frank Tip. Refactoring using type constraints. In *Static Analysis*, pages 1–17. Springer, 2007.

[208] Frank Tip, Robert M Fuhrer, Adam Kieżun, Michael D Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(3):1–47, 2011.

[209] Michael I Schwartzbach and Jens Palsberg. Object-oriented type systems, 1994.

[210] Irene Manotas, Lori Pollock, and James Clause. Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, pages 503–514, 2014.

[211] J Roger Hindley. *Basic simple type theory*. Number 42 in Type theory. Cambridge University Press, 1997.

[212] Ralph Loader. Higher order beta matching is undecidable. *Logic Journal of the IGPL*, 11(1):51–68, 2003.

[213] Barbara H Liskov and Jeannette M Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[214] Joseph A Goguen and José Meseguer. *Security policies and security models*. IEEE, 1982.

[215] Riccardo Focardi, Sabina Rossi, and Andrei Sabelfeld. Bridging language-based and process calculi security. In *Foundations of Software Science and Computational Structures*, pages 299–315. Springer, 2005.

[216] Tao Bao and Xiangyu Zhang. On-the-fly detection of instability problems in floating-point program execution. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 817–832, 2013.

[217] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[218] James R Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[219] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, 2007.

[220] Bob Page. A report on the internet worm. *Risks Digest*, 7(76), 1988.

[221] Donald E Knuth. Notes on the van emde boas construction of priority deques: An instructive use of recursion. *Letter to Peter van Emde Boas*, 1977.

[222] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff. Escape analysis for Java. *Acm Sigplan Notices*, 34(10):1–19, 1999.

[223] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[224] Pinocchio. How does one debug nan values in tensorflow?, 2016. [Online; accessed 04-March-2020].

[225] Robert M Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[226] Jonathan Schaffer. What not to multiply without necessity. *Australasian Journal of Philosophy*, 93(4):644–664, 2015.

[227] Judea Pearl. *Causality*. Cambridge University Press, 2009.

[228] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. Ldx: Causality inference by lightweight dual execution. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 503–515, 2016.

[229] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.

[230] Karen Chan, Andrea Saltelli, and E Marian Scott. *Sensitivity analysis*. Wiley, 2000.

[231] Edwin T Jaynes. *Probability theory: The logic of science*. Cambridge university press, 2003.

[232] Kelly Androutsopoulos, David Clark, Haitao Dan, Robert M Hierons, and Mark Harman. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In *Proceedings of the 36th International Conference on Software Engineering*, pages 573–583. ACM, 2014.

[233] Claude Elwood Shannon. A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423, 1948.

[234] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275. IEEE, 2003.

[235] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40(6):190–200, 2005.

[236] Stuart I Feldman and Channing B Brown. Igor: A system for program debugging via reversible execution. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 112–123, 1988.

[237] MXVMJ Sheldon and Ganesh Venkitachalam Boris Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2007)*, 2007.

[238] Earl T Barr and Mark Marron. Tardis: Affordable time-travel debugging in managed runtimes. *ACM SIGPLAN Notices*, 49(10):67–82, 2014.

[239] Higor A de Souza, Marcos L Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347*, 2016.

[240] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, pages 609–620. IEEE Press, 2017.

[241] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[242] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: A benchmark of JavaScript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101. IEEE, 2019.